Development of a Tree-Based Model for the Fast Generation of Large Point Clouds Representing Particle Showers in Calorimeters

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Moritz Alfons Wilhelm Scham, M. Sc.

aus

Bad Saulgau, Deutschland

Berichter: Prof. Dr. Kerstin Borras
Prof. Dr. Michael Krämer
Prof. Dr. Gregor Kasieczka

Tag der mündlichen Prüfung: 17.01.2025

Abstract

In High-Energy Physics, detailed and time-consuming simulations are needed to describe particle showers in calorimeters. These particle showers are recorded as energy deposits (hits) in the cells of the detector. To mitigate the computational demands of such simulations, surrogate models are widely studied. In this thesis, Generative Adversarial Networks (GANs) are investigated as a fast and flexible approach. If the cells of the detector could be represented by a regular grid, a GAN model would usually use (De-)Convolution layers to up/down-scale the number of voxels. However, due to the often irregular geometry in modern high-granular calorimeters and the small fraction of cells with a hit in such detectors, a grid representation is often not feasible. By representing the shower as a point cloud (PC), i.e., a set of real vectors, these issues can be addressed. In PCs, the complex dependencies between the points must be correctly modeled. Particle showers are inherently tree-like processes, as each particle is produced by the decay or the detector interaction of a particle of the previous generation. With this inductive bias, a GAN has been developed, that generates such PCs in a tree-based manner. For this model, numerous new components for Graph Neural Networks (GNNs) have been developed that allow up/down-scaling of PCs. This model is applied to two popular benchmark datasets, which both can be represented as PCs: JetNet, a dataset containing jet constituents, and CaloChallenge, a dataset containing particle showers in calorimeters. The novel model achieves a good fidelity on both datasets.

Contents

CHAPTER 1

Introduction

1.1. Particle Showers and Generative Models

In particle colliders, such as the Large Hardon Collider (LHC), particles are accelerated and collided at high energies and resulting in the production of heavier and rare, unstable particles. These particles are studied to obtain insights into the features of the interactions and properties of the produced particles and to discover novel, so-far unknown interactions or particles. Depending on the interaction, these collisions may produce a wide variety of particles. These frequently unstable particles may produce additional particles through decays. The collision point is surrounded by a detector that records the energy depositions of the produced particles within the detector material. From these energy depositions, the type and energy of the particles can be reconstructed. Modern detectors frequently feature two major components:

A tracker, situated close to the interaction point, records the trajectory of the particles. The trajectories of charged particles are bent by a magnetic field, allowing to infer the momentum and charge sign from the curvature of the trajectory and the strength of the field.

A calorimeter, situated further away from the interaction point, measures the energy and direction of particles by stopping them. In interactions with the material, the particles can deposit their energy in the calorimeter. A high-energetic particle will produce less energetic particles through detector interactions and decays. These produced particles can then generate further particles themselves. This yields a cascade of particle called a shower. The shower is stopped when the energy of the particles falls below the material-specific threshold, and the energy has been absorbed by the calorimeter almost completely.

The Need for Generative Models for Particle Showers

In particle physics, intricate simulations meticulously replicate the fundamental physics processes, measurement processes, and details of the experimental apparatus. However, accurate simulations of large and complex detectors, especially calorimeters, using current Monte Carlo-based tools such as those implemented in the Geant4 [1] toolkit are computationally intensive. Currently, more than half of the worldwide LHC grid resources are used for the generation and processing of simulated data [2]. Due to the high number of particles in showers, the simulation of calorimeters is responsible for the majority of the run time needed to simulate an event for the LHC experiments, e.g., Atlas [3].

For the upcoming High-Luminosity phase of the LHC (HL-LHC) [4], the computational requirements will exceed the computational resources if no significant speed-ups can be

achieved, e.g., for the CMS experiment by 2028 [5], projecting the current technologies. This will become even more demanding for future high-granularity calorimeters, such as the CMS HGCAL [6], with its complex geometry and extremely high number of channels.

To address these challenges, fast simulation approaches based on Deep Learning, including Generative Adversarial Networks (GANs) [7] and Variational Autoencoders (VAEs) [8], have been explored early [9–14] and deployed recently [15].

1.2. Development of a Tree-based Generative Model

In many calorimeters, the cells within the layers are arranged in a grid, allowing their energy depositions to be represented as a 2D matrix. However, for large calorimeters with irregular geometry, it is more advantageous to represent particle showers as points in space. These so-called Point Clouds (PCs) are particularly efficient when the data is sparse, meaning only few cells have energy depositions.

Particle showers are inherently tree-like processes: They start with a single particle (the root), and further particles (the child nodes) are generated in decays and interactions with the material. Each particle produces its "children" independently of the rest of the shower.

The DeepTree model, developed in this thesis, can produce a PC in a way that follows this tree structure. It starts with a single root node, sampled from noise, and maps this node iteratively to more and more points.

This is the first time that a tree-based generative model is used for the generation of data describing a tree-like physics process.

Generating particle showers with a PC-based model requires the production of a massive number of points and involves complex pre- and postprocessing. Moreover, few established metrics and benchmarks are available. A related task is the generation of jets, where competitive benchmarks are available, the processing is simpler, and the PCs representing the jets can be much smaller. Thus, the fidelity is first established employing jets, and the scaling potential is later demonstrated using a calorimeter dataset.

Milestones

To transform DeepTree from an idea into a working model capable of generating realistic particle showers, the development was divided into the following milestones:

1 PC Upscaling Method
A generator layer must be designed to introduce a new level of nodes into a tree structure, enabling hierarchical PC upscaling.

2 Generator Proof-of-Concept
A proof-of-concept for the generator should be established before further development.

3 Scalable Critic with Iterative PC Downscaling
A critic for PCs should be developed that maintains an efficient runtime even at high cardinalities. PC-based critics often aggregate points in a single step, causing a rapid reduction in the number of elements. A more gradual reduction in the number of elements could improve both convergence and performance. To achieve this, the critic should employ an iterative downscaling method.

4 Fidelity Proof
The model must demonstrate the ability to accurately reproduce the desired distributions.

5 Systematic Quantification of Architecture Choices
The effects of various architectural choices should be systematically quantified to optimize the model's performance. Metrics should be mapped to a common, quantifiable scale.

6 Invertible Transformation of Discrete Cells to Continuous Space
A method should be developed to transform discrete calorimeter cells into a continuous space in an invertible manner.

7 Mitigation Strategy for Multiple Points in the Same Cells
A PC-based model may produce several points that are mapped to the same cell, whereas a cell can have at most one energy deposit. A dedicated mitigation strategy must be developed to avoid these 'multi-hits'.

8 Scaling Proof
The model's ability to handle large PCs, such as those required for representing particle showers, should be demonstrated.

Thesis Structure

This thesis presents the development of a PC-based model named DeepTree. A brief overview of the related physics and ML topics is given in Section 2.2 and Chapter 3, respectively. DeepTree is a GAN consisting of two components: a generator that maps noise to the data space and a critic that separates real data from generated data. The generator's objective is to produce outputs indistinguishable from real data. The model architecture and training process are detailed in Chapter 5 (Milestones 1 and 3).

The model's fidelity is first demonstrated by generating the jet constituents that result from parton hadronization. These constituents naturally form a PC. Subsequently, the model's scaling potential is validated by generating the energy deposits of particle showers, which can also be represented as a PC. The JetNet [16, 17] dataset, which includes jet constituents and accompanying metrics [18], is detailed in Chapter 6. Initially, the 30-constituent subset (JetNet-30) is used for the development of the generator (Chapter 7), employing the critic of another model. By achieving competitive results on JetNet-30, the feasibility of the PC upscaling approach is confirmed (Milestone 2).

Subsequently, the model, including the DeepTree critic, is scaled to JetNet-150 [17], where each jet contains up to 150 constituents (Chapter 8). During this phase, a new scalable critic is developed. With the sensitive metrics of JetNet at hand, the fidelity of DeepTree is demonstrated (Milestone 4).

The effects of various design choices are systematically reviewed in the ablation study presented in Chapter 9, thereby achieving Milestone 5.

Following the demonstration of the model's fidelity on this dataset, its scalability is tested on a calorimeter dataset (Milestone 8). Following this path, an invertible transformation method for calorimeter cells and a mitigation strategy for multi-hits are developed (Milestones 6 and 7). For this purpose, the CaloChallenge dataset (Chapter 10) is utilized.

CHAPTER 2

Showers, Calorimeters, and Jets

Chapter Abstract In this chapter, an overview of calorimeters and the particle showers recorded within them is provided. Additionally, the commonly used simulation approach is highlighted.

2.1. Interactions of Particles with Matter

The type, rate, and effect of particle interactions with matter vary significantly depending on the particle and the material [19, Sec. 6.2]. For the calorimeters discussed later, only electromagnetic and nuclear interactions are relevant. In typical High-Energy Physics experiments, only stable particles, such as photons (γ), electrons/positrons ($e^{\pm}$), charged hadrons (p, $\pi^{\pm}$, $K^{\pm}$), and neutral hadrons (n, K_L^0), can reach the calorimeter [20] and initiate a particle shower by interacting with the material. Within the shower, other, less stable particles may also be present. The dominant type of interaction for a particle changes with its energy. These interactions lead to the particle losing energy as is travels through the material. This material-specific energy loss is referred to as the *stopping power* $-\frac{dE}{dx}$, which describes the energy loss per unit distance.

2.1.1. Charged Particles

Electrons and Positrons

For electrons and positrons, multiple processes contribute to a material's stopping power:

1 Møller/Bhabha scattering
For electrons/positrons at low energies, Møller/Bhabha scattering ($e^-e^- \rightarrow e^-e^-$ / $e^+e^- \rightarrow e^+e^-$) exhibits a noticable contribution to the cross section to the otherwise dominant ionization.

2 Ionization
In ionization, the energy transferred from a free particle traversing matter to an electron bound to an atom or molecule is sufficient to free the electron[1]. Ionization is the most common process over a wide range of energies. Trackers rely on ionization to record the path of a charged particle.

3 Bremsstrahlung

When the electromagnetic field of, for example, an atom, bends the path of a charged particle, photons are radiated. At high energies, bremsstrahlung becomes the dominant contribution to the stopping power of electrons and positrons.

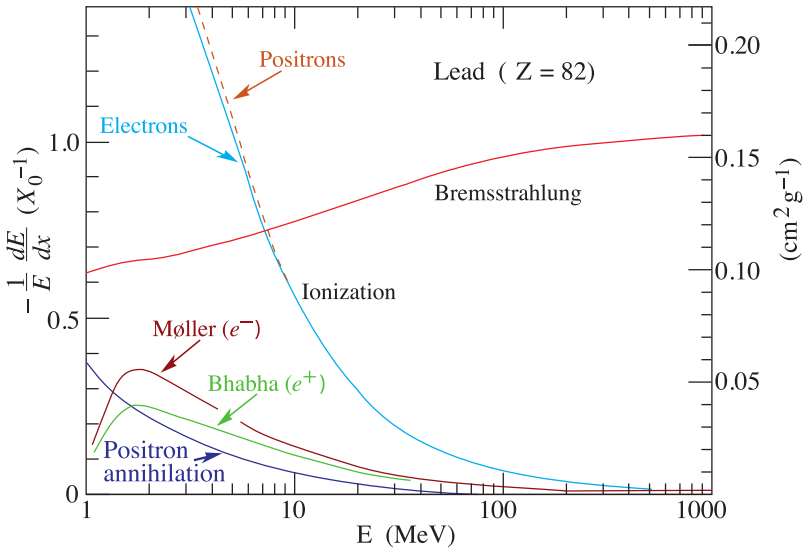


Figure 2.1. | Stopping Power over Energy in Lead as a Function of Electron / Positron Energy.|
See Section 2.1.1 for the discussion. Taken from [21].

Figure 2.1 illustrates the stopping power of lead, divided by the energy of the electron or positron. For electrons at low energies, Møller scattering has the highest cross section, while at higher energies, ionization becomes dominant, followed by bremsstrahlung. In addition to bremsstrahlung, charged particles can also radiate photons through Cherenkov or transition radiation [21, p. 33.7]. Cherenkov radiation occurs when the velocity of the charged particle exceeds the phase velocity of light in the material. This principle is used to measure the energy of particles, for example, in Cherenkov light detectors. Transition radiation occurs when a charged particle transitions between materials with different optical properties. However, typically neither of these two processes contributes significantly to the stopping power in particle detectors.

Muons and Heavy, Charged Particles

In many respects, the muon can be described as a heavier version of the electron. However, the stopping power due to bremsstrahlung is proportional to m^{-4}, significantly reducing its effect for muons compared to electrons. For muons and other charged particles heavier than electrons, such as protons or ions, the Bethe formula [21, Eq. 33.2] describes the energy loss per unit distance:

$$-\frac{dE}{dx} = Kz^2\frac{Z}{A}\frac{1}{\beta^2}\left[\frac{1}{2}\ln\left(\frac{2m_e c^2\beta^2\gamma^2 W_{\max}}{I^2}\right) - \beta^2 - \frac{\delta}{2}\right]. \tag{2.1}$$

[1] "Electron (positron) scattering is considered ionization when the energy loss per collision is below 0.255 MeV, and as Møller (Bhabha) scattering when it is above." [21, p.17]

Here, A / Z / I represents the material's atomic mass / atomic number / excitation energy, m / z denotes the particle's mass / charge, and δ is a term to correct for density effects. K is a constant, and m_e is the electron mass. $W_{\max}$ represents the maximum possible energy transfer to an electron in a single collision. In Fig. 2.2, different contributions to the stopping power for muons in copper are shown. Most notably is the gently increasing slope with a low stopping power starting at 100 MeV and extending to 100 GeV. Particles at the global minimum of this spectrum are referred to as minimum ionizing particles (MIPs). Their characteristic energy deposition [19, Eq. 7.5] of

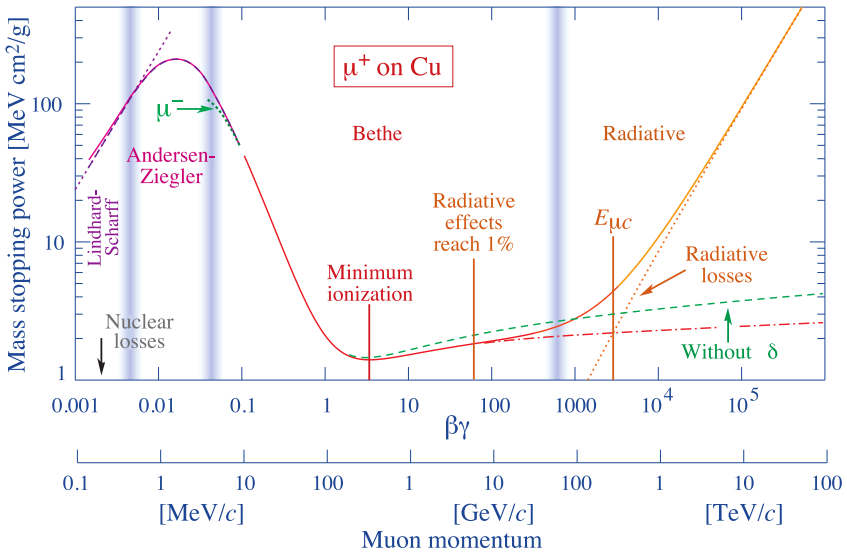


Figure 2.2. | Stopping Power over as a Function of Muon Energy.|
See Section 2.1.1 for the discussion. Taken from [21].

$$-\frac{dE}{dx} = 2.35 - 1.47 \log Z \frac{\text{MeVcm}^2}{\text{g}} \tag{2.2}$$

can be used for the calibration of calorimeter cells.

2.1.2. Photons

For photons, four processes are relevant in their interaction with matter, listed in order of increasing energy [21]:

1 Photoelectric Effect
The photon is absorbed by an atom, which then releases a free electron.

2 Rayleigh Scattering
The photon scatters elastically with an electron, without transferring sufficient energy to free the electron ($\gamma + e^- \to \gamma + e^-$).

3 Compton Scattering
The photon scatters inelastically with an electron, transferring part of its momentum to the electron, which is then freed ($\gamma + e^- \to \gamma + e^-$).

4 Pair Production

In the presence of the strong electric field of a nucleus, the photon is converted into an electron-positron pair ($\gamma \rightarrow e^+e^-$).

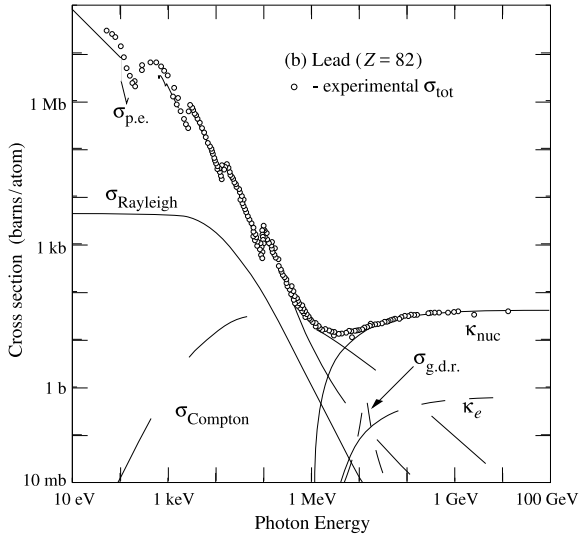


Figure 2.3. | Contributions to the Cross Section of Photons in Lead.|
See Section 2.1.1 for the discussion. Taken from [21].

Figure 2.3 shows the cross section of different photon interactions in lead. While the photoelectric effect ($\sigma_{p.e.}$) dominates at low energies below 100 keV, the Compton effect (σ_{Compton}) is most influential at mid-range energies, and pair production ($\kappa_{\text{nuc}} + \kappa_e$) dominates at energies above 100 MeV.

2.1.3. Electromagnetic Particle Showers

Through many of the described processes, a single high-energy free particle can be converted into two free particles, either by decay, radiation, or by ionizing an atom or molecule. For example, through bremsstrahlung, a charged particle can generate an additional photon, which can then create an e^+e^- pair through pair production. These free particles can continue to generate more free particles. Through these processes, a single charged particle can initiate a cascade of particles, known as a *particle shower*, specifically an *electromagnetic shower* in this context. With each interaction, the average energy of the free particles decreases, increasing the likelihood of being absorbed by the material. As a result, the free particles either exit the material or are absorbed, terminating the shower. The material-specific distance over which an electron's energy is reduced by a factor of e^{-1} is referred to as the *radiation length* X_0. This quantity determines the length of an electromagnetic shower of a given energy for a specific material. The longitudinal and transverse profile of the shower, as well as the number of energy depositions in the calorimeter, are determined by the energy of the shower-initiating particle, and can therefore be used to reconstruct it (see, e.g., Fig. 33.20 in Ref. [21]).

2.1.4. Hadronic Particle Showers

While high-energy charged hadrons lose their energy through electromagnetic processes such as bremsstrahlung and ionization, similar to muons, neutral hadrons interact with the nuclei through the strong interaction [20].

The hadronic processes can be categorized into two components [19, Sec. 6.2.5]:

1 Energetic Component
The interaction produces secondary hadrons ($p, n, \pi^{\pm}, \pi^0 \to \gamma\gamma$) with momenta that constitute a significant fraction of the primary hadron's momentum, typically at the GeV scale.

2 Soft Component
A large portion of the energy is consumed by nuclear processes such as excitation, nucleon evaporation, spallation, etc. This results in secondary particles ($e^{\pm}, \gamma, n$) with characteristic energies on the nuclear MeV scale.

Similar to *electromagnetic showers*, a high-energy hadron will produce a *hadronic shower* through these processes. Due to these complex hadronic and nuclear processes, reconstructing the energy of the shower-initiating particle is more challenging for hadronic showers. Additionally, hadronic showers produce neutrinos, which do not deposit their energy in the calorimeter. Through the production of neutral pions, which decay into photons, a hadronic shower contains an embedded electromagnetic shower. As energy can only be transferred from the hadronic to the electromagnetic part, but not in the other direction, the expected fraction of energy contained in the electromagnetic part increases during the evolution of the shower.

While the datasets studied in this thesis contain only electromagnetic showers, their structure is similar enough from an ML perspective that the results can likely be transferred to hadronic showers [22].

2.2. Calorimeters

Calorimeters [19] are used to measure the energy of particles by recording their energy depositions within a material. These devices are classified into *sampling* and *homogeneous* types. In a sampling calorimeter, alternating layers of *passive* material, which interacts with the particles, and *active* material, where the energy is recorded, are used. The passive material typically has a high atomic number Z to minimize the mean free path, ensuring that the particle deposits its full energy within the calorimeter. In a homogeneous calorimeter, the particle interaction and energy recording occur within the same material.

2.2.1. Types

The method of converting deposited energies into electric signals varies significantly depending on the type of calorimeter used, its position within the detector, the expected radiation dose, the event rate, and even the presence of a magnetic field.

- In scintillators, the incoming particles ionize the atoms, creating electron-hole pairs. When these pairs recombine, photons are emitted and subsequently measured. An

example of this type is the electromagnetic calorimeter at the CMS experiment [23], which uses crystal scintillators. The CMS HGCAL [6], described later in this section, uses plastic scintillators in some parts.

- In silicon detectors, ionized charges generate electron-hole pairs in the depletion region of a silicon diode. Due to the applied electric field, these charges drift before recombining, producing a current pulse that is measured. This method offers several advantages, including high spatial resolution, radiation resistance, and the ability to operate at high event rates. For these reasons, silicon detectors are frequently used in trackers [24]. The CMS HGCAL utilizes this technology.

2.2.2. Energy Resolution

The primary design goal of calorimeters is the best possible energy resolution:

$$\left(\frac{\Delta E}{E}\right)^2 = \underbrace{\left(\frac{a}{\sqrt{E}}\right)^2}_{\text{Sampling}} + \underbrace{\left(\frac{b}{E}\right)^2}_{\text{Noise}} + \underbrace{c^2}_{\text{Constant}} \quad , \tag{2.3}$$

where a, b, and c are detector constants [19, Eq. 6.23]. The *sampling* term arises from statistical fluctuations. In theory, a homogeneous calorimeter can yield the optimal energy resolution, as they are significantly less affected by sampling fluctuations, because the full energy of the particle is deposited in active material. This assumes that the shower is fully contained in the calorimeter. The energy independent *noise* term arises from the noise inherent to the electronics. Lastly, the *constant* term can be created by a multitude of sources, like imperfections in the material or the calibration.

2.2.3. CMS HGCAL

The CMS High Granularity Calorimeter (HGCAL) [25] is the endcap calorimeter designed for the upcoming high-luminosity phase of the Large Hadron Collider (LHC).

The design goals and associated advantages are as follows [25, p. 13]:

1 High Density
The calorimeter is designed with high density to capture the full shower of particles.

2 Radiation Tolerance
The calorimeter must maintain its energy resolution under the high radiation levels expected during the high-luminosity phase.

3 Lateral Granularity (orthogonal to the beam axis)
High lateral granularity improves shower separation, enables the observation of narrow jets, and aids the construction of sharp jet borders, which helps exclude more energy deposits from pileup.

4 Longitudinal Granularity (along the beam axis)
High longitudinal granularity increases the number of ‘slices’ measured, reducing uncertainty from fluctuations. This granularity also enables the use of pattern-matching algorithms for better particle classification and discrimination. Moreover,

since pileup energy is mostly deposited in the first layers, high longitudinal granularity can help reject these contributions.

5 Precision Time Measurement

Precision timing of energy depositions enhances pileup rejection and aids in the reconstruction of the primary vertex.

6 Contribution to Trigger Decision

Contributing to the trigger decision is expected to enhance overall efficiency.

The HGCAL is a sampling calorimeter that employs highly granular silicon cells in regions of high radiation and scintillator tiles further from the interaction point. The resulting complex geometry is shown in Fig. 2.4. To optimize the number of silicon cells produced from a wafer, the wafers are cut into a hexagonal shape and divided into hexagonal cells. The wafers close to the interaction point are particularly granular, with a cell surface area of 0.52 cm^2, while those further away have a surface area of 1.18 cm^2. Moreover, to minimize radiation damage, the thickness varies between 100, 200, and 300 µm. The scintillator tiles are arranged in a grid-like geometry, with tiles that shrink in size towards the interaction point. As passive materials, copper, stainless steel, lead, and a copper-tungsten composite are used.

While the electromagnetic part of the calorimeter utilizes only the silicon cells, the later layers of the hadronic part combine both types of cells. In total, the HGCAL will contain over 6 million cells. This high granularity allows for the precise reconstruction of the primary vertex, enabling the rejection of pileup energy depositions that do not originate from the primary vertex. The high resolution of the HGCAL also allows it to differentiate showers produced by ($\pi^0 \rightarrow \gamma\gamma$) from $e^{\pm}/\gamma$ showers, even for showers occuring close to the beamline. Additionally, the inherent timing capabilities of the silicon sensors will be aiding the rejection of pileup events, the locating of the primary vertex, and assisting in reconstruction.

From a Machine Learning perspective, the HGCAL presents a unique challenge: The combination of hexagonal cells and grid-like tiles results in a highly irregular geometry that cannot be directly represented by a matrix. As a result, most established ML methods, such as those used for image generation, cannot be directly applied. The very high granularity of the HGCAL means repersenting every cell withs energy deposited cell would correspond to a vast output space for any model.

Additionally, simulating a shower in such a highly granular calorimeter would frequently produce a calorimeter image characterized by a high degree of sparsity, with the majority of cells containing no energy deposition. While the approaches discussed in this thesis are applicable to any calorimeter, the assumptions made about the data are specifically motivated by the conditions of the HGCAL during the High-Luminosity phase, scheduled for 2030 [26].

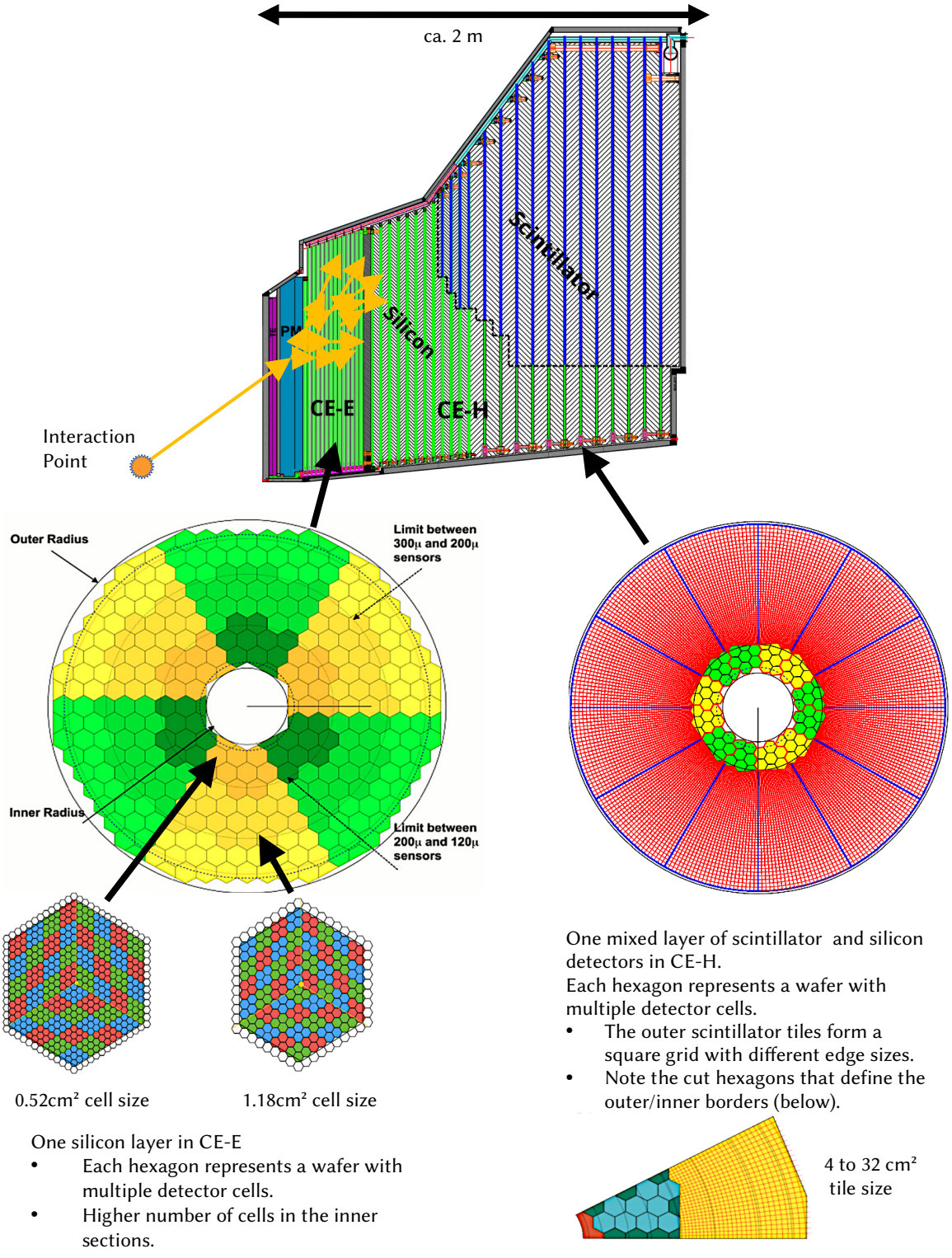


Figure 2.4. | Sketch of the CMS HGCAL.|
See Section 2.2.3 for the discussion. All figures are modified from [6]. The figure at the top shows a longitudinal cut-away view of the calorimeter. For the layers in the electromagnetic part (CE-E) and the first few layers of the hadronic part, hexagonal silicon wafers containing hexagonal silicon cells are used. Wafers closer to the beamline have a higher cell density, as shown in the lower right. This type of layer is depicted at the middle left. Layers further from the interaction point consist of silicon wafers near the beamline and scintillator tiles further away, as shown at the middle right. This results in a complex geometry, particularly at the boundary between the two cell types, as illustrated at the bottom right.

2.3. Calorimeter Simulation with GEANT4

Geant4 [1] is the state-of-the-art framework for simulating particle interactions with matter. A wide range of physics processes, including electromagnetic interactions, hadronic interactions, and decays in various materials, can be modeled using this framework. Geant4 operates on a three-dimensional representation of the targeted object, such as a detector in high-energy physics. Additionally, it supports a wide variety of particles.

The simulation begins with the primary particles, which are assigned specific properties, such as type, energy, and direction, at their origin. Each particle is tracked by Geant4 as it moves through the geometry, step by step. At each step, the probabilities for all possible interactions are calculated based particle properties and the encountered material. The physics process is then determined by sampling according to these probabilities. The distance the particle travels before the interaction occurs is sampled according to the rate of the selected process. Once the particle is shifted to its new position, the corresponding process is evaluated. This evaluation may involve energy loss, scattering, the generation of secondary particles, or absorption. If the particle is neither absorbed nor decayed, the simulation continues to the next step. If the particle crosses a material boundary, the probabilities are recalculated for the new material, and a new process is sampled. The simulation concludes when all particles have either been absorbed or have exited the defined volume.

It should be noted that this simulation can result in multiple energy deposits within a single cell. These *multi-hits* are later combined into a single energy deposit by a separate simulation step.

2.4. Jets

Jets [27] are collimated sprays of hadrons produced by high-energy final state quarks and gluons. Such a parton initiates a cascade of QCD radiations, where one parton produces additional partons (e.g., $g \to q\bar{q}$, $q \to qg$, or $g \to gg$). Once the particle energies drop to the hadronization scale, confinement forces the partons to form hadrons.

A jet algorithm combines the reconstructed particles into a jet based on their position and energy. To obtain jet observables that can be predicted using perturbative QCD, such an algorithm must be invariant under *collinear splittings* and *infrared radiation*. In a collinear splitting, a parton splits into two partons with the same momentum vector direction. Infrared radiation refers to the emission of low-energy gluons at small angles. The anti-k_T algorithm [28] satisfies both requirements.

For this algorithm, the distance between a particle or jet i and j is defined as

$$d_{ij} = \min\left(p_{Ti}^{-2}, p_{Tj}^{-2}\right) \frac{(y_i - y_j)^2 + (\phi_i - \phi_j)^2}{R^2}, \tag{2.4}$$

where p_T is the transverse momentum, y is the rapidity, ϕ is the azimuthal angle, and R is the radius parameter. Based on this distance, the algorithm sequentially combines the jet with the closest particle. Due to the negative exponent of p_T, high-p_T particles are included first. The combination stops, when the "distance to the beam" p_{Ti}^{-2} is smaller than d_{ij} for all particles.

This algorithm is used to construct the jets in the JetNet dataset, which are used in this thesis.

CHAPTER 3

Deep Learning

CHAPTER ABSTRACT In this chapter, an overview of the prevalent Deep Learning methods is given. Specifically, the relevant methods for the DEEPTREE model, developed in this thesis, are introduced. Finally, related models are discussed.

In more and more applications, Machine Learning (ML) models, and specifically Neural Networks (NNs), have displaced classical methods. This is especially true for generative tasks, such as chatbots with GPT-4 [29] or image generation with DALL-E 3 [30].

Neural Networks NNs are compositions of layers, which are typically differentiable functions with adaptable parameters $\boldsymbol{\theta}$. If a NN has many layers, the network is considered *deep*, and the method is referred to as *Deep Learning*. Under certain conditions, NNs are *universal approximators*, meaning that a sufficiently large NN can model any function to arbitrary precision [31]. In the optimization of a NN, called training, the parameters $\boldsymbol{\theta}$ are adjusted to minimize a loss function L.

Inductive Bias As stated in Ref. [32], "[...] we use the *[inductive] term* bias to refer to any basis for choosing one generalization over another, other than strict consistency with the observed training instances." Choosing a model with the right inductive bias can both improve performance on unseen data and aid in searching the solution space, i.e., optimizing the model parameters [33].

Optimization by Empirical Risk Minimization

In *supervised learning*, a model, here a NN, predicts a target variable Y depending on an input variable X with a joint distribution $p_{\mathrm{D}}(X, Y)$. The dimension of Y is typically much smaller than that of X. The training dataset of size n consists of pairs of observations of the input and target variables $D = \{(x_1, y_1), \dots, (x_n, y_n)\}$. The output of the NN, $\hat{y}_i = \mathrm{NN}_{\boldsymbol{\theta}}(x_i)$, is compared to the target y_i using a scalar loss function $L(y_i, \hat{y}_i)$. The expected loss for parameters $\boldsymbol{\theta}$ is referred to as the *risk*.

$$R(\boldsymbol{\theta}) = \mathbb{E}_{(x,y)\sim p_{\mathrm{D}}}[L(x, \mathrm{NN}_{\boldsymbol{\theta}}(y))]$$

The goal of the optimization is to minimize this risk. Because p_{D} is unknown, the risk is estimated on the training dataset as

$$\tilde{R}(\boldsymbol{\theta}) = \frac{1}{n} \sum_{(x,y)\in D} L(y_i, \mathrm{NN}_{\boldsymbol{\theta}}(x_i)). \tag{3.1}$$

The minimization of this quantity is known as Empirical Risk Minimization [34, Sec. 8.1.1]. The inductive bias of the model can allow it to correctly predict the target for samples that follow the p_{D} distribution, even if they are not part of the training sample. This is referred to as the model's ability to *generalize*.

Generative Models In a *generative* task, a model reproduces the distribution of a random variable X, sometimes given a conditioning variable Y. The loss captures how well the generated distribution $p_{\mathrm{G}}(X|Y)$ aligns with the true distribution $p_{\mathrm{D}}(X|Y)$.

Losses

Depending on the task, different loss functions are minimized. In a supervised setting, the goal is to produce an output $\hat{y}$ matching the target y for a given input x. In a regression task, the mean squared error $(y - \hat{y})^2$ is appropriate, while for binary classification, the cross entropy $y \log \hat{y}$ is the natural choice. Generative modeling is a special case because the goal is to produce a distribution that follows the data distribution. Thus, the loss functions quantify how well the generated distribution matches the data distribution.

Chapter Outline First, Feed-Forward Neural Networks (FFNs), which serve as building blocks for many models, are introduced in Section 3.1. The optimization of NNs in general is explained in Section 3.2. NNs typically have a large number of parameters, which makes regularization techniques necessary for their successful optimization (Section 3.3). The attention mechanism, which is at the core of many state-of-the-art models, is presented in Section 3.5. In Section 3.6, ML methods for graph-structured data will be introduced. The generative approach of the DeepTree model is a Generative Adversarial Network (GAN), as detailed in Section 3.4. Different approaches for handling calorimeter data in NNs are summarized in Chapter 4. Lastly, related PC-based GANs, later employed as benchmarks, are presented in Section 4.6.

3.1. Feed-Forward Neural Networks

Feed-Forward Neural Networks (FFNs) [34], or Multilayer Perceptrons, are the fundamental architecture of Deep Learning. Figure 3.1 shows a scheme of such an FFN. They are concatenations of *linear layers* $\mathbf{f}$ and *activation* functions σ:

$$\text{FFN} = \sigma_n \circ \mathbf{f}_n \circ \ldots \circ \sigma_1 \circ \mathbf{f}_1. \tag{3.2}$$

Linear Layers The linear layers provide a linear transformation with a weight matrix $\mathbf{W} \in \mathbb{R}^{o \times i}$ and a bias vector $\mathbf{b} \in \mathbb{R}^o$ as parameters:

$$\mathbf{f}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{3.3}$$

The dimensions of $\mathbf{W}$ and $\mathbf{b}$ depend on the size of the input ($\mathbb{R}^i$) and output vector ($\mathbb{R}^o$).

Activation Functions The activation functions must be non-linear; otherwise, the FFN becomes just one linear operation. Usually, they are applied to each element of the input vector independently. A notable exception is the softmax activation [35], frequently used for classification tasks. Ref. [36] provides a benchmark for the numerous available activation functions.

The most common choices [37] are ReLU [38], GELU [39], Sigmoid [40], Tanh [34, Sec. 6.3.2], and LeakyReLU [41]. In this thesis, LeakyReLU, Sigmoid, and GELU are used:

$$\text{LeakyReLU}(x) = \begin{cases} x & x \geq 0 \\ s \cdot x & x < 0 \end{cases} \quad \text{with parameter } s \in (0, 1), \tag{3.4}$$

$$\text{Sigmoid}(x) = (1 + \exp(-x))^{-1}, \tag{3.5}$$

$$\text{GELU}(x) = x\Phi(x), \tag{3.6}$$

where Φ is the cumulative distribution function of the standard normal distribution.

FFNs are *universal approximators*, capable of approximating any function to an arbitrary degree of precision if they are sufficiently large [31]. While they achieve competitive results on their own in tasks such as classification, they frequently serve as building blocks for more complex models. For the optimization of an FFN, the calculation of the gradient is necessary (Section 3.2). This calculation is demonstrated in detail in Appendix F.

3.1.1. Initialization of Weights and Biases

The initial values of the weight and bias can play an important role for the convergence of a network. Incorrect initialization can lead to exploding or vanishing values, halting the optimization before it begins. The Glorot [42] and He [43] initializations are frequently used. In PyTorch [44], weights and biases are initialized by default by sampling from $U([-\frac{1}{\sqrt{i}}, \frac{1}{\sqrt{i}}])$, where i is the size of the input vector.

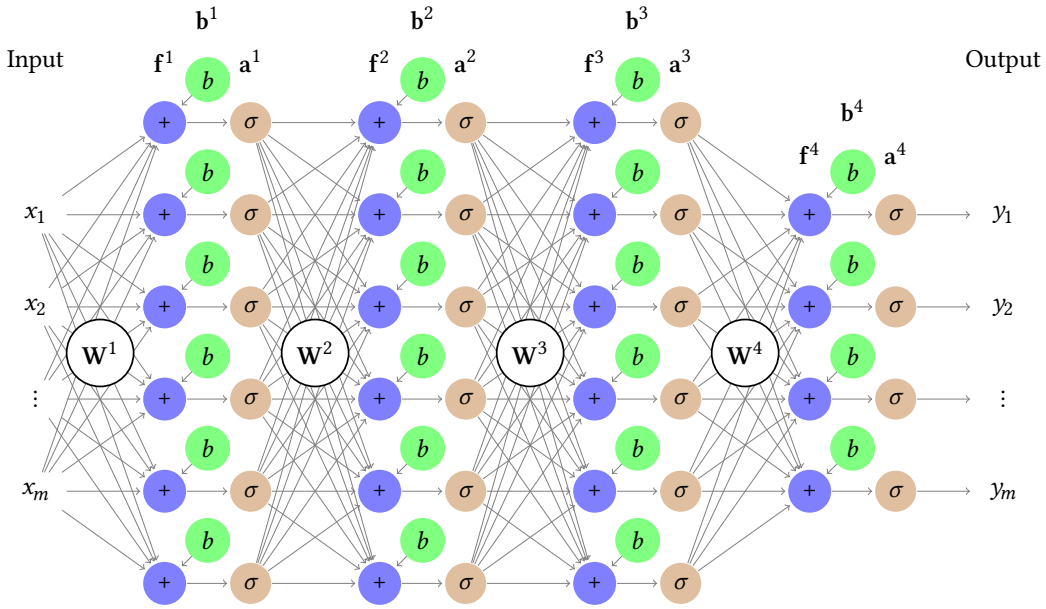


Figure 3.1. | Schematic diagram of an FFN with four Linear Layers.|
The input variables x_i are multiplied with a separate weight $W^1_{i,j}$ for each node j in the first linear layer, represented by the blue "+" node. Each of these nodes sums over the weighted inputs and the bias, represented by the green "b" node. The sum is then passed on to an activation node, represented by the brown "σ" node. The next linear layer takes these activations as input and follows the same procedure with a different weight matrix and bias vector.

3.2. Optimization of Neural Networks

3.2.1. Gradient Descent

The goal of the optimization of NNs is to minimize the empirical risk $\tilde{R}(\boldsymbol{\theta})$ (Eq. 3.1). As the networks are differentiable with respect to their parameters $\boldsymbol{\theta}$ almost everywhere, the gradient descent method [45, Sec. 5.2.4] can be used to minimize this function. In this method, the gradient of the empirical risk, scaled with a learning rate $\eta > 0$, is iteratively subtracted from the parameters:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \tilde{R}(\boldsymbol{\theta}), \text{ where } \nabla_{\boldsymbol{\theta}} = \begin{pmatrix} \partial_{\theta_1} \\ \partial_{\theta_2} \\ \vdots \end{pmatrix}. \tag{3.7}$$

The optimization process that follows this update rule is commonly referred to as *stochastic gradient descent* (SGD). While convergence to a minimum is not mathematically guaranteed without further requirements on $\tilde{R}(\boldsymbol{\theta})$, notably convexity [46, Eq. 8.2], in practice this approach works well for a wide variety of architectures. Calculating the gradient of the empirical risk on the entire training dataset for each update of the parameters is typically inefficient. In the most common approach, the training dataset is divided into smaller subsets called *(mini)batches*. In each step, the gradient of the risk is computed on a (mini)batch and used to update the parameters as described. In an *epoch*, the parameters are updated with each batch once. This method is known as *minibatch gradient descent*.

3.2.2. Backpropagation

NNs are usually compositions of functions $\text{NN} = f_1 \circ f_2 \circ f_3 \ldots$. According to the chain rule, the derivative of the NN with respect to a parameter of such a function contains the derivatives of all preceding functions in the composition. Recomputing these derivatives for each parameter is very inefficient. Backpropagation [40] offers an efficient way to compute these derivatives for deep NNs.

Computational Graph

For this method, the loss and the NN are represented by a *computational graph*[1] [34, Sec. 6.5.1]. In this directed graph, functions, parameters, and inputs are represented as nodes. The edges indicate the direction of the dependencies: Each node is connected to the nodes that depend on it. If an edge connects node a to node b, b is termed the *successor* of a and a is termed the *predecessor* of b. The graph has a *universal sink*; thus, starting from any node and following the edges always leads to a single node, the loss node L. The *sources*, i.e., nodes with outgoing but no incoming edges, are the inputs and parameters of the NN. There are no *isolated* nodes, i.e., nodes without incoming or outgoing edges, in the graph.

Gradient Computation

The computation of the gradients takes place in two stages: In the *forward pass*, the inputs are propagated through the network. The algorithm iteratively computes the values of

[1] For an introduction to graphs in general, see Section 3.6.1.

the nodes, referred to as *activations*, for all nodes where the inputs are available. Thus, it follows the edges of the graph until the loss value is computed. Note that saving these activations greatly increases the memory demand of a model during training compared to evaluation.

During the *backward pass*, the derivative of the loss is iteratively propagated back from the loss node $L(y, \hat{y})$ to the parameters $\boldsymbol{\theta}$. The algorithm starts with the loss node, which is assigned the derivative 1. The backward pass takes place in four steps:

1. The derivatives of the current node are computed with respect to each of the predecessors. They are evaluated at the activation of the respective predecessor that was saved during the forward pass.

2. The computed derivatives are multiplied by the derivative assigned to the node. Through this product, the chain rule is implemented.

3. The derivatives are then assigned to the respective predecessors. If a node has multiple successors, the derivatives are summed.

4. The algorithm continues with the next node that has been assigned a derivative from each of its successors.

In this way, the algorithm follows the edges in the opposite direction until all trainable nodes have been assigned a derivative.

Backpropagation Example

As an example, let f_1, f_2, and f_3 be the functions composing the NN:

$$\text{NN} = f_3(f_2(x_2, \theta_2, f_1(x_1, \theta_1)), f_1(x_1, \theta_1), \theta_3) \tag{3.8}$$

For this NN, the gradient of the loss $L(\text{NN}_{\boldsymbol{\theta}})$ can be calculated as

$$\nabla_{\boldsymbol{\theta}} L = \begin{pmatrix} \partial_{\theta_1} L \\ \partial_{\theta_2} L \\ \partial_{\theta_3} L \end{pmatrix} = \begin{pmatrix} \frac{\partial L}{\partial f_3}\left(\frac{\partial f_3}{\partial f_2}\frac{\partial f_2}{\partial f_1} + \frac{\partial f_3}{\partial f_1}\right)\frac{\partial f_1}{\partial \theta_1} \\ \frac{\partial L}{\partial f_3}\frac{\partial f_3}{\partial f_2}\frac{\partial f_2}{\partial \theta_2} \\ \frac{\partial L}{\partial f_3}\frac{\partial f_3}{\partial \theta_3} \end{pmatrix}. \tag{3.9}$$

Note, that the same terms appear multiple times for different parameters. Now, this gradient should be obtained with backpropagation. Figure 3.2 shows the computational graph for Eq. 3.8.

L Starting with the loss node L, the derivative of the current node is computed for each successor. For L, this is only f_3. First, the derivative $\partial_{f_3} L$ is assigned to the respective node f_3.

f_3 The node f_3 has f_1,f_2, and θ_3 as inputs. The derivatives of f_3 are calculated for each of these inputs ($\partial_{f_1} f_3, \partial_{f_2} f_3, \partial_{\theta_3} f_3$), multiplied with the assigned derivative $\partial_{f_3} L$ and assigned to the f_1, f_2 and θ_3 nodes. Here, θ_3 receives the derivative $\partial_{f_3} L \cdot \partial_{\theta_3} f_3$.

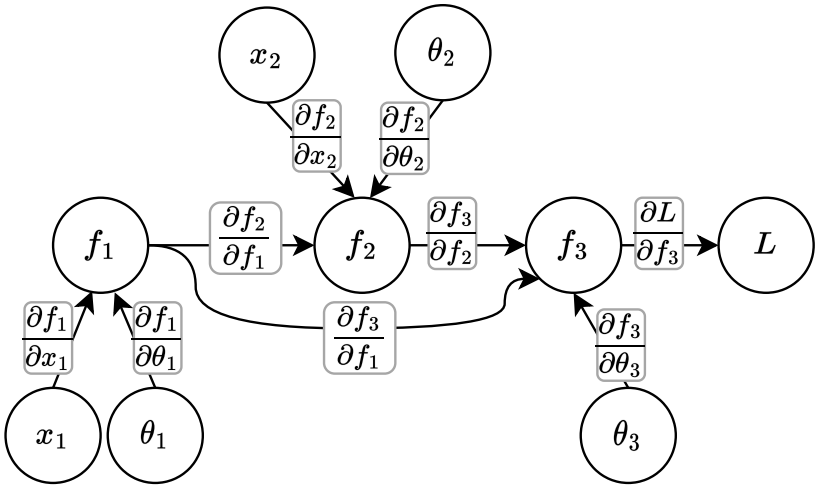


Figure 3.2. | The Computational Graph for the example NN (Eq. 3.8) with the Derivatives for Backpropagation.|
See Section 3.2 for the discussion. Each node depends on its predecessors, i.e., the nodes connected to it. The boxes next to the edges show the partial derivatives of the successors with respect to the predecessor nodes. If a node has multiple successors, the derivatives of the successors are added. The derivative of the loss with respect to a node can be obtained by multiplying all boxes on the path to the loss node.

f_2 The derivatives of f_2 with respect to f_1 and θ_2 are then computed and assigned to the respective nodes. Since the focus is solely on the gradients of the parameters, the derivative with respect to the input x_2 does not need to be calculated. Here, θ_2 receives the derivative $\partial_{f_3} L \cdot \partial_{f_2} f_3 \cdot \partial_{\theta_2} f_2$.

f_1 The nodes f_1 has assigned derivatives from f_2 and f_3, that are summed up to $\partial_{f_3} L \cdot \left(\partial_{f_2} f_3 \partial_{f_1} f_2 + \partial_{f_1} f_3\right)$. The derivative of f_1 is computed for its input θ_1 ($\partial_{\theta_1} f_1$), multiplied with the assigned derivative. This yields the derivative for the parameter θ_1, $\frac{\partial L}{\partial f_3}\left(\frac{\partial f_3}{\partial f_2}\frac{\partial f_2}{\partial f_1} + \frac{\partial f_3}{\partial f_1}\right)\frac{\partial f_1}{\partial \theta_1}$.

3.2.3. The Adam Optimizer

Optimizers, like SGD (Eq. 3.7), provide an update rule for the parameters of the model. Their goal is to achieve fast and stable convergence to an as-low-as-possible minimum. One of the most popular optimizers is Adam [47]. It assigns each parameter a separate, adaptive learning rate. It increases (decreases) the learning rate for a parameter that receives a large (small) gradient with low (high) variance. The update rule (Eq. 3.7) for a parameter θ is modified to:

$$\theta \leftarrow \theta - \eta \frac{\hat{m}}{\sqrt{\hat{v}} + \varepsilon} \partial_\theta \tilde{R}(\boldsymbol{\theta}), \tag{3.10}$$

where η is the original, fixed learning rate and ε is a small value to avoid division by zero. $\hat{m}$ and $\hat{v}$ are approximations for the first and second moments[2] of the gradient for θ. The

[2]The first moment of a random variable is the mean, the second moment is the variance plus the square of the mean.

estimates for the first and second moments, m and v, are initialized to 0 and updated with the derivative at each step:

$$m \leftarrow \beta_1 m + (1 - \beta_1)\partial_\theta \tilde{R}(\boldsymbol{\theta}), \tag{3.11}$$

$$v \leftarrow \beta_2 v + (1 - \beta_2)\left(\partial_\theta \tilde{R}(\boldsymbol{\theta})\right)^2. \tag{3.12}$$

The parameters $\beta_1, \beta_2 \in (0, 1)$ determine how quickly m and v adapt to the most recent gradients; higher values slow down the change. In PyTorch, these values default to 0.9 and 0.999. For a small number of steps, this introduces a bias towards 0 for both moments. This is corrected by scaling the estimates with:

$$\hat{m} = m / \left(1 - \beta_1^t\right) \xrightarrow{t \to \infty} m,$$
$$\hat{v} = v / \left(1 - \beta_2^t\right) \xrightarrow{t \to \infty} v,$$

where t is the number of steps.

3.3. Regularization

A NN may contain billions of parameters that allow it to rapidly adapt to model any function. On the other hand, this flexibility also makes them unstable and susceptible to overtraining, where the model adapts to the specific statistical fluctuations of the training set and does not perform well on other samples. A NN may perform well on the training dataset (low training loss), but worse on an independent validation sample (high validation loss), even if it follows the same distribution as the training dataset. Additionally, deep NNs may suffer from exploding (vanishing) gradients, where the magnitude of the gradient may increase (decrease) as they are propagated through the layers. Both cases can stop the learning process.

Regularization techniques should *restrict* the optimization of the NN to mitigate these problems without degrading performance. In the following section, common regularization techniques that are used in this thesis are outlined.

3.3.1. Early Stopping

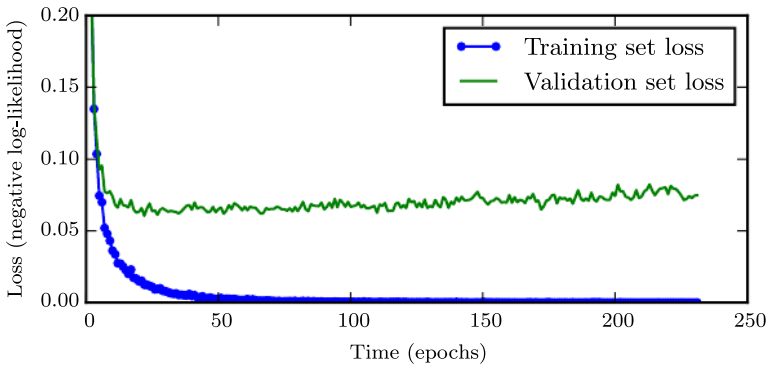


Figure 3.3. | Development of the Training/Validation Loss of an Example Classifier during the Training.|
See Section 3.3 for the discussion. Taken from Ref. [34].

Typically, a NN first approximates the target distribution and shows overtraining only later in the training process. Figure 3.3 shows the typical development of the loss of a NN, in this case a classifier, during the training. The loss is computed on the training set and the validation set. At first, the loss drops rapidly on both samples. At some point, the loss on the validation set stops dropping and starts to rise slowly again (overtraining). By selecting the parameter set at the step with the lowest loss on the validation set, this overtraining can be avoided. This is called *Early Stopping* [34, Sec. 7.8]. As the validation set has been used to select the model configuration, it can no longer serve as an independent test set. Therefore, the dataset must be split into a training, test, and validation set, where the validation set is typically the smallest of the three.

3.3.2. Dropout

Dropout [48] is a technique that effectively turns a densely connected FFN into a more stable ensemble of sparsely[3] connected FFNs. In each training step, a fraction $p \in (0, 1)$ of the weights are temporarily deactivated, i.e., set to zero. When the loss is calculated, the deactivated nodes do not contribute to the activation of the following layer. Therefore, the respective weights and biases remain unchanged during the gradient step. Thus, each gradient step takes place on a different, thinned-out version of the FFN. This reduces the influence of individual weights on the FFN output and limits the adaptation to statistical fluctuations in the training dataset. During inference, dropout is deactivated and the full set of weights is used. While dropout is active, the activations are scaled up by a factor of $(1 - p)^{-1}$ to compensate for the deactivated nodes.

3.3.3. Weight Decay

In weight decay, the gradient $\partial_\theta \tilde{R}(\boldsymbol{\theta})$ in the update rule (Eq. 3.7) is replaced by $\lambda\theta + \partial_\theta \tilde{R}(\boldsymbol{\theta})$, where λ, with $\lambda > 0$, is a parameter. For Adam, this changes the update rule and the estimation of the moments (Eqs. 3.10 and 3.11). In full, the update rule becomes[4]

$$
\begin{aligned}
g &\leftarrow (\lambda\theta + \partial_\theta \tilde{R}(\boldsymbol{\theta})),\\
m &\leftarrow \beta_1 m + (1-\beta_1) g,\\
v &\leftarrow \beta_2 v + (1-\beta_2) g^2,\\
\hat{m} &\leftarrow m / \left(1-\beta_1^t\right),\\
\hat{v} &\leftarrow v / \left(1-\beta_2^t\right),\\
\theta &\leftarrow \theta - \eta \frac{\hat{m}}{\sqrt{\hat{v}} + \varepsilon} g,
\end{aligned}
$$

where t is the number of past gradient steps. With this change, each weight is moved toward zero proportional to its magnitude if there is no opposing gradient present. This limits the magnitude of the weights and can stabilize the training.

As demonstrated in Ref. [49], weight decay and L_2 regularization of the weight ([34, Eq. 5.1]) coincide for SGD but not for Adam, as the weight decay term influences m and v. The authors instead propose a variant of Adam, where the weight decay is decoupled from the calculation of m and v (AdamW).

3.3.4. Input Normalizations for Point Clouds

This section introduces the two approaches for normalizing point clouds: Batch Normalization and Layer Normalization.

Batch Normalization

Batch Normalization [50] scales the mean and variance to 0 and 1, respectively. For this, the mean and variance of each feature are first estimated on a batch and then used to

[3] The sparse connection here refers to some elements in the weight matrix that are fixed to 0. Typically, each node is connected with a trainable weight to each node in the next layer, as shown in Fig. 3.1.

[4] See pytorch.org/docs/2.0/generated/torch.optim.Adam.html, accessed 22.07.2024.

normalize this batch. The batches of PCs have three dimensions: A batch dimension B, a point dimension P, and a feature dimension F. While B and F are fixed, P varies with the cardinality, i.e., the number of points of the PCs within the batch. In the FFNs of the DeepTree model (Chapter 5), the normalization takes place over both B and P. This yields the following algorithm for updating the batch x:

$$
\begin{aligned}
\mu_f &\leftarrow \frac{1}{BP} \sum_{b=1}^{B} \sum_{p=1}^{P} x_{b,p,f} && \forall f, \\
\sigma_f^2 &\leftarrow \frac{1}{BP} \sum_{b=1}^{B} \sum_{p=1}^{P} \left(\mu_f - x_{b,p,f}\right)^2 && \forall f, \\
x_{b,p,f} &\leftarrow \frac{x_{b,p,f} - \mu_f}{\sqrt{\sigma_f^2 + \varepsilon}} && \forall b, p, f, \\
x_{b,p,f} &\leftarrow \gamma_f x_{b,p,f} + \beta_f && \forall b, p, f.
\end{aligned}
$$

ε is a small value to avoid divisions by zero, and γ_f and β_f are trainable parameters. They are initialized with 1 (γ_f) and 0 (β_f). For the DeepTree model, the training of γ_f and β_f has been deactivated. This makes Batch Normalization a more aggressive regularization method, as the means and the variance remain fixed. During training, running estimates of the μ_f and σ_f^2 are computed similarly to $\hat{m}$ and $\hat{v}$ in Section 3.2.3. These estimates are used during evaluation instead of computing μ_f and σ_f^2 on the test/validation batches.

By fixing the scale of the input variables, Batch Normalization can help to avoid exploding or vanishing gradients. Moreover, some activation functions only have an effect if the input distribution is partly positive and partly negative (ReLU and LeakyReLU) or has a small width (standard deviation around 1) around 0 (sigmoid). Thus, Batch Normalization can stabilize the training process and improve performance.

Batch Normalization scales the features in the feature dimension (F) independently by computing the mean and variance along the batch dimension (B) and the point dimension (P). This normalization ensures that all features in a batch are scaled consistently but allows for different scales for each point and each PC.

Layer Normalization

Layer Normalization [51] works similarly to Batch Normalization, but instead of scaling the last dimension (F) and aggregating the first dimensions (B, P), it does the opposite. For PCs, the number of points varies with each PC. Therefore, a scale for each point (P) cannot be computed independent of the batch index (B). Simply put, $\mu_{b,p}$ and $\sigma_{b,p}$ can be computed, whereas μ_p and σ_p cannot. Thus, to apply Layer Normalization to batches of PCs, the mean

and variance must be computed along F for each point in the batch independently.

$$\begin{aligned}
\mu_{b,p} &\leftarrow \frac{1}{F}\sum_{f=1}^{F} x_{b,p,f} && \forall b, p \\
\sigma^2_{b,p} &\leftarrow \frac{1}{F}\sum_{f=1}^{F} \left(\mu_{b,p} - x_{b,p,f}\right)^2 && \forall b, p \\
x_{b,p,f} &\leftarrow \frac{x_{b,p,f} - \mu_{b,p}}{\sqrt{\sigma^2_{b,p} + \varepsilon}} && \forall b, p, f, \\
x_{b,p,f} &\leftarrow \gamma_{b,p} x_{b,p,f} + \beta_{b,p} && \forall b, p, f.
\end{aligned}$$

As each point is scaled independently, the difference in magnitude between the points is removed. This is likely the cause for the observed performance deterioration when applying Layer Normalization to the DeepTree model. Layer Normalization was originally introduced for Natural Language Processing, where the vectors are tokens representing words with no inherent meaning to their magnitude, contrary to PCs.

While other, dedicated normalization techniques for graphs or PCs exist [52], they did not yield better performance for the DeepTree model.

3.3.5. Parameter Normalizations

Instead of scaling the input as in Batch Normalization, Weight Normalization [53] and Spectral Normalization [54] scale the weight matrices of linear layers.

Weight Normalization

In Weight Normalization, $\mathbf{W}$ is split into a normed direction component and a scalar magnitude component g that are optimized separately:

$$\tilde{\mathbf{W}}_{i,j} = g\frac{\mathbf{W}_{i,j}}{\sqrt{\sum_k \mathbf{W}^2_{i,k}}}. \tag{3.13}$$

Thus, each row of the matrix is normalized.

Spectral Normalization

Spectral Normalization [54] scales the weight matrix $\mathbf{W}$ using

$$\tilde{\mathbf{W}} = \frac{\mathbf{W}}{\mathrm{lsv}(\mathbf{W})}, \tag{3.14}$$

where lsv is the Spectral Norm, i.e., the largest singular value, of $\mathbf{W}$:

$$\mathrm{lsv}(\mathbf{W}) = \max_{\|\mathbf{h}\|\neq\mathbf{0}} \frac{||\mathbf{W}\mathbf{h}||}{||\mathbf{h}||}. \tag{3.15}$$

In other words, Spectral Normalization limits the factor by which the normalization of a vector can change through the linear layer to a maximum of 1. Thus, the Lipschitz constant of the matrix is fixed to 1.

Since Parameter Normalizations apply to the parameters rather than the input variables, they need to be recomputed at each step during training but not afterward. Moreover, they can be used simultaneously with Input Normalizations, like Batch Normalization and Layer Normalization.

Application to WGANs

For the Wasserstein GANs (WGANs, Section 3.4.2), a component of the model, the critic, must satisfy the Lipschitz criterion. This can be achieved by normalizing the weight matrices of the linear layers. Initially, Ref. [55] proposed using Weight Normalization [53] for GANs. However, Ref. [56] demonstrated that Weight Normalization often imposes excessive constraints on the matrix and recommended using Spectral Normalization instead. Spectral Normalization introduces a single constraint for the entire matrix, thereby avoiding this issue.

3.4. Generative Adversarial Networks

3.4.1. Introduction

Generative Adversarial Networks (GANs) [7] are a class of generative models. Contrary to, e.g., Variational Autoencoders (VAEs) [8] or Normalizing Flows (NFs) [57], they do not explicitly model the likelihood of the data distribution. A GAN consists of two parts with opposite training objectives: the *generator* G and the *discriminator* D. The generator G maps a noise vector z from a (usually Normal) distribution p_z to the generator distribution p_G in the domain $\mathcal{X}$ of the data distribution p_D. The discriminator D maps from $\mathcal{X}$ to (0,1). The objective of the discriminator is to classify each sample x as either coming from the generator ("fake" = 0) or from the dataset ("real" = 1). In contrast, the generator aims to produce samples that the discriminator classifies as being part of the dataset.

MiniMax Objective In Ref. [7], the following binary cross-entropy (BCE) objective is proposed, which is minimized for the generator and maximized for the discriminator:

$$\min_G \max_D \mathbb{E}_{x\sim p_\mathrm{data}} [\log D(x)] + \mathbb{E}_{z\sim p_z} [\log(1 - D(G(z)))] \,. \tag{3.16}$$

Equivalence of Successful Optimization and Perfect Fidelity For an optimal discriminator $D^\star(x) = \frac{p_\mathrm{D}(x)}{p_\mathrm{D}(x)+p_\mathrm{G}(x)}$, this objective equals

$$\begin{aligned}
&\mathbb{E}_{x\sim p_\mathrm{data}} \left[\log\left(\frac{p_\mathrm{D}(x)}{p_\mathrm{D}(x) + p_\mathrm{G}(x)}\right)\right] + \mathbb{E}_{z\sim p_z} \left[\log\left(\frac{p_\mathrm{G}(G(z))}{p_\mathrm{D}(G(z)) + p_\mathrm{G}(G(z))}\right)\right] \\
&= D_\mathrm{KL}\left(p_\mathrm{D} | \frac{p_\mathrm{D} + p_\mathrm{G}}{2}\right) + D_\mathrm{KL}\left(p_\mathrm{G} | \frac{p_\mathrm{D} + p_\mathrm{G}}{2}\right) - \log(4) \\
&= 2D_\mathrm{JS}(p_\mathrm{D}|p_\mathrm{G}) - \log(4)
\end{aligned}$$

The Kullback-Leibler divergence D_KL and the Jensen-Shannon divergence D_JS are defined as

$$D_\mathrm{KL}(P|Q) = \mathbb{E}_{x\sim P} \log\left(\frac{P(x)}{Q(x)}\right) = \sum_{x\in X} P(x) \log\left(\frac{P(x)}{Q(x)}\right), \tag{3.17}$$

$$D_\mathrm{JS}(P|Q) = \frac{1}{2} D_\mathrm{KL}(P|Q) + \frac{1}{2} D_\mathrm{KL}(Q|P), \tag{3.18}$$

for the probability distributions P and Q on the sample space X. The D_JS is non-negative and zero if, and only if, $p_\mathrm{D}(x) = p_\mathrm{G}(x)\ \forall x \in \mathcal{X}$. Thus, if this objective is minimized, the distribution is perfectly reproduced.

Alternating Optimization The model is optimized by alternately training the generator and the discriminator. The objective (Eq. 3.16) can be decomposed into separate loss functions for the generator and the discriminator:

$$L_D = -\mathbb{E}_{x\sim p_\mathrm{data}}[\log D(x)] - \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))], \tag{3.19}$$

$$L_G = \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))]. \tag{3.20}$$

Non-saturating Loss The objective would yield $\log(1 - D(G(z)))$ as the generator loss, but the authors noted that this loss might not provide a sufficient gradient: "Early in learning, when [the generator] is poor, [the discriminator] can reject samples with high confidence because they are clearly different from the training data. In this case, $\log(1 - D(G(z)))$ saturates. Rather than training [the generator] to minimize $\log(1 - D(G(z)))$, we can train [it] to maximize $\log D(G(z))$." [7] This yields the following loss terms, referred to as non-saturating loss:

$$L_D = -\mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] - \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))], \tag{3.21}$$

$$L_G = \mathbb{E}_{z\sim p_z}[\log D(G(z))]. \tag{3.22}$$

3.4.2. Wasserstein GAN

Since their invention, multiple variants have been introduced to improve the stability of GANs. In Ref. [58], a "Wasserstein GAN" (WGAN) is introduced. It is motivated by Optimal Transport [59]. The generator should now be optimized to minimize the Wasserstein-1 distance W_1 (also called Earth Movers distance):

$$W_1(p_{\text{D}}, p_{\text{G}}) = \inf_{\gamma\in\Pi(p_{\text{D}},p_{\text{G}})} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|],$$

where $\Pi(p_{\text{D}}, p_{\text{G}})$ is the set of all joint distributions with marginals p_{D} and p_{G}. The authors demonstrate that the W_1 distance — contrary to, e.g., the D_{JS} — provides a usable gradient, even if p_{D} and p_{G} have a disjoint support. The Kantorovich-Rubinstein duality [59] relates it to the supremum over all 1-Lipschitz functions that map from the sample space to $\mathbb{R}$:

$$W_1(p_{\text{D}}, p_{\text{G}}) = \sup_{\|f\|_L\leq 1} \mathbb{E}_{x\sim p_{\text{D}}}[f(x)] - \mathbb{E}_{x\sim p_{\text{G}}}[f(x)].$$

The authors propose to approximate the W_1 distance by optimizing a parametric 1-Lipschitz function called *critic* $C : \mathcal{X} \to \mathbb{R}$, effectively replacing the discriminator[5]. The objective of the critic is to map the real / fake samples to $+\infty/-\infty$. This yields the following loss functions:

$$L_C = -\mathbb{E}_{x\sim p_{\text{data}}}[C(x)] + \mathbb{E}_{z\sim p_z}[C(G(z))], \tag{3.23}$$

$$L_G = -\mathbb{E}_{z\sim p_z}[C(G(z))]. \tag{3.24}$$

Enforcing the Lipschitz Criterion

To enforce the Lipschitz continuity on the critic, its weights are restricted to a compact space. In Ref. [58], the interval $[-0.01, 0.01]$ is chosen. It is noted that this 'weight clipping' might have significant downsides, such as vanishing gradients if the interval is too small, or a long training time if the interval is too large. Moreover, this restriction might lead the weights to take values on the boundary of the interval [60]. In the following paragraphs, other options to enforce the Lipschitz continuity are presented.

[5] A NN that takes the role of the discriminator while mapping to $\mathbb{R}$ instead of (0,1) is referred to as a critic.

Gradient Penalty In Ref. [60], the loss of the critic is expanded by the frequently used gradient penalty:

$$\lambda \, \mathbb{E}_{\hat{x}\sim\hat{\mathcal{X}}} \left[(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2 \right], \tag{3.25}$$

with $\hat{x} = \alpha x + (1-\alpha)\tilde{x}$, $x \sim p_{\mathrm{D}}$, $\tilde{x} \sim p_{\mathrm{G}}$, and $\alpha \sim U[0,1]$. In other words, the critic is penalized if its gradient deviates from 1. For this, the normalization of the gradient is evaluated on a sample $\hat{x}$ that is randomly interpolated between a real and a fake sample. Thus, the gradient is limited along the path from p_{G} to p_{D}. The hyperparameter λ controls the influence of this loss term. To fulfill the Lipschitz continuity, only gradients with a magnitude greater than 1 (instead of $\neq 1$) would need to be penalized. However, the authors argue that this is not a major constraint on the critic, as the optimal critic has gradients with normalization 1 almost everywhere in p_{D} and p_{G} [60, Corollary 1]. If the data space $\mathcal{X}$ contains more complex structures than vectors, the interpolation between the samples from p_{D} and p_{G} becomes challenging. This is especially true in the case of PCs, as they are unordered and have a varying size. To produce an interpolated PC, one would not only need to find a way to pair the points from the two PCs but also a way to handle the differing cardinality[6]. While not impossible, this is certainly a major obstacle for applying the gradient penalty to PC-based GANs.

Weight Normalization & Spectral Normalization Instead of penalizing the gradient of the critic for a random sample, one can design the critic as a Lipschitz continuous function over the entire input space. A composition $f \circ g$ of Lipschitz continuous functions f, g (with Lipschitz constants L_f, L_g) is Lipschitz continuous itself, as

$$\|f(g(\mathbf{x}_1)) - f(g(\mathbf{x}_2))\| \leq L_f \|g(\mathbf{x}_1) - g(\mathbf{x}_2)\| \leq L_f L_g \|\mathbf{x}_1 - \mathbf{x}_2\|.$$

Thus, if the critic is designed as a composition of Lipschitz continuous functions, it is Lipschitz continuous itself. In the case of FFNs, this means choosing a Lipschitz continuous activation function, e.g., ReLU, and limiting the weights of the linear layers.

Weight Normalization and Spectral Normalization (Section 3.3.5) both rescale the weight matrices $\mathbf{W} \in \mathbb{R}^{n\times m}$ of each linear layer in the network. Moreover, both normalizations enforce the Lipschitz criterion on the rescaled matrix $\tilde{\mathbf{W}}$: In Weight Normalization (Eq. 3.13), the Lipschitz criterion is fulfilled with constant $|g|\sqrt{m}$:

$$\begin{aligned} \|\tilde{\mathbf{W}}(\underbrace{\mathbf{x}_1 - \mathbf{x}_2}_{\Delta})\| &= \sqrt{\sum_i^n \left(\sum_j^m \tilde{\mathbf{W}}_{ij} \Delta_j \right)^2} \\ &= \sqrt{\sum_i^n \left(g \sum_j^m \frac{\mathbf{W}_{i,j}}{\sqrt{\sum_k \mathbf{W}_{i,k}^2}} \Delta_j \right)^2} \end{aligned}$$

[6] The PC-based GANs in Section 4.6 and the DeepTree model developed in this thesis sample the cardinality from the training dataset to produce a PC of this size. During training, the respective PC could be sampled alongside the cardinality to compute the gradient penalty. However, this still leaves the problem of matching the points between the generated and the simulated PC.

By Cauchy–Schwarz inequality:

$$\leq |g| \sqrt{\sum_i^n \left(\left(\sum_j^m \underbrace{\left(\frac{\mathbf{W}_{i,j}}{\sqrt{\sum_k \mathbf{W}_{i,k}^2}} \right)^2}_{\leq 1} \right) \left(\sum_j^m \Delta_j^2 \right) \right)}$$
$$\leq |g| \sqrt{m} ||\Delta||.$$

In Spectral Normalization, the weight matrix $\mathbf{W}$ is rescaled to $\tilde{\mathbf{W}}$ using the largest singular value of $\mathbf{W}$ (Eq. 3.14). Thus, the Lipschitz criterion is fulfilled by

$$||\tilde{\mathbf{W}}\Delta|| = ||\frac{\mathbf{W}}{\max_{||\mathbf{h}||\neq \mathbf{0}} \frac{||\mathbf{W}\mathbf{h}||}{||\mathbf{h}||}} \Delta|| = \frac{||\mathbf{W}\frac{\Delta}{||\Delta||}|| \cdot ||\Delta||}{\max_{||\mathbf{h}||\neq \mathbf{0}} \frac{||\mathbf{W}\mathbf{h}||}{||\mathbf{h}||}} \leq ||\Delta||.$$

Weight Normalization introduces one constraint for each row of the matrix and may lead the critic to consider only a single feature [56]. Therefore, Spectral Normalization, which does not have this problem, is recommended for GANs.

3.4.3. Least Squares and Hinge Objectives

In Refs. [61, 62], new training objectives are proposed to address the issues of vanishing and exploding gradients. With the cross-entropy loss (Eq. 3.21), a sample that is distributed similarly to the data distribution receives a vanishing loss [61][7]. Instead of using the logarithm in the critic's objective, the L_2 distance to the target of 1/0 for real/fake samples is recommended in Ref. [61]:

$$L_C = \frac{1}{2} \mathbb{E}_{x\sim p_{\text{data}}}[(C(x)-1)^2] + \frac{1}{2} \mathbb{E}_{z\sim p_z}[C(G(z))^2], \tag{3.26}$$
$$L_G = \frac{1}{2} \mathbb{E}_{z\sim p_z}[(C(G(z))-1)^2]. \tag{3.27}$$

This adjustment ensures that the loss remains limited for $C(G(z)) \to 0$, while samples far from the target still have a significant influence on the gradient.

In Ref. [62], the Hinge loss has been proposed for GAN training:

$$L_C = -\mathbb{E}_{x\sim p_{\text{data}}}[\min(0, -1 + C(x))] - \mathbb{E}_{z\sim p_z}[\min(0, -1 - C(G(z)))], \tag{3.28}$$
$$L_G = -\mathbb{E}_{z\sim p_z}[C(G(z))]. \tag{3.29}$$

The loss matches the Wasserstein loss, but for the minimum expressions in the critic loss. These terms break the gradient for real/fake samples that have reached 1/-1. This offers two advantages: First, it prevents the critic from running to $\pm\infty$. Secondly, it halts the training of the critic for correctly separated samples. If the optimization were to continue, the critic would become more and more certain of its assignment and would likely provide a worse gradient to the generator. Instead of continuing to increase the separation between the samples, the generator is given a chance to 'catch up' to the critic.

[7] A critic output of $D(G(z)) = 0.99$ results in a loss of $L_G \approx 0.01$, while an output of $D(G(z)) = 0.01$ results in a loss of $L_G \approx 4.6$.

3.4.4. Advantages and Disadvantages

The major advantage of GANs is their performance: GANs have significantly advanced the state-of-the-art in image generation, with notable examples such as DCGAN [63], StyleGAN [64], and CycleGAN [65]. Additionally, the GAN approach is highly flexible: In principle, any differentiable parametric function can be used as a discriminator or generator if it maps to and from the correct domain. Moreover, GANs are typically fast in data generation. On the other hand, GANs also come with significant disadvantages:

- By design, GANs do not provide access to the likelihood, unlike VAEs or NFs. This also makes their evaluation more challenging. During training, metrics need to be computed on the generator output to track the progress of the training.
- GANs frequently suffer from *mode collapse*, where the generator fails to produce the full support of the distribution. For example, in a GAN trained on a dataset containing handwritten digits 0 to 10 (like MNIST [66]), the GAN might produce only sevens. This can be explained by the following, simplified learning process: Typically, a discriminator can easily distinguish whether a sample is within the support of the distribution. Thus, the generator first learns to produce samples from (a subset of) the support of the distribution [67]. In the next step, the discriminator learns the data distribution. It penalizes, i.e., assigns a value closer to 0 to samples from regions in the phase space where $p_\mathrm{D} < p_\mathrm{G}$ and vice versa. Over a series of training steps, the generator adapts and aims to reproduce the data distribution. However, if the generator learns to produce similar samples classified as data early in the training, it is incentivized to reproduce only these similar samples. If the discriminator fails to sufficiently penalize repeated generations of these samples, the generator may learn to trick each successive generation of the discriminator instead of actually learning the data distribution. This way, the discriminator is locked in a local minimum and continues producing the same value [68].
- The generator is trained by differentiating the discriminator output with respect to the generator weights. Through the product rule, each additional layer after the layer of a given weight adds a factor to the gradient of that weight. In networks with many layers, like GANs, this may lead to exploding or vanishing gradients [69, 70]. Vanishing gradients are especially problematic for GANs, as they usually change the dimension of the data, making it difficult to implement residual connections that could mitigate this problem.
- A major challenge with GANs is their generally unstable training process, even when mitigation strategies like Wasserstein loss or spectral normalization are applied.

In summary, the most significant advantage of GANs is their fidelity, while their most significant drawback is their notoriously difficult and unstable training [67].

Recently, there has been a shift from GANs to diffusion models, largely due to the latter's even better fidelity and more stable training processes. On the other hand, when it comes to data generation, a diffusion model is typically much slower than a comparable GAN.

3.5. Attention Mechanism

The most central building block in many generative state-of-the-art models [29, 30] is the transformer shown in Fig. 3.4. It is built on the Attention Mechanism [71], more specifically the "scaled dot product attention" and the *Multi-HeadAttention* constructed from it. It was proposed in the context of natural language processing, where it assigns an attention weight, i.e., a measure of importance, between the elements of two sequences of words A and B with lengths L and S. The mechanism is formulated in terms of a query $Q \in \mathbb{R}^{L\times d_k}$, a key $K \in \mathbb{R}^{S\times d_k}$, and a value $V \in \mathbb{R}^{S\times d_v}$.

$$\text{Attention}(Q,K,V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \in \mathbb{R}^{L\times d_v}, \tag{3.30}$$

where the softmax is applied to the rows of the matrix:

$$\text{softmax}(A)_{i,j} = \frac{\exp A_{i,j}}{\sum_{k=0}^{n} \exp A_{k,j}} \text{ for } A \in \mathbb{R}^{n,m}. \tag{3.31}$$

Note that K and V have the same length and thus can be produced by embedding the same sequence into d_k and d_v respectively. Q and K, on the other hand, both need to be embedded into d_k but may have differing lengths.

Instead of computing a single attention at a time, the authors also introduce *Multi-HeadAttention*, where the attention function is evaluated for h different linear mappings of Q, K, and V to a common dimension d_{model}. The resulting h vectors are concatenated and mapped to $\mathbb{R}^{L\times d_v}$ with a trainable matrix:

$$\text{Multi}-\text{HeadAttention}(Q,K,V) = \underbrace{\text{Concat}\left(\text{head}_1,\ldots,\text{head}_h\right)^T}_{\in\mathbb{R}^{L\times(h\cdot d_v)}} \underbrace{W_0}_{\in\mathbb{R}^{(h\cdot d_v)\times d_v}} \in \mathbb{R}^{L\times d_v}, \tag{3.32}$$

where

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V), \tag{3.33}$$

where the $W_i^Q, W_i^K \in \mathbb{R}^{d_k\times d_{\text{model}}}$ and $W_i^V \in \mathbb{R}^{d_v\times d_{\text{model}}}$.

The attention mechanism is mainly employed in two ways: as *CrossAttention* to combine two sequences, or as *SelfAttention* to update a sequence with itself. With mappings f^Q, f^K, and f^V that embed the sequences into d_{model}, the CrossAttention for the two sequences is defined as

$$\text{CrossAttention}(A,B) = \text{Multi}-\text{HeadAttention}(f^Q(B), f^K(A), f^V(A)) \in \mathbb{R}^{L\times d_v}. \tag{3.34}$$

For a single sequence A, SelfAttention is defined as

$$\text{SelfAttention}(A) = \text{Multi}-\text{HeadAttention}(f^Q(A), f^K(A), f^V(A)) \in \mathbb{R}^{L\times d_v}. \tag{3.35}$$

SelfAttention and CrossAttention are the central building blocks in the transformer architecture.

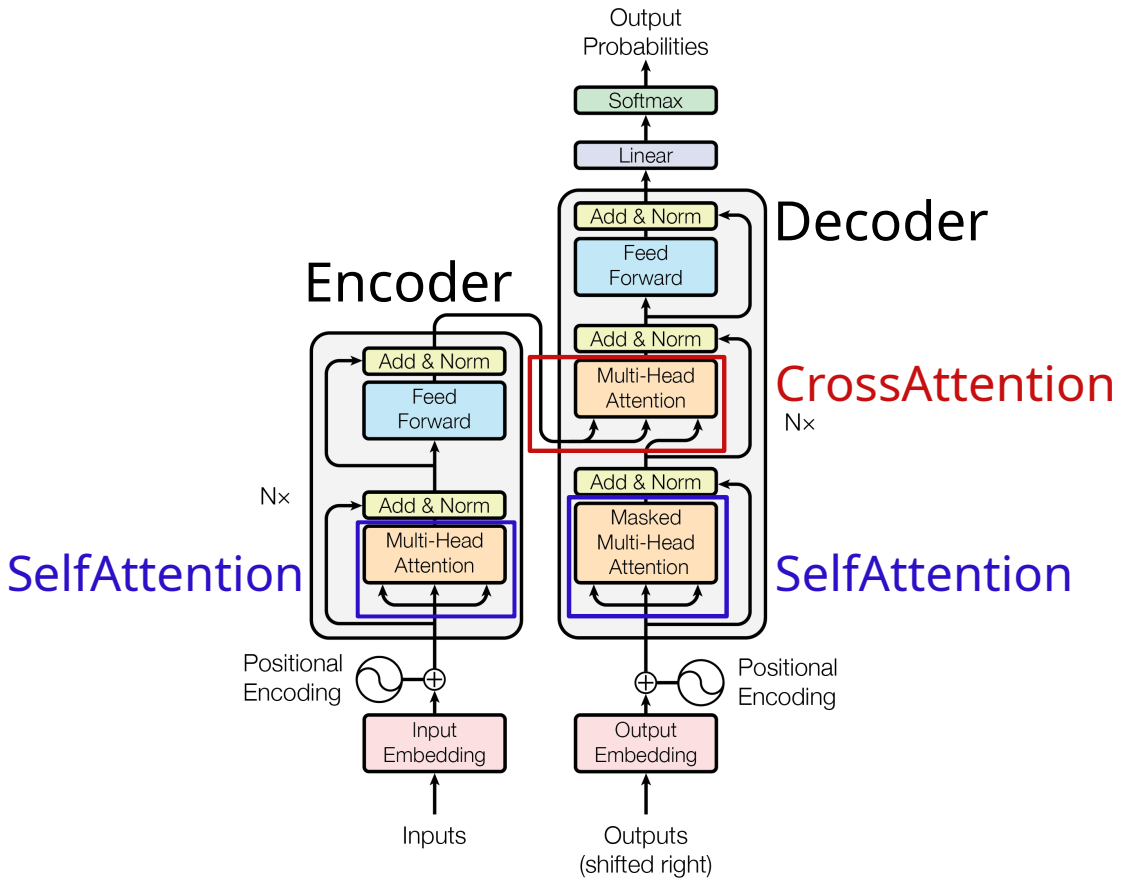


Figure 3.4. | The Transformer Architecture.|
See Section 3.5 for the discussion. Adapted from Ref. [71].

Example Task: Translation Using Transformers

Figure 3.4 shows the transformer architecture. It consists of two parts: the encoder, which encodes a sequence, and the decoder, which transforms a different sequence using the encoder input. The input to the encoder is referred to as the input sequence, while the input to the decoder is referred to as the output sequence, as shown in Fig. 3.4. The decoder produces an output vector for each element in the output sequence, depending on the encoded input sequence. For a translation task, the input sequence represents the sentence that should be translated.

Text Representation To be processed by a transformer, words need to be mapped to numbers. First, the words are converted to tokens, which may represent words, syllables, or characters. The two *vocabularies* provide two bijective mappings between the tokens and integer indices for both languages. Each of the indices is represented by a specific vector. With the tokens now represented by vectors, a positional encoding is added to inform the model about the position of the token in the sequence.

Processing Steps Both the embedded input and the output sequence are updated with Self-Attention. For the encoder block, the input sequence is then updated with an FFN.

The updated input sequence is fed into the Cross-Attention of the decoder as K and V. The output sequence enters the Cross-Attention as Q, thus producing a sequence with the same length as the output sequence. The resulting sequence is passed through an FFN, completing the decoder block. The encoder and the decoder blocks are repeated several times. Each time, the encoder output is passed to the respective decoder's Cross-Attention.

Output & Loss Finally, each vector of the decoder's output sequence is mapped to the space of possible tokens with a linear layer. Softmax is applied to the new vectors to produce the probability for each token. From these probabilities and the target tokens, the cross-entropy loss can be computed.

Training The transformer is trained and evaluated by providing the encoder with the full input sentence and the decoder with a part of the output sentence. The sentences are framed with *start* and *stop* tokens, which indicate the beginning and end of the sequence. The decoder should predict the next token of the translated sentence. During training, multiple output sequences are produced from a single translated sentence: In the first step, the output sequence consists only of the *start* token. Afterward, the tokens of the translated sentence are appended one by one in the following steps. In each step, the output sequence is passed through the decoder. The resulting probabilities of the last token in the output sequence are compared to the next token in the translated sentence, and a loss is computed from the difference. In the final step, the output sequence is the start token plus all tokens of the translated sentence, and the loss is computed from the difference to a *stop* token.

Evaluation During the evaluation, the output sequence starts as a single start token. In each evaluation step, the output sequence is passed through the transformer with the fixed input sequence. The decoder produces a probability vector for each of the tokens in the output sequence, but only the last one is used to predict the next token. This token is then appended to the output sequence. These evaluation steps continue until a stop token is predicted.

Transformers for Point Clouds

Transformers were originally designed for ordered sequences, such as sentences, which contrasts with the unordered nature of PCs focused on in this thesis. However, the attention functions consist of permutation equivariant operations (Section 4.5). In fact, the positional encoding is necessary to break the equivariance for transformers [71, Sec. 3.5]. Therefore, these functions are well-suited for PC-based models. The run time of Self-Attention, however, scales quadratically with the number of inputs and thus is not applicable to large PCs. The later described MDMA model (see Section 4.6.3) employs Cross-Attention to a central node to implement an attention mechanism with linear scaling.

A transformer can process multiple sequences/PCs of varying size simultaneously by padding them to the size of the largest sequences/PCs and masking the additional tokens/-points. However, this becomes increasingly computationally inefficient with larger cardinalities. The graph attention in Section 3.6.5 translates the attention mechanism to graphs and PCs by formulating it as a message-passing step (Section 3.6). This graph attention is used for the critic of the DeepTree model developed in this thesis (Section 5.2).

3.6. Graph Neural Networks

The introduction given in this section largely follows H. William's book [72] on Graph Neural Networks (GNNs).

3.6.1. Graphs, Point Clouds, and Trees

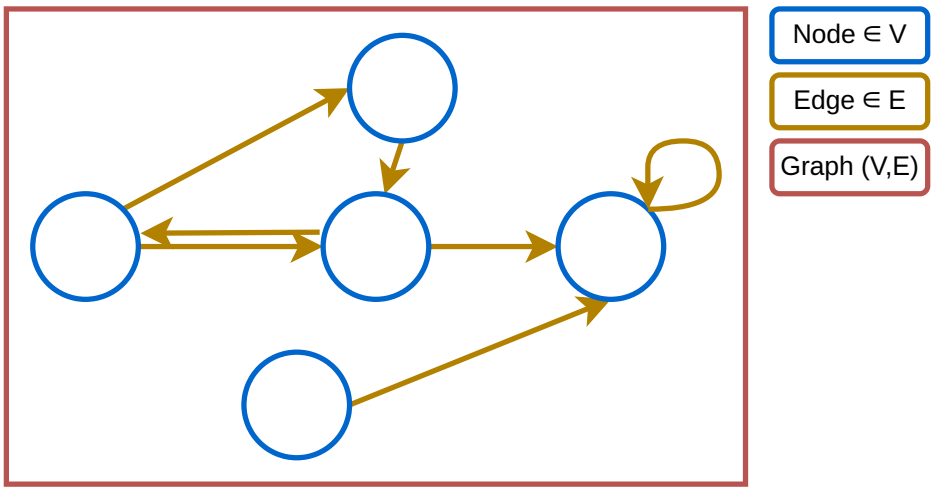


Figure 3.5. | Example Graph.|
See Section 3.6.1 for the discussion.

Figure Fig. 3.5 shows a sketch of an example graph. A graph $G = (V, E)$ consists of a set of nodes V and a set of edges $E \subset V \times V$ that connect the nodes to each other. An edge is a tuple $(u, v) \in E$ connecting the source node u to the target node v. The set of edges E is frequently represented by the adjacency matrix:

$$A_{i,j} = \begin{cases} 1 & (i,j) \in E \\ 0 & (i,j) \notin E \end{cases} \tag{3.36}$$

Note that this representation is not unique as the nodes are unordered.
A graph is *undirected* if, and only if, for each edge, there exists another edge pointing in the opposite direction:

$$(u, v) \in E \iff (v, u) \in E. \tag{3.37}$$

Connections from a node to itself are referred to as *self-loops*. If there exists a series of edges in E with $[(u, s_1), (s_1, s_2), ..., (s_{k-2}, s_k), (s_{k-1}, v)]$, then u is connected to v by a *path* of length k (called a "k hop path"). The *distance* from u to v is the length of the shortest path.

The incoming/outgoing *degree* $d^{\text{in/out}}$ of a node u is the number of incoming/outgoing edges:

$$d^{\text{in}}(u) = \#\{(s,t) \in E : t = u\}, \tag{3.38}$$

$$d^{\text{out}}(u) = \#\{(s,t) \in E : s = u\}. \tag{3.39}$$

For undirected graphs, d^{in} and d^{out} coincide. For this thesis, the *neighborhood* $\mathcal{N}$ of a node is the set of nodes that are connected **to** it:

$$\mathcal{N}(u) = \{s : (s, u) \in E\}. \tag{3.40}$$

The number of nodes is called the *cardinality* $C = |V|$, and the number of edges the *edge count*. The *feature matrix* $\mathbf{X} \in \mathbb{R}^{|V| \times d}$ for $d \in \mathbb{N}$ contains the *feature vectors* of the nodes as rows. The feature vector of the node with index i is denoted by $\mathbf{x}_i$. In this context, the nodes are used to refer to the associated feature vectors.

Point Clouds

In this thesis, PCs are defined as graphs without edges, containing nodes with real feature vectors of the same dimension[8]. Most GNNs rely on edges to transfer information between nodes. Therefore, to use GNNs with PCs, edges need to be added. This can be done, for example, by constructing edges using the k-NN algorithm [73]. Further approaches are discussed in Section 4.6.

Trees

A tree is a graph (Section 3.6.1) with a specific set of connections. Each edge in this graph connects a *parent* to one of its *children*. Nodes without children are called *leaves*. All nodes have one parent, except for the *root* node, which has no parent. The *ancestors* of a node are all nodes on the path between the root and the respective node, including its parent, its parent's parent, and so forth. The root node is an ancestor to all nodes except itself. The *distance* between two nodes is defined as the number of edges on the shortest path between them. The distance to the root node equals the number of ancestors minus one. The ith *level* of a tree consists of all nodes that are at a distance i from the root node.

Challenges in Applying Deep Learning to Graphs

Graphs are versatile data structures capable of representing diverse structures, ranging from molecular structures [74] to social networks [75]. Grids and sequences can also be represented by graphs, with edges connecting neighboring nodes on the grid or sequential nodes in a sequence. However, this flexibility comes at a cost. Many established Deep Learning techniques, such as Convolutional Neural Networks [76] for grid-based data and Recurrent Neural Networks [40] for sequences, cannot be directly applied to graphs. Additionally, graphs present challenges with varying cardinalities and node degrees, complicating their representation as matrices. This often excludes the use of the matrix operations, which are integral to many commonly used Deep Learning techniques. Consequently, Deep Learning on graph-structured data requires not only specialized architectures but also a specialized set of algorithms to effectively process these structures [77].

Adapting GNNs to Varied Graph Types and Tasks

The structure of a graph varies significantly depending on the type of data it represents. For instance, a graph representing molecules may have only a few nodes, whereas a graph representing a social network may have millions. A graph depicting emails between users may feature multiple edges between two nodes. On the other hand, a graph representing points on a surface measured by a LiDAR (Light Detection and Ranging) sensor may have no edges at all. This diversity yields a multitude of different tasks, such as classifying or regressing a value for an edge, node, or graph, or predicting the existence or properties

[8] Note that the most commonly used definition of PCs requires the feature vectors to be of dimension 3, representing Cartesian coordinates. This requirement is not imposed in this thesis.

of unknown edges or nodes. Additionally, the processing of datasets can differ: In a node classification task involving many small graphs, batches of graphs are created, and a GNN is applied to each batch. In contrast, a social network dataset may consist of a single, massive graph. For processing with a GNN, this graph may need to be divided into subgraphs. This large variance in graph structures, tasks, and representations necessitates a large variety of GNN architectures tailored to address specific problems.

3.6.2. Message Passing

The dominant Deep Learning paradigm for graphs is *Message Passing* [72, 77, 78] (also known as Neural Message Passing). This method describes an update rule for the nodes of the graph, composed of three steps:

1. For each edge $\mathbf{e}_{j,i}$ connecting a source node $\mathbf{x}_j$ to a target node $\mathbf{x}_i$, messages $\mathbf{m}_{i,j}$ are computed using a message function ϕ.
2. The messages are then aggregated for each target node with a permutation-invariant aggregation function $\bigoplus$, typically a sum or mean function.
3. Finally, with the aggregated messages $\bigoplus_{j \in \mathcal{N}(i)} \mathbf{m}_{i,j}$ and the target node $\mathbf{x}_i$, an update function γ computes the new value of $\mathbf{x}_i$.

This update rule can be written as:

$$\mathbf{x}_i \leftarrow \underbrace{\gamma(\mathbf{x}_i, \underbrace{\bigoplus_{j \in \mathcal{N}(i)} \underbrace{\phi(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{j,i})}_{\text{Message}}}_{\text{Aggregate}})}_{\text{Update}}. \tag{3.41}$$

The functions γ, ϕ, and $\bigoplus$ must be differentiable and are chosen based on the specific Message Passing Layer (MPL). Depending on the MPL, γ, ϕ, or both may contain trainable parameters.

For an even more general Message Passing approach, where the edges and graph features are updated as well, see Ref. [72, Section 5.6]. In practice, MPLs function as the elementary transformation for GNNs, similar to Linear Layers for FFNs (Section 3.1). After an MPL, each node encodes information about the nodes in its direct neighborhood $\mathcal{N}$ (1 hop), but not about the neighbors of its neighbors (2 hop). To encode information from node v in node u, one MPL is needed for each hop distance between u and v. This algorithm therefore focuses on *local* features, i.e., features of nodes with a small distance to the target node. To combine information from distant nodes, many MPLs are needed, often leading to over-smoothing, where nodes become more and more similar to each other [72, Sec. 5.3]. To counteract this over-smoothing, multiple mitigation strategies, such as skip connections and gated updates, have been developed [72, Sec. 5.3.1-5.3.2].

3.6.3. Graph Convolutions

In convolutions, two functions f and h are combined into a single function by computing the integral over the product and shifting the argument of h with the integration variable:

$$(f \star h)(\mathbf{x}) = \int_{\mathbb{R}^d} f(\mathbf{y}) h(\mathbf{x} - \mathbf{y}) \mathrm{d}\mathbf{y}.$$

For functions on a finite set S, the discrete convolution is defined as:

$$(f \star h)(i) = \sum_{j \in S} f(j)h(i-j).$$

In Convolutional Neural Networks (CNNs) [79], the input matrix $f \in \mathbb{R}^{n_1 \times n_2}$, representing a picture, and the 2D filter $h \in \mathbb{R}^{m_1 \times m_2}$ are interpreted as functions that map from their index to the value at the given position. f and h map values outside the range of their dimensions to 0 ("zero padding"). With that, their discrete convolution can be computed as:

$$(f \star h)(i,j) = \sum r, s \in \mathbb{N} f(r,s)h(i-r, j-s).$$

For images, on which CNNs usually operate, the NN should be constructed in a translation-invariant manner so that the extracted features are independent of their position in the image. Thus, CNNs benefit from the translation equivariance of the convolutions, e.g., for a single dimension

$$(f \star h)(i+a) = \sum_{r \in \mathbb{N}} f(r)h(i+a-r) = \sum_{r' \in \mathbb{N}} f(i+a-r')h(r'),$$

or equivalently with "$\cdot$" as the argument placeholder

$$(f \star h)(\cdot + a) = f(\cdot) \star h(\cdot + a) = f(\cdot + a) \star h(\cdot).$$

Note, that because of the zero padding, the summation variable can be changed without changing the bounds. This result demonstrates that shifting the filter, the image, or the convolution is equivalent. If the translation equivariant convolution is followed by a translation invariant operation, e.g., maximum pooling, the result is translation invariant. See Ref. [80] for a general discussion.

The convolution theorem states that the convolution can be computed by the element-wise product $\odot$ of the Fourier transforms:

$$(f \star h) = \mathscr{F}^{-1}\left(\mathscr{F}(f) \odot \mathscr{F}(h)\right).$$

By defining a Graph Fourier transformation $\mathscr{GF}$, the convolutions can be extended to graphs. Here, f and g are functions that assign a value to each node in the graph. The Laplacian $\mathbf{L}$ of an undirected graph is defined as

$$\mathbf{L} = \mathbf{D} - \mathbf{A},$$

where $\mathbf{A} \in \{0,1\}^{|V| \times |V|}$ is the adjacency matrix and $\mathbf{D}$ the degree matrix diag $((d(1), ..., d(|V|)))$. This Laplacian can be decomposed into the orthogonal matrices $\mathbf{U}$, containing the eigenvectors u_i as columns, and $\mathbf{\Lambda}$, containing the eigenvalues on the diagonal Λ_i:

$$\mathbf{L} = \mathbf{U}^\top \mathbf{\Lambda} \mathbf{U}.$$

The Graph Fourier transformation is then defined by a projection on the eigenvectors:

$$\begin{aligned} \hat{\mathbf{f}} = \mathscr{GF}[\mathbf{f}] &= \mathbf{U}^\top \mathbf{f} &&= \sum_i f(i) u_i^\top, \\ \mathbf{f} = \mathscr{GF}^{-1}[\hat{\mathbf{f}}] &= \mathbf{U}\hat{\mathbf{f}} &&= \sum_i \hat{f}(\Lambda_i) u_i. \end{aligned}$$

These operations are inverse because of the orthogonality of $\mathbf{U}$. This allows for the definition of a graph convolution for a 'signal' $\mathbf{f}$ and the 'filter' $\mathbf{h}$:

$$(\mathbf{f} \star \mathbf{h}) = \mathbf{U}(\mathbf{U}^\top \mathbf{f} \odot \mathbf{U}^\top \mathbf{h}).$$

By representing the filter $\mathbf{h}$ by its Graph Fourier coefficients $\boldsymbol{\theta} = \mathbf{U}^\top \mathbf{h}$, the convolution can be rewritten as

$$\begin{aligned}(\mathbf{f} \star \mathbf{h}) &= \mathbf{U}(\mathbf{U}^\top \mathbf{f} \odot \boldsymbol{\theta}),\\ &= (\mathbf{U}\,\mathrm{diag}\,(\boldsymbol{\theta})\,\mathbf{U}^\top)\mathbf{f}.\end{aligned}$$

The first model using graph convolutions is constructed in Ref. [81]. For a graph with $|V|$ nodes of size C, the node matrix $\mathbf{X}$ has dimensions $|V| \times C$. This convolution, where $\mathbf{X}$ takes the place of the signal, is used to update $\mathbf{X}$:

$$\mathbf{X}_i \leftarrow \sigma(\mathbf{X} \star \mathbf{h}) = \sigma\left(\mathbf{U} \sum_{k=1}^{C} \mathrm{diag}\,(\boldsymbol{\theta})_{k,i}\, \mathbf{U}^\top \mathbf{X}_k\right) \;\forall i \in \{1, ..., |V|\},$$

where $\boldsymbol{\theta}$ contains the coefficients of the filter $\mathbf{h}$ and σ is an activation function. The arrow denotes that the value on the left is replaced with the value on the right. In Ref. [81], it is proposed to restrict the Laplacian to the lowest d eigenvalues and set the other columns in $\mathbf{U}$ and other entries in $\boldsymbol{\theta}$ to zero. However, this operation remains computationally expensive because the decomposition of the Laplacian needs to be computed, and the gradient must be backpropagated through it. Additionally, it is noted that the removed eigenvectors may contain meaningful information about the graph.

In Ref. [82], this is developed into the much simpler **GCNConv** layer:

$$\mathbf{X} \leftarrow \sigma(\tilde{\mathbf{A}}\mathbf{X}\boldsymbol{\theta}),$$

where $\tilde{\mathbf{A}} = (\mathbf{D} + \mathbb{1})^{1/2}(\mathbb{1} + \mathbf{A})(\mathbf{D} + \mathbb{1})^{1/2}$ is a normalized version of the adjacency matrix $\mathbf{A}$ with added self-loops, σ is an activation function, and $\boldsymbol{\theta}$ is a trainable matrix. It is shown that this GCNConv layer can be derived as a first-order approximation of the previously introduced filters. Moreover, it can be expressed as an MPL:

$$\mathbf{x}_i \leftarrow \boldsymbol{\theta}^\top \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{\mathbf{x}_j}{\sqrt{d(i)d(j)}} \tag{3.42}$$

The neighbors of the target node are directly used as messages and normalized with the geometric mean of the degrees of the source and target nodes. This prevents nodes with high degrees from causing numerical instabilities and dominating the graph. Additionally, self-loops are added to the graph.

3.6.4. GINConv

Instead of using a simple, linear mapping like GCNConv, **GINConv** [83] uses an FFN as the update function. Moreover, the self-loops are scaled with a trainable weight ε.

$$\mathbf{x}_i \leftarrow h_{\boldsymbol{\theta}}\left(\mathbf{x}_i \cdot (1 + \varepsilon) + \sum_{j \in \mathcal{N}(i)} \mathrm{ReLU}(\mathbf{x}_j)\right)$$

The authors prove that a GNN built from GINConv layers can match the power of the Weisfeiler-Leman graph isomorphism test ([72, Sec. 7.3.3]), but a GNN built from GCNConv layers cannot. This MPL is used in the generator of the DeepTree model (Section 5.1) introduced later.

3.6.5. Graph Attention Layers

The attention mechanism (Section 3.5), and the transformer architecture constructed with it, has become the dominant building block of NNs for Natural Language Processing and Computer Vision. In Ref. [84], this attention mechanism is reformulated for GNNs by the "Graph ATtention layer" (**GATConv**). This layer uses the update rule

$$\mathbf{x}_i \leftarrow \sum_{j \in \mathcal{N}(i) \cup \{i\}} \boldsymbol{\alpha}_{i,j} \boldsymbol{\theta}_t \mathbf{x}_j. \tag{3.43}$$

Attention is now used to weigh the source node(s) j (key and value) to the target node i (query). For two neighboring nodes, the score of an edge (i, j) is defined as

$$e_{i,j} = \text{LeakyReLU}\left(\mathbf{a}_s^\top \boldsymbol{\theta}_s \mathbf{x}_i + \mathbf{a}_t^\top \boldsymbol{\theta}_t \mathbf{x}_j\right). \tag{3.44}$$

The attention weight $\boldsymbol{\alpha}_{i,j}$ of source node j to target node i is computed by scaling the score $e_{i,j}$ with the softmax over the neighborhood of i:

$$\boldsymbol{\alpha}_{i,j} = \underset{\mathcal{N}(i)\cup\{i\}}{\text{softmax}}\, e_{i,j} = \frac{e_{i,j}}{\sum_{k \in \mathcal{N}(i)\cup\{i\}} e_{i,k}}. \tag{3.45}$$

The attention vectors $\mathbf{a}_s$, $\mathbf{a}_t$ and the matrices $\boldsymbol{\theta}_s$, $\boldsymbol{\theta}_t$ are trainable parameters of the network.

Additionally, the authors also offer an extension to multiple (K) attention heads and propose averaging the output:

$$\mathbf{x}_i \leftarrow \frac{1}{K} \sum_{h=1}^{K} \sum_{j \in \mathcal{N}(i) \cup \{i\}} \boldsymbol{\alpha}_{i,j}^h \boldsymbol{\theta}_t^h \mathbf{x}_j. \tag{3.46}$$

Static Attention Problem The weight of a neighbor j scales with the scalar product of a trainable attention vector and a linear mapping of its feature vector $\mathbf{a}_t^\top x_j$. In Ref. [85], it is noted that the scale, but not the ranking (the ordering by size), of the attention weights $\boldsymbol{\alpha}_{i,j}$ depends on the target node $\mathbf{x}_i$. Therefore, a global ranking of nodes exists. This defeats the purpose of the attention mechanism, where the weights should indicate, which node is important for which other node. Instead, in the **GATv2Conv** layer, the calculation of the edge scores is modified to

$$e_{i,j} = \mathbf{a}^\top \text{LeakyReLU}\left(\boldsymbol{\theta}_s \mathbf{x}_i + \boldsymbol{\theta}_t \mathbf{x}_j\right), \tag{3.47}$$

with $\mathbf{a}$ as a trainable vector and $\boldsymbol{\theta}_s$, $\boldsymbol{\theta}_t$ as trainable matrices. To maximize the attention weight, the projection of the source node $\boldsymbol{\theta}_t \mathbf{x}_j$ may now shift the projection of the target $\boldsymbol{\theta}_s \mathbf{x}_i$ in the direction of the attention vector $\mathbf{a}$.

Modification to the Attention Weights In the update rule for GATConv/GATv2Conv, the projection of the source node $\boldsymbol{\theta}_t \mathbf{x}_j$ appears twice, directly and indirectly in the calculation of the attention weight (Eqs. 3.43 and 3.47). Instead of using $\boldsymbol{\theta}_s \mathbf{x}_i + \boldsymbol{\theta}_t \mathbf{x}_j$ in the calculation of $\boldsymbol{\alpha}_{i,j}$, the **GATmConv** variant employed in Section 5.2 uses $\boldsymbol{\theta}_s \mathbf{x}_i - \boldsymbol{\theta}_t \mathbf{x}_j$. In this way, a message $\boldsymbol{\theta}_t \mathbf{x}_j$ that shifts $\boldsymbol{\theta}_s \mathbf{x}_i$ away from the attention vector $\mathbf{a}$ will have a higher, instead of a lower, attention weight.

Chapter Review In this chapter, the ML background for the DeepTree model is provided. First, simple FFNs and the fundamental methods for optimization and regularization are introduced (Sections 3.1 to 3.3). GANs, the generative approach followed by DeepTree, are detailed in Section 3.4. The frequently used attention mechanism is described in Section 3.5. As will be detailed in the following chapter, it is often advantageous to represent the showers as PCs, which are graphs without edges. The NNs that operate on these graphs or PCs, along with the extension of the attention mechanism to graphs, are detailed in Section 3.6.

CHAPTER 4

Handling Calorimeter Data in Generative Models

Chapter Abstract The data produced by calorimeters varies significantly in structure. This chapter provides an overview of the approaches used to handle this data in generative models. Additionally, generative models related to the DeepTree model are presented.

4.1. Data Representation

A collection, such as an array or vector, is considered *sparse* if many of its elements contain zero values. In the context of a calorimeter, a shower manifests as energy depositions in the calorimeter cells. Here, the *sparsity* refers to the fraction of cells with no energy deposition.

Two primary methods are used to represent the calorimeter's state in vector form:

1 Fixed-Size Position Representation

In this method, the position of an element within the vector corresponds to the specific cell where energy is deposited. The vector's size is determined by the total number of calorimeter cells. When cells are distributed along each coordinate independently, the calorimeter follows a *regular grid* structure. In this case, the calorimeter can be represented as a multidimensional matrix, where each cell is accessible via a tuple of indices. An example of this is provided by the CaloChallenge dataset 2/3, which features a cyclical geometry, as discussed in Chapter 10.

2 Variable-Size Index Representation

This method uses a variable-sized set of tuples. Each tuple contains a cell index and the associated energy deposition. Cells without energy deposition are not represented. This yields a dense representation of otherwise sparse data, leading to vectors of varying sizes. In scenarios with high sparsity, the memory required for the index representation can be orders of magnitude smaller than that of the position representation. When cell indices are converted into coordinates, the set transforms into a PC, as will be discussed in the following.

4.2. Neural Networks for Regular Calorimeters

For calorimeters that can be represented by a regular grid, models frequently employ convolutions [79], e.g., Refs. [86–89]. In image processing, convolutions are commonly used to capture features within an image. A filter, which is a trainable matrix, is passed over the image. For each position of the filter, its values are multiplied with the values of the pixels below it and then summed up, producing one value for each position. Similarly, convolutions can be applied to pixels or voxels that represent the cells of a regular calorimeter. The size of the filter determines the run time, the number of parameters, and the extent of the distance between pixels it can combine. Convolutions are inherently local and translation-invariant, which are helpful biases for images and calorimeter images.

Recently, the "Vision Transformer" [90] approach has gained popularity in image processing [91]. In this approach, an image is divided into patches of fixed dimensions. These patches are then converted into a sequence of tokens and provided to a transformer. This method has only recently been adapted for calorimeter simulation [92].

However, neither of these approaches is directly applicable when the calorimeter geometry is irregular.

4.3. Neural Networks for Irregular Calorimeters

Most existing approaches represent particle showers in the detector using pixels or voxels. Especially for high-granularity calorimeters like the CMS HGCAL [6], the calorimeter cells are highly irregular, and the data is often sparse. A review of existing methods for modeling calorimeters with irregular geometries reveals the following approaches:

Dense Mapping A trainable matrix is used to map from the position representation to a latent space. In this method, neighboring and distant cells are connected in the same way, introducing no helpful inductive bias for the model. The number of parameters is the number of cells times the size of the latent space. Thus, for calorimeters with many cells, this method is likely to fail due to the large matrix size. However, for smaller calorimeters and sufficiently large datasets, this may be a viable option, as used by, e.g., Refs. [89, 93, 94].

1D Convolutions Assuming the calorimeter is represented by a vector in the position representation, 1D convolutional filters can be applied to this vector. A major advantage over dense mapping is that the number of parameters scales with the size of the filter rather than the number of cells. However, with this (arbitrary) ordering, pairs of cells that are equidistant in space might have different distances in their indices. This largely defeats the purpose of convolutions, which aim to identify repeating patterns as they move over the input. Moreover, to combine information about cells to each other with large index differences, an equally large filter would be needed. Thus, the use cases for this approach are very limited. This approach is employed by, e.g., Refs. [94, 95].

Superfine Grid Instead of simulating the irregular geometry of the calorimeter, one can simulate the calorimeter using a regular, superfine grid. The resulting calorimeter image can be approached with regular convolutions, thus benefiting from their inductive

bias. To map the generated superfine calorimeter image to the real calorimeter, the superfine cells must be merged into real cells. The difficulty and speed of this combination, especially for irregular geometries and a high number of cells, remain unclear. A major downside is that the superfine calorimeter image might be much larger than the actual calorimeter, which, in turn, might be significantly larger than a dense representation like the index representation. This becomes especially problematic for high granularity calorimeters that already contain numerous cells.

Learnable Geometry Mapping In Ref. [96], the "Geometry Latent Mapping" (GLaM), a learnable mapping between the irregular calorimeter and a regularized version of the calorimeter, is introduced. The method achieves state-of-the-art fidelity on the CaloChallenge dataset 1 (Section 2.2). However, constructing such a regularized version of the calorimeter might be more difficult for other calorimeters, and the scaling behavior to calorimeters with more cells is unclear. For a more detailed discussion, see Appendix E.1.

Vector Quantization The vector-quantized VAE (VQVAE) [97] introduces a way to discretize the latent space of a VAE. On this discrete latent space, transformers can be applied. While this changes the representation of the shower to a sequence of tokens, the encoder and decoder rely on one of the other data handling approaches and therefore face the same issues and restrictions regarding their applicability. For a more detailed description, see Appendix E.2.

Point Clouds The approach chosen for the DeepTree model are PCs, which are discussed in detail in the following section.

4.4. Point Clouds

PCs are a natural representation for particle showers, which are essentially energy depositions at specific points in space, i.e., in the calorimeter cells. A PC is an unordered set of vectors with the same size. To represent a particle shower as a PC, vectors containing the hit energy and the position of the respective cell are collected in a set. An index representation, can easily be converted to a PC by replacing the cell index with the cell's position, as described at the beginning of this section. PCs have several properties that introduce specific advantages and disadvantages (labeled with '▲' and '▼') for the modeling of particle showers.

Dense Representation for Sparse Data In PCs, cells without an energy deposition are not represented, making them a dense representation for the sparse calorimeter data. This has multiple upsides:

- ▲ Like the index representation, PCs are memory efficient if only a small fraction of cells contains hits, i.e., if the sparsity is high.
- ▲ This smaller size also means that the output of the model can be smaller by orders of magnitude compared to a sparse representation. In general, a model with a smaller output size is easier to train and faster to evaluate.

- ▲ In a sparse representation, most elements do not carry any information, effectively diluting the information content. A generative model trained on such a representation must first learn to ignore these irrelevant elements before learning to reproduce the meaningful ones. Furthermore, it may be very challenging for a model to produce an energy deposition in one cell without energy depositions in its neighboring cells. These problems are avoided with a dense representation, like a PC.

Calorimeter Independent Using a PC representation makes the data structure, and therefore the model, independent of the geometry of the calorimeter. This has multiple effects:

- ▲ Showers of *any* calorimeter can be represented.
- ▲ The same model architecture can be applied to any calorimeter. Thus, a model pretrained on one calorimeter could be transferred to another without the need to repeat the entire training process. This opens up the possibility of developing a foundation model for calorimeter simulation.
- ▼ In a grid-based model, certain properties of the calorimeter, such as the number of layers or the number of cells, are intrinsic to the model's architecture. In contrast, a PC-based model has no inherent information about the calorimeter. Therefore, the model must learn the calorimeter geometry.

Continuous Representation for Discrete Calorimeter Cells While PCs are continuous and can take any position in space, calorimeter cells are discrete.

- ▼ NNs typically map a continuous input to a continuous output[1]. To evaluate the model, the points generated in continuous space need to be mapped to discrete calorimeter cells. Each cell may only be assigned one energy, so the mapping must be injective. Thus, either the mapping needs to be injective by itself, or multiple energy depositions assigned to the same cell have to be combined (Section 10.3.4).
- ▼ The dominant method to train NNs is via gradient descent, which requires the NNs to be differentiable. However, calorimeter cells are discrete in nature. This introduces multiple issues for both training and evaluation, especially for GANs. The critic could quickly learn the discrete distances between neighboring calorimeter cells and greatly penalize the generator for any single point not on the grid, effectively derailing the GAN training. A similar problem exists for likelihood-based models like Variational Autoencoders [8] and Normalizing Flows [57]. A PC with a single point that does not align with a cell would need to be assigned a likelihood of 0, which might destabilize the training. In this thesis, this issue is approached by mapping the discrete cells to a continuous space for training (Section 10.3).

[1]There are exceptions, such as transformers (Section 3.5) producing tokens or the VQVAE producing codebook vectors. However, the quantization in these cases is not intrinsic to the model but is obtained by taking the argmax of the transformer's output or matching the NN output to the closest codebook vector.

Varying Size The number of energy depositions varies from shower to shower, and consequently, the size of the PC varies as well.

- ▼ Unlike grid-based data, PCs cannot be represented as matrices with constant dimensions. The handling of PCs is more challenging due to the varying cardinality.

- ▼ While modeling the cardinality is possible, PC-based models frequently require the cardinality of the PC they should generate (Section 4.6).

Use Case These advantages and disadvantages make PCs a *better* representation for *more granular* calorimeters, where sparsity is higher and denser cells form a 'more continuous' space.

Existing PC-based Models

While there are existing PC models that have been developed on PC datasets like, e.g., ShapeNet [98], the main focus of these models is to produce PCs that reproduce the shapes or surfaces of 3D objects. These points are largely independent of each other, or the dependency between the points plays a minor role, which is why many of these models produce independently distributed points [99, Sub. C]. For calorimeter simulation, the dependencies between points must be correlated, making ShapeNet an unsuitable dataset for this purpose. The PC-based GANs designed for jets and particle showers are presented in Section 4.6.

4.5. Permutation Equivariance and Invariance

PCs and the energy depositions they represent are inherently unordered. The order of the points in memory is therefore arbitrary. Designing a network that does not depend on this order avoids the need to learn this order independence.

Two types of order independence are distinguished based on the network's output space. A network that transforms a PC by mapping each point in the input PC to a corresponding point in the output PC should preserve the order (equivariance). If the network maps a PC X to another space, the output should be independent of the order (invariance).

A function f is permutation equivariant if, and only if,

$$f(PX) = Pf(X), \tag{4.1}$$

for any permutation P. Only functions that do not change the cardinality of the PC can be permutation equivariant.

Similarly, a function g is permutation invariant if, and only if,

$$g(PX) = g(X), \tag{4.2}$$

for any permutation P.

Concatenation of Permutation Invariant and Equivariant Function It follows that a function composed of a permutation invariant function g concatenated with a permutation equivariant function f is permutation invariant:

$$g \circ f(PX) = g(f(PX)) = g(Pf(X)) = g(f(X)) = g \circ f(X). \tag{4.3}$$

Let f take the PC X and a vector as input, and be equivariant under permutations of X. The equivariance is preserved if the vector is produced by a permutation invariant function g:

$$f(PX, g(PX)) = f(PX, g(X)) = Pf(X, g(X)).$$

Thus, permutation invariant or equivariant networks can be designed by chaining permutation invariant or equivariant layers. In this thesis, invariance and equivariance always refer to behavior under permutation.

4.6. Related Point Cloud-based GANs for Jets and Calorimeter Showers

As described in Section 4.4, representing particle showers as PCs instead of voxels has multiple advantages. In a GAN approach, the generator would need to map from a random vector to a PC of varying cardinality. Such a mapping is difficult to construct.

Refinement Approach

PC-based GANs [99–103] frequently choose a *refinement* approach shown in Fig. 4.1: The generator starts out with a PC of the desired size with points sampled from noise. A series of update blocks is applied to the PC, first in the generator and then in the critic. In the final step of the critic, the points are aggregated and mapped to a single value, usually with an FFN (Section 3.1).

Frequently, the update blocks match one of the following three categories, described in Message Passing (Section 3.6) terms:

Update via Dense Connections Each node is updated with the full information from all other nodes. While this approach maximizes the information accessible to each node, it may also make it challenging to filter out the information relevant to a specific node. Additionally, a significant drawback of this method is its quadratic time complexity for the cardinality. An example of this approach is Ref. [99].

Local Update Edges are constructed between the nodes, commonly by k-NN or a distance criterion. The resulting graph is then updated with an MPL. This method focuses on local relations between the nodes, such as the distance, and might make it difficult to model global distributions like mass or hit energy sum. The run time of the MPL is proportional to the number of constructed edges. Ref. [104] is an example of this method.

Update via Central Node A first MPL collects messages from each node to a single, separate node, the *central node*. In a second MPL, each of the nodes in the PC receives a message from this central node. This approach is more sensitive to global properties of the PCs and less to local ones. Because the number of messages is the cardinality of the PC for both MPLs, the time complexity scales linearly with the cardinality. Examples of this method are provided in Refs. [100–103].

Frequently, the update block includes an FFN, that updates the points individually before or after the "Message Passing" step(s).
In an image-based GAN approach like DCGAN [63], convolutions are applied to iteratively up- or downscale the data: The generator would apply convolutions in sequence to iteratively upscale a random vector to an image and then the critic would apply convolutions to iteratively downscale the image to a scalar. Such operations cannot be directly translated to PCs, which is why this refinement approach is prevalent for PC-based GANs.

In the following parts of this section, the architectures of selected PC-based GANs for particle showers (Section 2.1) are presented. Particle showers are inherently tree-based processes, as each particle is produced by the decay or detector interaction of a particle

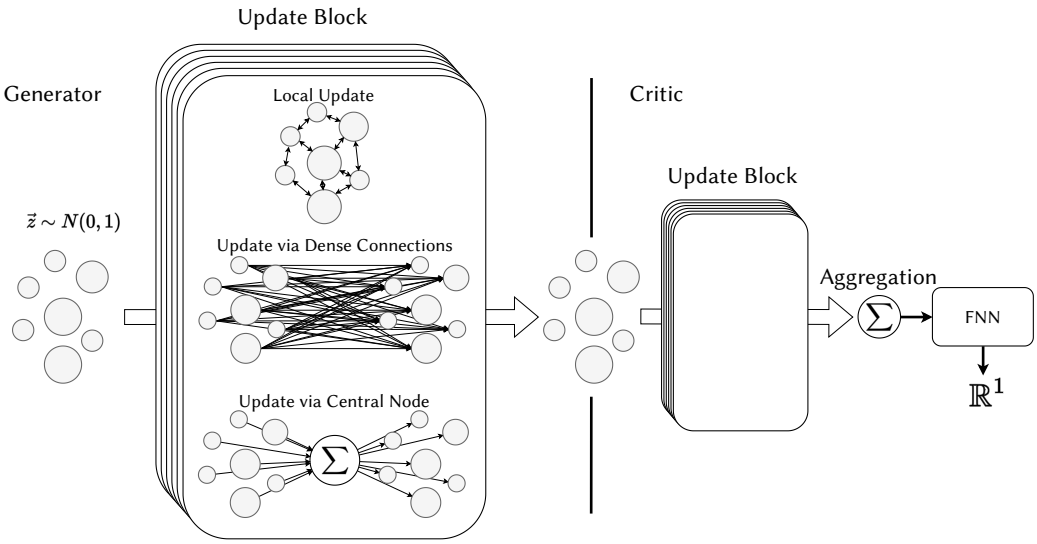


Figure 4.1. | Scheme of the Refinement Approach common in PC-based GANs| Described in Section 4.6

of the previous generation. An existing GAN that produces PCs in a tree-based manner is TREE-GAN [105]. While applying TREE-GAN (Section 4.6.4) to this task was not successful, it inspired the DEEPTREE model [106, 107], which has been developed in the course of this thesis and is also presented in this section.

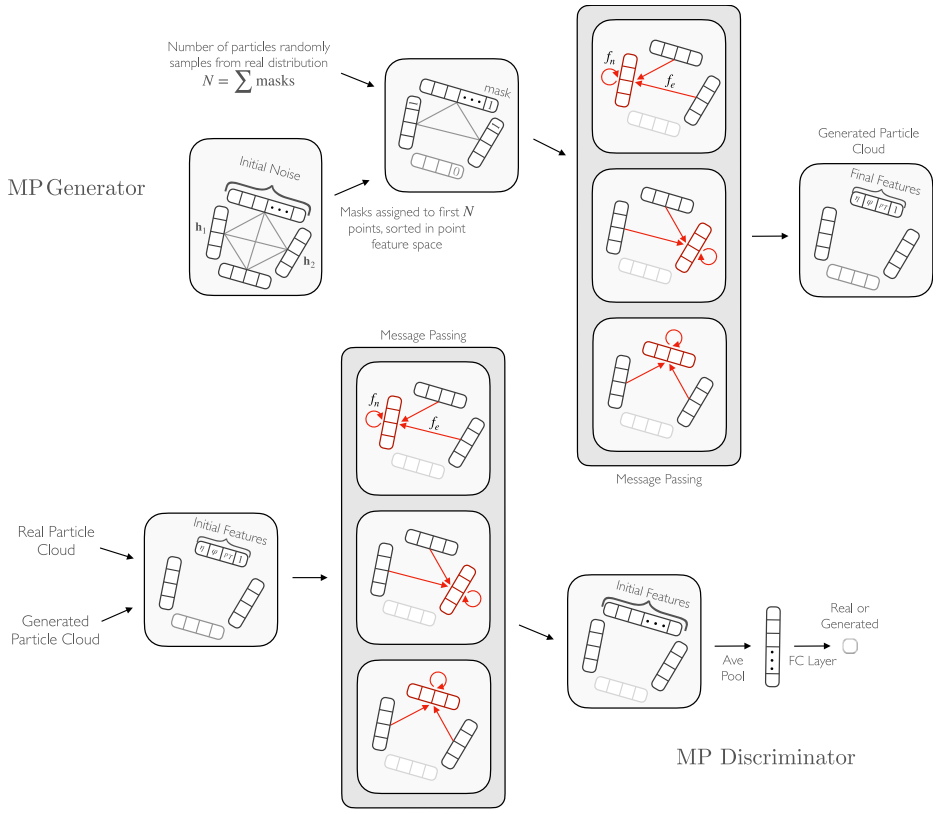


Figure 4.2. | The MP-GAN model.|
See Section 4.6.1 for the discussion. Taken from Ref. [99] and modified.

4.6.1. MP-GAN

The 'Message Passing GAN' (MP-GAN) [99] is the baseline model provided with the JetNet dataset. It outperforms numerous PC-based GANs like r-GAN [108], GraphCNN-GAN [109], and tree-GAN [105] on the JetNet dataset. MP-GAN follows the refinement approach described in Section 4.6 and Fig. 4.1, specifically the 'Update via Dense Connections' method.

Both the generator and the critic are composed of a sequence of two MPLs each. They follow the update rule:

$$\mathbf{x}_i' = \mathrm{FFN}_u\left(\mathbf{x}_i, \sum_{j\in\mathcal{N}(i)} \mathrm{FFN}_m(\mathbf{x}_i, \mathbf{x}_j)\right),$$

where the two FFNs consist of three fully connected linear layers each. For these MPLs, the points are densely connected, meaning the neighborhood of each point is the entire PC. The generator starts out with a PC of randomly initialized points that are updated by two MPLs. The discriminator processes an input PC through its two MPLs, averages the points, and then maps the average to a scalar using a single linear layer.

One downside of this implementation is that the dense connections in the MPLs connect each point with every other point, resulting in the aforementioned quadratic scaling of time complexity with the cardinality. While this is manageable with a cardinality of 30 for

JetNet-30, it becomes impractical for JetNet-150 and even more so for particle shower datasets with higher cardinalities.

The critic of this model is used for the proof-of-concept of the DeepTree generator in Section 7.3.

4.6.2. EPiC-GAN

The 'Equivariant Point Cloud GAN' (EPiC-GAN) [103] was the first model to achieve good results on the JetNet-150 dataset. This model adopts the refinement approach outlined in Section 4.6 and Fig. 4.1, specifically the 'Update via Central Node' method.

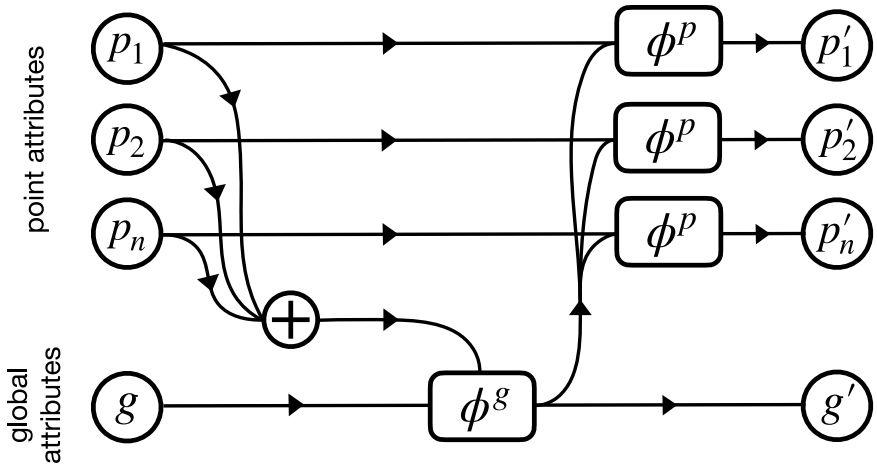


Figure 4.3. | The update block of EPiC-GAN.|
See Section 4.6.2 for the discussion. Taken from Ref. [103]. "The global function φ^g and point function φ^p are learned by NNs. The $\oplus$ symbol indicates the aggregation function $\rho^{p\to g}$, with both element-wise summation and average pooling."

EPiC-GAN starts with a randomly initialized PC. The cardinality n is sampled from the dataset. Figure 4.3 illustrates the update block used in both the generator and the critic. The FFN ϕ^g takes an aggregation ('$\oplus$') of the points (referred to as 'point attributes') p_i and the central node (referred to as 'global attributes') ϕ, to update the central node. This aggregation is the concatenation of the feature-wise sum and the feature-wise mean. A second FFN, ϕ^p, is then evaluated for each point in parallel to update the points with the central node. Through these two message passing steps to and from the central node, each EPiC layer computes at most n messages, thereby achieving a run time complexity proportional to the cardinality.

Both the generator and the critic consist of a sequence of the described update blocks. Before the first update block, both the generator and critic contain an FFN transforming the points separately. The generator contains an additional FFN that transforms the points after the last update block. In the critic, the central node is initialized by applying the sum-and-mean aggregation to the input PC, followed by an FFN. The same structure is used to map the PC to the scalar after the last update block.

The implementation of EPiC-GAN uses a regular tensor structure, i.e., matrices with fixed dimensions. During the training, PCs are grouped into batches with the same cardinality.

The authors compute the metrics after recentering the jet, i.e., recomputing the relative variables. This changes the scale of the computed metrics. Ref. [102] contains the recomputed values without this recentering. These values are used to compare the performance of this model.

4.6.3. MDMA

The 'Mean-fielD Matching Attentive GAN' (MDMA) [102] is a PC-based GAN that uses the attention mechanism (Section 3.5) to implement the refinement approach, specifically the 'Update via Central Node' (Section 4.6 and Fig. 4.1).

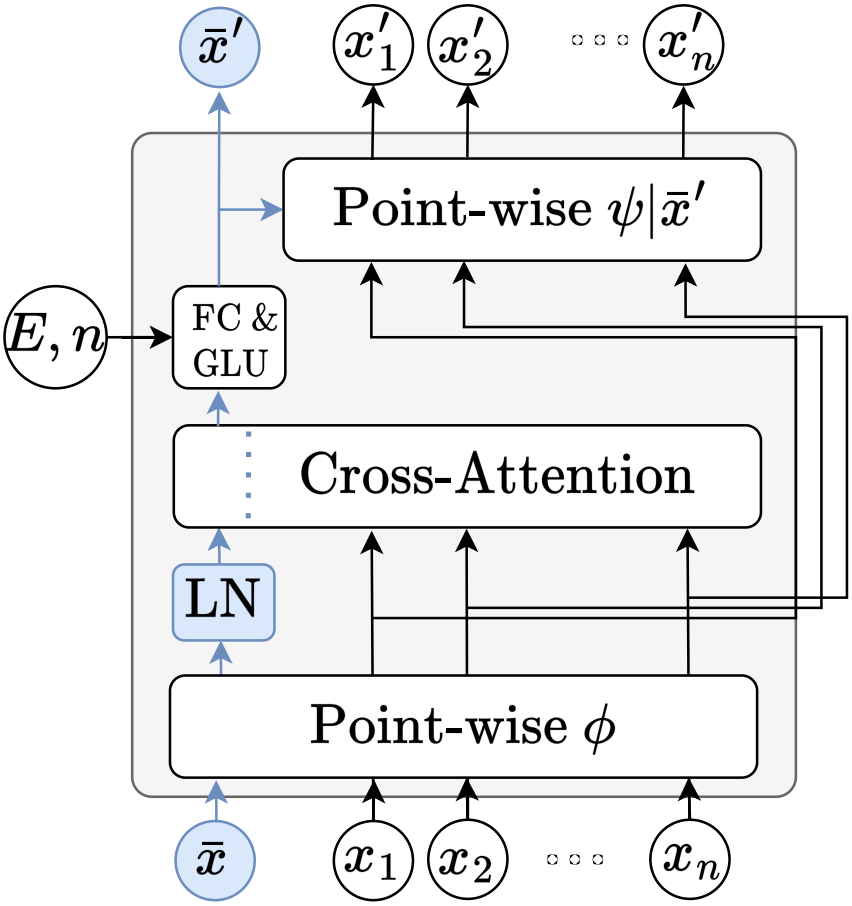


Figure 4.4. | The update block of MDMA.|
See Section 4.6.3 for the discussion. Taken from Ref. [102].

Like MP-GAN and EPiC-GAN, MDMA also starts out with a randomly initialized PC x. The cardinality n is sampled from the dataset. A *mean field* point, $\bar{x}$, is initialized with the feature-wise mean of the points in the PC. It will serve as the central node. Figure 4.4 shows the update block used in both the generator and the critic. First, all points are passed through a linear layer ('Particle-wise ϕ') that maps each point individually to f features. The mean field is then normalized with Layer Normalization (Section 3.3). Now, CrossAttention (Section 3.5) is used to update the mean field with all points from the PC. For this, the mean field is passed to the Multi-HeadAttention (Eq. 3.32) as the query $Q \in \mathbb{R}^{1\times f}$, the other points are passed as both key $K \in \mathbb{R}^{n\times f}$ and value $V \in \mathbb{R}^{n\times f}$. In the CrossAttention, Q is first multiplied with K^{T}, yielding a dimension of $1 \times n$. After `softmax`, $\frac{QK^{\mathrm{T}}}{\sqrt{f}} \in \mathbb{R}^{1\times n}$ is multiplied with V, yielding a dimension of $1 \times f$. Because Q contains a single element, the mean field $\bar{x}$, MDMA achieves a scaling of the time complexity proportional to the cardinality. In the GNN terms, this corresponds to a message passing to the central node. After

concatenating the new mean field with the cardinality, they are passed through a linear layer. This vector is appended to each point and passed on as the new mean field to a further update block. Lastly, the points are passed individually through a final linear layer ('Particle-wise $\psi|\bar{x}'$'). In the GNN terms, this would correspond to the message passing from the central node to the other points.

The update blocks for generator and critic differ in activation, normalization, residual connections and other details. In the critic, the mean field of the last update block is mapped with a linear layer to a scalar.

MDMA yields competitive results for the JetNet (Chapter 6) and CaloChallenge (Chapter 10) but has recently been surpassed by diffusion-based models [110, 111], though at the cost of a slower generation time. Based on MDMA, a successor model with a higher fidelity was proposed [112]. It uses the same update blocks, but is a "conditional flow matching" [113] model instead of a GAN.

4.6.4. TREE-GAN

The idea to generate a PC using a tree-based upscaling in a GAN was first introduced in TREE-GAN [105]. The authors target the generation of PCs representing the surfaces of 3D objects like planes, tables, or cars. As a critic, the r-GAN [108] critic is used. The following description uses the tree terminology introduced in Section 3.6.1.

Model Description

The generator is constructed from a series of alternating 'branching' and 'graph convolution' layers that are applied to a single root node $p_1^{l=0} \in \mathbb{R}^{96}$ sampled from a standard normal distribution. In the branching layer l, each of the input points p_i^l is mapped with a matrix V_i^{l+1} point to d_l new points:

$$p_j^{l+1} = [V_i^{l+1} p_i^l]_j \quad \text{for} \quad j \in \{1, \dots, d_l\},$$

where $[A]_j$ denotes the j-th column of a matrix A. After applying k branching layers, the cardinality has been scaled up to $\prod_{l=1}^{k} d_l$ points. The authors use a different branching factor of 2 for all but the last branching layer: $\{d_l\}_{l=1}^{7} = \{1, 2, 2, 2, 2, 2, 64\}$. After each branching layer, the points are updated with a graph convolution layer:

$$p_i^{l+1} = \sigma\left(\overbrace{\mathbf{F}_K^l(p_i^l)}^{\text{Loop}} + \overbrace{\sum_{q_j \in \mathscr{A}(p_i^l)} U_j^l q_j}^{\text{Ancestor}} + \overbrace{\widetilde{b^l}}^{\text{Bias}} \right),$$

where $\mathscr{A}(p)$ are the ancestors of node p.

The *loop* term transforms the newly constructed leaves. First, LeakyReLU activation is applied to them, followed by a linear layer without bias W that maps the number of features to f_{l+1}.

$$\mathbf{F}_K^l(p_i^l) = \underbrace{W}_{f_{l+1} \times f_l} \text{LeakyReLU}(\underbrace{p_j^l}_{\widetilde{f_l}}). \tag{4.4}$$

In the *ancestor* term, the U_j^l matrices project the ancestors of each leaf to n_f features. As the number of features differs for each level of the tree, a separate matrix is needed for each preceding level of the tree. All the terms are summed up and assigned as the new features of the leaves. For all but the final layer, LeakyReLU is applied as activation.

The *bias* term is a trainable vector.

Clustering Issues

By omitting the *ancestor* term, the authors demonstrate that the distance between points in the final layer is determined by the number of common ancestors [105, Eq. 9]. This causes TREE-GAN to frequently position nodes originating from the same parent close to each other. The resulting clusters are shown on the right in Figure 4.5, where points are colored according to their parent. While this clustering may be advantageous when modeling the surface of objects, it becomes a disadvantage when point density is crucial, as is the case with the particle showers targeted in this thesis.

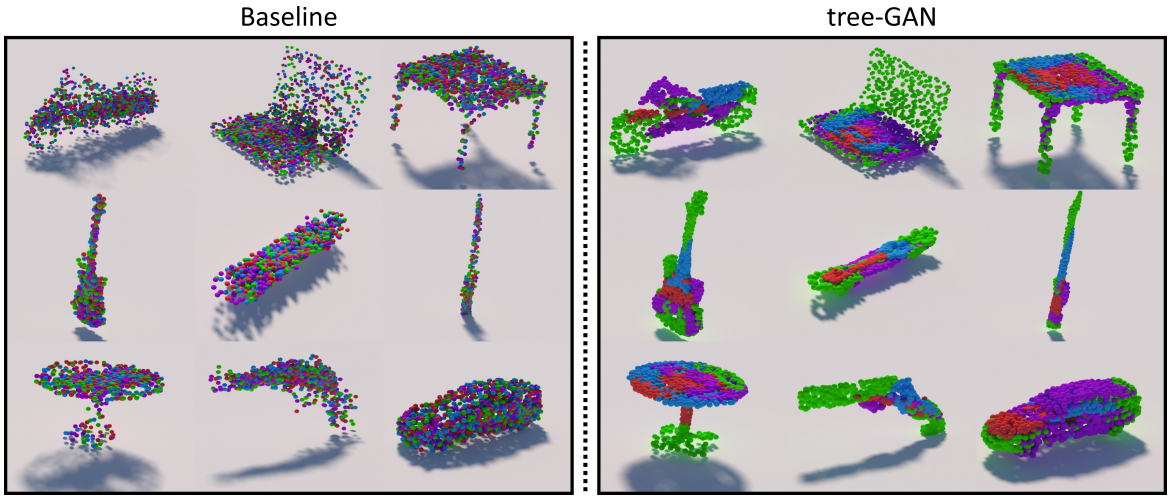


Figure 4.5. | 3D PCs Generated by TREE-GAN.|
"Unsupervised 3D [PCs] generated by the TREE-GAN for multiple classes (e.g., Motorbike, Laptop, Table, Guitar, [...]). Our TREE-GAN [...] can also produce [PCs] for semantic parts of objects, which are denoted by different colors." [105].

Modeling a 2D Gaussian

The effects of this can be seen in Fig. 4.6, where TREE-GAN has been trained to produce a 2D Gaussian distribution ($\boldsymbol{\mu} = [1, 1]$, $\boldsymbol{\sigma} = [[1, 0.5], [0.5, 1]]$). The overall shape of the Gaussian distribution is matched by TREE-GAN, but there are some gaps in the tails of the distribution. The density, however, is grossly misrepresented, as a large fraction of the points are placed close to (0,0). As TREE-GAN targets the generation of surfaces, modeling the density is secondary to modeling the shape correctly. This is the opposite of the particle shower generation aimed at in this thesis.

By replacing the matrix operations of TREE-GAN, the performance of the model can be improved. Figure 4.7 (Fig. 4.8) shows the 2D histogram for a model where the branching matrices V^l (ancestor matrices U^l) have been replaced with FFNs. These FFNs consist of 8 linear layers with LeakyReLU activation, and the number of hidden nodes matches the output dimension. Further hyperparameters are shown in Table 4.1. In both cases, the 2D histograms show 8 distinct peaks instead of a smooth Gaussian distribution, effectively splitting the singular peak into 8 compared to the unchanged model.

In addition to its insufficient performance, TREE-GAN faces a significant challenge when generating large PCs: The number of matrices, and consequently the number of parameters required for the branching and ancestor terms in the graph convolution, increases rapidly with the number of levels. This issue becomes particularly problematic for PCs with high cardinalities, where a tree with many levels is needed to produce such PCs.

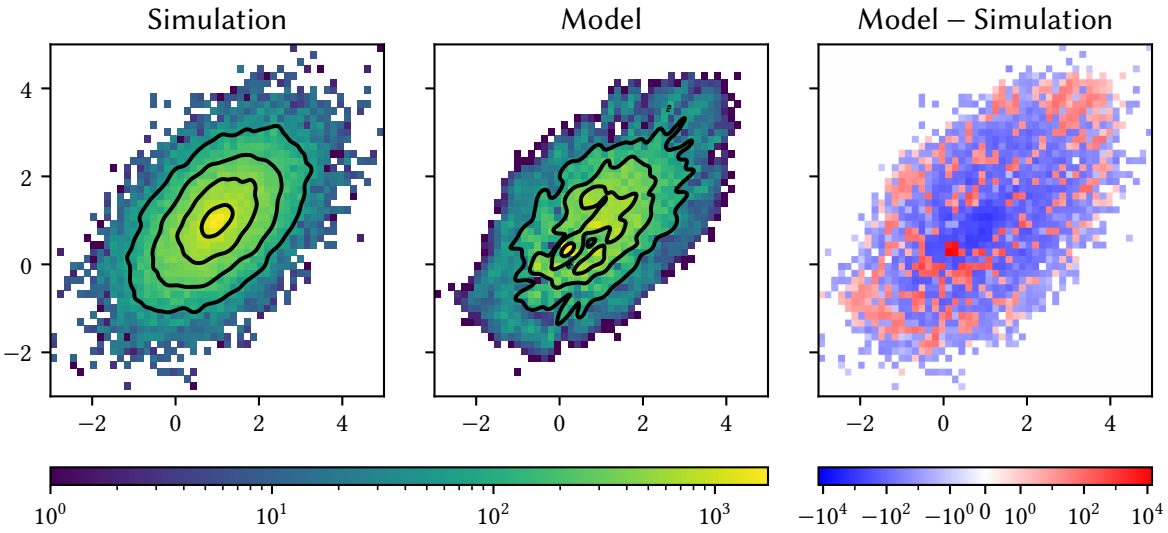


Figure 4.6. | 2D Histograms Comparing the output of TREE-GAN to the Target Gaussian Distribution.|
See Section 4.6.4 for the discussion. The plot design is explained in A.2. The model was trained for 36k steps. Approximately 165k points are shown.

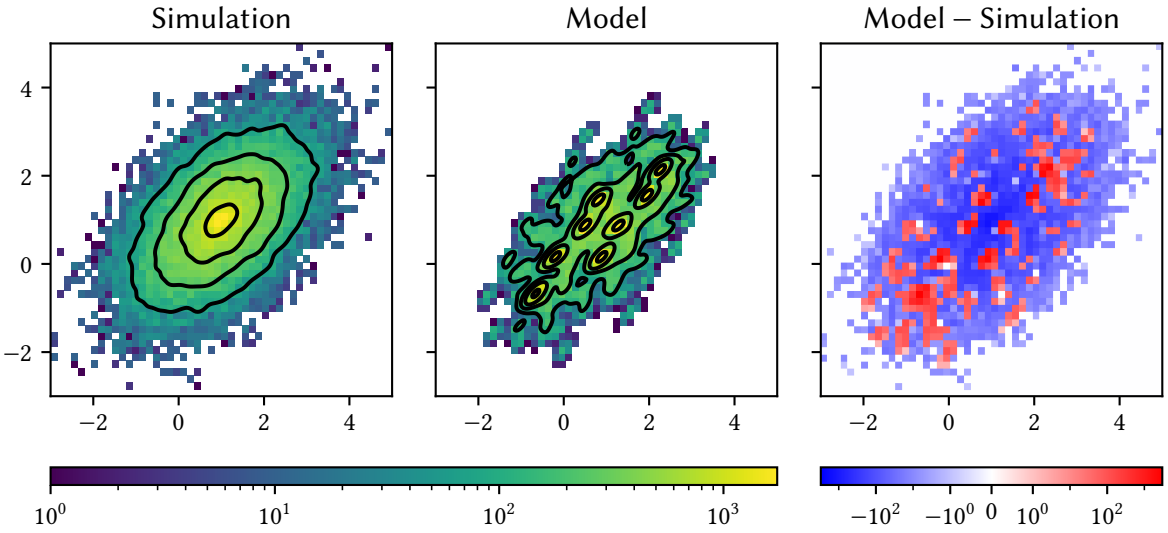


Figure 4.7. | 2D Histograms Comparing the output of TREE-GAN with the Modified Branching to the Target Gaussian Distribution.|
See Section 4.6.4 for the discussion. The branching matrices in TREE-GAN have been replaced with FFN. The plot design is explained in A.2. The model was trained for 44k steps. Approximately 165k points are shown.

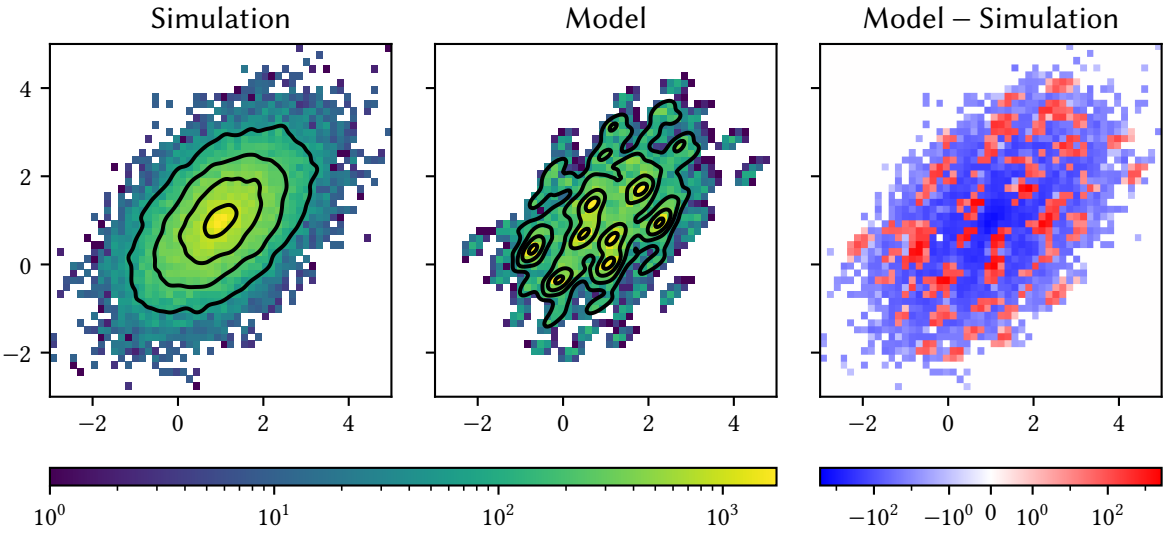


Figure 4.8. | 2D Histograms Comparing the output of TREE-GAN with the Modified Ancestor Convolution to the Target Gaussian Distribution.|
See Section 4.6.4 for the discussion. The ancestor convolution matrices in TREE-GAN have been replaced with FFN. The plot design is explained in A.2. The model was trained for 60k steps. Approximately 165k points are shown.

Table 4.1. | Hyperparameters for Training TREE-GAN on a Gaussian Distribution|
See Section 4.6.4 for the discussion.

Parameter	Generator	Discriminator
Loss	Cross-entropy	
Batch size	50	
Points per sample	512	
Optimizer	Adam	
Learning rate	10^{-4}	
Weight decay	$2 \cdot 10^{-4}$	
Gradient Steps per generator step	1	2
Features (n_f)	[96, 64, 64, 64, 2]	[2, 64, 128, 256]
Branching factor (d_l)	[2, 2, 2, 64]	–
Support (K)	10	–

Chapter Review Particle showers can vary significantly between calorimeters, requiring different approaches to handle them in a generative model. In this chapter, these approaches and their advantages are discussed (Sections 4.1 to 4.4). The assumptions of an irregular, highly granular calorimeter lead to the use of PCs as the data structure for the DeepTree model. Since PCs have no intrinsic order, PC-based models should be independent of the order, as discussed in Section 4.5. Lastly, related PC-based GANs, which will later serve as benchmarks for the DeepTree model, are detailed in Section 4.6.

CHAPTER 5

The DeepTree Model

Chapter Abstract In this chapter, the design of the DeepTree GAN, developed by me, is described briefly. It is motivated by the tree-like nature of particle showers. The generator uses a tree structure to map a single node to a PC, while the critic maps the input PC iteratively to smaller and smaller PCs. As such, both parts of the model change the size of the PC and thus do not follow the popular refinement approach detailed in Section 4.6. The performance of this model and the design choices are evaluated in the following chapters.

In a tree-based process, the properties of a node depend solely on its ancestors. Particle showers follow a tree-like structure, where the properties of particles determine the properties of their decay products and energy depositions. Modeling the architecture to mirror this tree structure may not only facilitate the PC upscaling, but also introduce a beneficial inductive bias.

The idea of DeepTree was to develop a tree-based generative model for PCs, that benefits from this inductive bias. Due to their flexibility in architecture, fast generation, and high fidelity, this model was developed as a GAN.

Generator Approach

The DeepTree generator starts out with a single point. At each step, a mapping generates a fixed number of new points from each existing point. This process continues until the desired number of points is reached. By starting from a single point and iteratively expanding it into a larger PC, the input and output dimensions of each mapping are constrained, thereby reducing the number of required parameters. Directly mapping a single point to a large PC would require an infeasible number of parameters.

tree-GAN

While tree-GAN [105] follows a tree-based approach as well, applying tree-GAN to jets or particle showers leads to subpar results, as shown in Ref. [99]. Modifications to tree-GAN could not sufficiently improve the model (Section 4.6.4). Although the DeepTree generator is based on similar ideas as tree-GAN, its model architecture differs significantly. A detailed comparison to tree-GAN is provided in Section 5.1.

Invariance and Equivariance in the Refinement Approach

The PC-based GANs for jets and particle showers typically employ a refinement approach (Section 4.6). In this approach, the generator starts out with a PC of the desired cardinality and subsequently transforms it. This design allows for a permutation invariant critic and a permutation equivariant generator (Section 4.5). However, the DeepTree generator changes the cardinality and thus cannot be equivariant[1]. The DeepTree critic, by contrast, is designed to be permutation invariant.

Critic Approach

In a critic, the input is mapped to a scalar value to separate the generated samples from real data. Typically, PC-based critics first transform the PC and then aggregate the points into a single vector, often using a sum. This vector is then mapped to a scalar, usually through an FFN. This aggregation causes a sudden reduction in dimensionality in a single step. Examples of this approach may be found in Refs. [99, 102, 103, 105]. During backpropagation, the gradients for all points must pass through this bottleneck. Although this does not theoretically limit the critic [114], it may pose practical challenges.

The goal of this critic is to avoid this bottleneck by using an iterative aggregation method. Furthermore, the critic should be permutation invariant and avoid quadratic time complexity with the cardinality C. This iterative aggregation is achieved with a pooling operation named *bipartite pool*. Each bipartite pool contains a PC with k trainable points. Edges are constructed from each input point to each point of the pool, forming a bipartite graph. A message passing layer (Section 3.6) applied to this graph creates a new PC combining information from both PCs, but with the cardinality k. The bipartite pool's time complexity scales with $c \cdot k$, thus meeting the scaling requirement. By using a sequence of bipartite pools with decreasing cardinality, the input PC is aggregated iteratively.

Development Strategy

Starting the development of a PC-based model directly on a particle shower dataset with 10^4 to 10^5 hits would present numerous challenges:

- Generating large PCs will inherently result in slow training and evaluation, leading to long turnaround times.
- In model development, the architecture is iteratively improved. If training fails completely, performance metrics cannot be evaluated, making it impossible to determine whether a change constitutes an improvement. This failure is more likely with models producing larger outputs. Preliminary experiments on smaller-scale PCs can demonstrate the model's fundamental viability, providing a basis for iterative refinements to tackle larger datasets.
- In a GAN, feedback from a critic is needed to train the generator. In the development of the generator, one may choose to use an 'established' critic, like the MP-GAN critic (Section 4.6.1), that has demonstrated competitive performance. This critic may not have the required memory or run time scaling for large PCs.

[1] An alternative design for a tree-based generator, invariant under certain permutations, is detailed in Appendix C.1. While achieving competitive results generating 30 points for JetNet-30 (Chapter 7), it ultimately failed to generate up to 150 points for JetNet-150.

- Reliable design decisions require well-studied, sensitive performance metrics. These metrics may be computationally expensive or unavailable for large-scale particle showers.
- As described in Section 4.4, discrete calorimeter cells need to be mapped to a continuous space for training and vice versa for evaluation. Both directions of this mapping introduce potential errors or failures to a particle shower model.

Therefore, it is advantageous to first develop the model on a suitable proxy task and move to calorimeter simulation once the model's capabilities have been established.

Jet Generation as a Proxy Task

The generation of jet constituents (see Section 2.4) serves as a natural proxy for calorimeter simulation. Both processes involve tree-like physics processes that generate large PCs under constraints, such as total energy or invariant mass. These constraints result in distinct global distributions, like the shower energy distribution or the double peak in the mass spectrum of top-quark initiated jets. This complexity makes jets challenging to model and provides sensitive, physics-based metrics for evaluation. Consequently, if a scalable model performs well on jets, it is likely to perform well on particle showers.

While the cardinality of a PC representing a particle shower can reach 10^5, the used jet datasets typically have cardinalities ranging from 10^2 to 10^3, making jet generation significantly faster than shower generation. Additionally, the hits of particle showers are positioned discretely in space (at the cells), whereas jet constituents are distributed in a continuous space. Thus, the translation between the model's continuous space and the data's discrete space, required for calorimeter cells, can be omitted.

In summary, developing a PC-based generative model using jets allows for fast turnaround times, the application of sensitive metrics, and circumvents issues related to the discrete nature of calorimeter cells. This makes jets an ideal test bed for developing models for particle showers.

Chapter Outline

In the following sections, the generator and critic architectures are briefly presented. This is followed by a description of the training process, and the various FFNs used across the model. The names of the hyperparameters are introduced in `Monospaced Font` in the following sections. The default hyperparameters are listed in Table 5.1. Chapter 9 contains a systematic review of the design choices. The model's capabilities are showcased in the subsequent chapters.

Table 5.1. | Default Hyperparameters for the DeepTree model|
See Chapter 5 for the discussion. This set of hyperparameters used is in Chapter 8 and serves as the reference for Chapters 7, 9 and 10.

Common		Ratio Gradient Steps	1:1
		Optimizer	Adam ($\beta_1 = 0.9, \beta_2 = 0.999$)
		Weight Decay	10^{-4}
		Batch Size	200
		Validation Interval	2000 Critic Steps
	Early Stopping (Section 5.3.3)	Window	1000
		Minimum Improvement	0.95
	FFNs (Section 5.3.4)	Order	[Linear, Dropout, Normalization, Activation]
		Hidden Nodes	100
		Linear Layers	3
		Bias	None
		Activation	LeakyReLU($s = 0.1$)
		Weight Initialization	$U(-\sqrt{k}, \sqrt{k})$ with $k = (\text{input size})^{-1}$
Generator (Section 5.1)		Loss	Hinge + 0.1 Feature Matching (Eq. 5.1)
		Learning Rate	10^{-5}
		Node Features by level	[64,33,20,10,3]
		Branching Factor by level	[1,2,3,5,5] ($\prod b_i = 150$)
		Global Feature Size	10
		Condition	[Cardinality]
	FFNs	Dropout	0
		Normalization	Batch Normalization
	Ancestor MPL (Section 5.1.2)	MPL	GINConv (Section 3.6.4)
		Number of MPLs	1
		Hidden Nodes	100
Critic (Section 5.2)		Loss	Hinge
		Learning Rate	$3 \cdot 10^{-5}$
		Point Dimensions by level	[3,10,10,10]
		Cardinality by level	[Cardinality,30,6,1]
		Condition	[Cardinality, p_T^{jet}, η^{jet}, m^{jet}]
	FFNs	Dropout	0.5
		Normalization	Spectral Normalization
	Bipartite Pool (Section 5.2.1)	MPL	GATmConv (Section 3.6.5)
		Attention Heads	16
		Normalization	None
		Dropout	0.0
	Embedding Layer (Section 5.2.2)	Latent Features	40
		Normalization	Batch Normalization
	Subcritics (Section 5.2.2)	Latent Features	40
		Central Node Dimension	40
		Number of CNUs	2

5.1. Generator

The DeepTree generator maps a noise vector to a PC using a tree-based structure. It was published in Ref. [106]. The following description uses the tree terminology introduced in Section 3.6.1.

Architecture

The generator is composed of multiple *generator levels*, each containing a set of components with independent parameters, as shown in Fig. 5.1. The tree begins as a single root node, sampled from Gaussian noise. As the tree progresses through the sequence of generator levels, the nodes are updated, and new levels are added to the tree. In each generator level, the following sequence of layers is executed:

1. The *global feature* layer aggregates the highest level of the tree, i.e., the leaves, into a *global feature* vector. This global feature is then passed to the branching layer and the ancestor MPL. The nodes themselves remain unchanged by this layer. A detailed description of this layer is provided in Section 5.1.3.

2. The *branching* layer takes the current leaves and maps each to a fixed number of new nodes. These new nodes are then attached as children to the node from which they were created, thereby adding a new level to the tree. This layer is described in detail in Section 5.1.1.

3. The *ancestor* Message Passing Layer (ancestor MPL) updates all nodes depending on their ancestors. This step does not alter the number of nodes or the number of features. A detailed description of this layer is provided in Section 5.1.2.

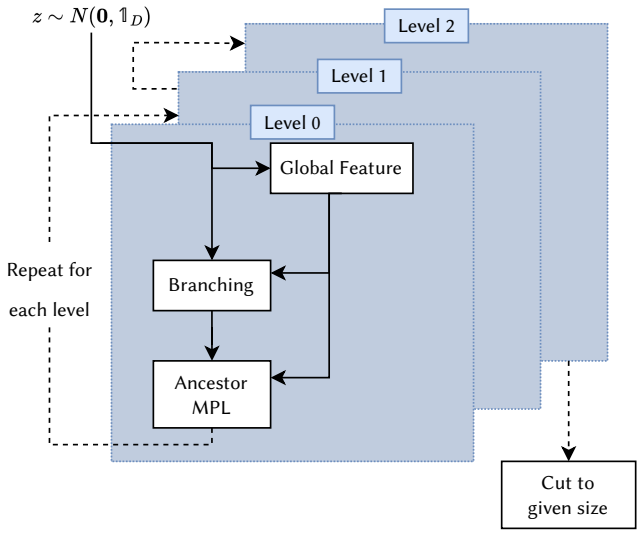


Figure 5.1. | The DeepTree Generator.|
See Section 5.1 for the discussion.

Level Counting The levels of the tree and the generator are indexed beginning from 0. The `Node Features` at each tree level are denoted by f_l, and the `Branching Factors` by b_l. It should be noted that the branching factor at level 0 is set to 1, corresponding to the root node ($b_0 = 1$). The node features decrease with each subsequent level ($f_l > f_{l+1}$ for all l). At generator level l, the branching layer maps a tree with l levels to a tree with $l + 1$ levels. This mapping increases the number of leaves by a factor of b_{l+1}. Simultaneously, the number of features per node decreases from f_l to f_{l+1}.

Output Cardinality The number of leaves $\prod_l b_l$, increases with each level until the desired cardinality is reached. The branching factors b_l are selected so that $\prod_l b_l$ meets or exceeds the maximum cardinality in the dataset. For each PC, the cardinality C is sampled from the dataset. The first C leaves from the highest level of the generator are then returned as the output PC.

5.1.1. Branching Layer

Figure 5.2 illustrates the branching layer of generator level 1. It takes a tree with two levels (root node + 1st level) as input and returns a tree with three levels as output. With the nodes from tree level 1 (dark green / orange) as the parents, their children (blue+green / red+yellow in tree level 2) are generated independently for each parent: First, the global feature is appended to the parent ('Concat'). Then, each parent (in $\mathbb{R}^{1\times f_l}$) is mapped by the *branching FFN* to its size times the number of branches ($\mathbb{R}^{b_{l+1}\times f_l}$). After splitting the output into b_{l+1} new children for each parent ('Reorder'), the parent is added to each of them as a residual connection ('+'). With the new children added as leaves to the tree, all the levels of the tree are stacked up and passed through a *dimensionality reduction FFN* ('Dim. Red. FFN'). This FFN reduces the number of features of all nodes from f_l to f_{l+1}. Thus, with each branching layer, the number of leaves increases while the number of features decreases.

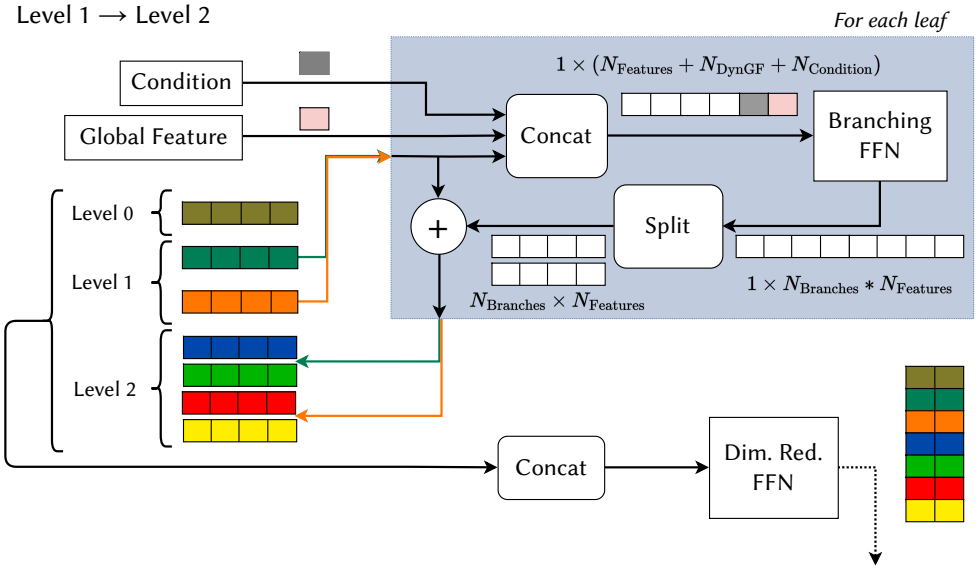


Figure 5.2. | The Branching Layer for Generator Level 1.|
See Section 5.1.1 for the discussion.

5.1.2. Ancestor Message Passing Layer

The ancestor MPL propagates information down the tree from each ancestor to its descendants. For this purpose, a graph is constructed, where each node is directly connected to its descendants. Additionally, self-loops, connecting each node to itself, are included in the graph. Figure 5.3 shows an example of this graph structure from the perspective of **node 4**, which is updated with the aggregated **messages** from nodes **1**, **2**, and **4**. The MPL used is GINConv Section 3.6.4, as implemented in PyG [77]. In GINConv, the **messages** are the **source nodes**, which, in this case, are the ancestors of the target node. These **messages** are aggregated by summing over all **messages** addressed to each **target node**. The aggregates are then added to the **target nodes** and passed through an FFN. Finally, the **target nodes** are updated with the output of the FFNs plus the original values of the **target nodes** as a residual connection. For the last linear layer of the FFN, normalization

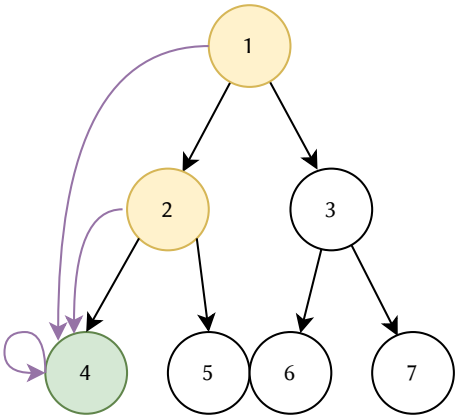


Figure 5.3. | Sketch of an example graph of the Ancestor MPL for Generator Level 2.| See Section 5.1.2 for the discussion.

and activation are not applied, so the network's output remains unrestricted. The FFN inside GINConv is constructed like the other FFNs in the generator (Section 5.3.4).

Run Time and Performance Impact

The ancestor MPLs compute a message for each ancestor of every node at each generator level. For a tree with k layers, $\sum_{k=0}^{k} \prod_{j=0}^{l} b_j$ messages need to be computed in the branching layer. The number of parameters increases linearly with the number of levels, since a single FFN is used for each ancestor MPL and level. However, the run time of the MPL is proportional to the number of messages, making the ancestor MPL the component with the worst timing complexity in the model. Removing the ancestor MPLs showed no significant impact on the run time for a cardinality of 150 (Section 9.4.3). However, this impact is expected to become significant when generating large PCs. The ancestor MPL is not strictly necessary, unlike the branching layer. Previous studies on JetNet-30 using a different branching layer (Appendix C.1) suggested a performance boost with its inclusion. However, this finding was not reproduced in the ablation study on JetNet-150 (Section 9.4). Therefore, whether this component should be included in the model remains inconclusive, especially for large PCs.

5.1.3. Global Feature Layer

Figure 5.4 shows the Global Feature Layer for Generator Level 2. The leaves (tree level 2) are first passed through an FFN ϕ_1 that maps them to n_{hm} features. The transformed leaves are summed up, and the resulting vector is passed through a second FFN ϕ_2 that maps it to the `Global Feature Size`. This vector is then concatenated with the generator's `conditioning variables`. n_{hm} is the harmonic mean of the `Node Features` and the `Global Feature Size`, rounded down.

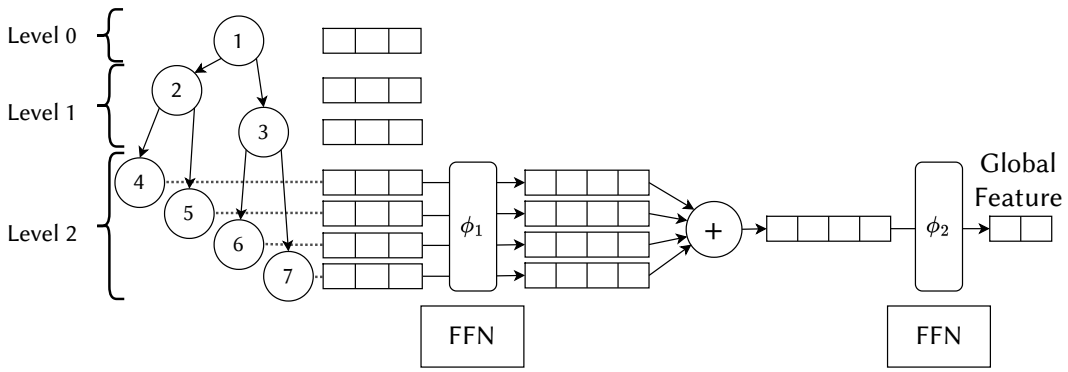


Figure 5.4. | The Global Feature Layer for Generator Level 2.|
See Section 5.1.3 for the discussion.

5.1.4. Comparison to TREE-GAN

Compared to TREE-GAN (Section 4.6.4), this generator introduces major changes:

Branching The branching mechanism (Section 5.1.1) in DEEPTREE uses two small FFNs per level: One for increasing the number of nodes and another for reducing the number of features. The first FFN maps n_i to $b \cdot n_i$, and the second FFN reduces the dimension of each node from n_i to n_o. In TREE-GAN, a separate matrix for each parent maps from the input dimension n_i to b times the output dimension n_o. The dimension is then reduced in the subsequent ancestor convolution. Because a separate matrix is used for each of the parents in each level, this uses $\sum_{l=1}^{k} \prod_{r=1}^{l-1} d_r$ matrices for a tree with k levels.

Ancestor Convolution After the branching, both models implement a graph convolution step to pass information from the ancestors to their descendants. In TREE-GAN, only the leaves are updated using the leaf itself and its ancestors. Since each ancestor has a different dimension, each of them is mapped to the dimension of the leaf using a separate trainable matrix. Moreover, a separate set of these matrices is required for each level of the tree. Therefore, the total number of matrices for this step is $\sum_{l=1}^{k} l = \frac{k(k+1)}{2}$, where k is the number of levels in the tree. In DEEPTREE, all nodes in the tree maintain the same dimension at all times. This allows all nodes to be updated simultaneously using a single FFN in the ancestor MPL for each layer.

The self-loop term in TREE-GAN is included in the message passing step, as all nodes receive a message from themselves.

Summary DEEPTREE employs more complex components, such as MPLs and FFNs, compared to the matrices used by TREE-GAN. However, its uniform treatment of all nodes significantly reduces the number of required mappings. Applying an MPL to a dynamic tree structure requires tracking the tree level and PC each node belongs to, which substantially increases the complexity for DEEPTREE.

5.2. Critic

In the most common design for PC-based critics, the critic first transforms the points, then aggregates them into a single vector, and finally maps that vector to a scalar output (Section 4.6). The DeepTree critic deviates from this approach in two major ways:

1. The cardinality of an input PC is reduced iteratively. This structure yields a model that resembles image-based GAN approaches like DCGAN [63], where the critic applies convolutions in sequence to iteratively downscale an image to a scalar.

2. The DeepTree critic contains multiple *subcritics* that are placed before and after the cardinality is reduced, i.e., before and after the pooling operations. The critic produces a separate scalar for each subcritic instead of a single scalar.

The DeepTree critic was published in Ref. [107].

5.2.1. Point Cloud Pooling with the Bipartite Pool

For the iterative reduction of the cardinality, a pooling operation must meet the following requirements:

- It must be differentiable to enable backpropagation.
- It must accept an arbitrary input cardinality, given the varying cardinality of the inputs.

Additionally, it should satisfy the following criteria:

- The pooling should be permutation invariant, as the input PCs are unordered. If not, the critic would need to learn this invariance.
- The run time should scale well with large PCs, ideally linearly with the input cardinality C.
- The pooling should map to a selectable but fixed cardinality k, allowing for a regularly shaped output that allows faster subsequent operations.
- For efficient implementation, the pooling must be applicable to batches containing PCs with different cardinalities.

Existing Point Cloud Pooling Operations

These requirements exclude many existing graph pooling layers from the PyG library[2]. For instance, the 'Dense Pooling Layers' require a fixed cardinality input and the k-NN-based or clustering-based pooling layers tend to have long run times. In TopK-based pooling, points that do not rank among the k largest values in a specified feature are removed from the PC. A series of experiments using two pooling layers based on this approach, TopKPooling [115] and SAGPooling [116], have not yielded successful results.

[2] See pytorch-geometric.readthedocs.io/en/2.5.3/modules/nn.html#pooling-layers.

Bipartite Pool

In the Bipartite Pool *proposed with this model, a bipartite graph is constructed, densely connecting the input PC to k trainable nodes ("seed nodes"). An MPL is then applied to this graph, returning k points.*
The dense connections result in an edge count of $k \cdot c$. Since each edge generates one message, the time complexity of the MPL is proportional to $k \cdot c$. GATmConv[3], with 16 `Attention Heads`, is used as the MPL (Section 3.6.5).

Comparison to "Pooling by Multi-HeadAttention"

With the employed graph-attention mechanism (Section 3.6.5), this pooling operation corresponds to a graph version of the "Pooling by [Multi-HeadAttention]" [117]. In this method, the attention mechanism (Section 3.5) is applied to the input tokens and k trainable *seed nodes*, creating a permutation-invariant pooling operation. The Multi-HeadAttention uses the input as the key and value, while the seed nodes serve as the query. However, the authors only apply this pooling with a single seed node ($k = 1$).

Processing Variable-Sized Point Clouds with Regular Shapes

The attention mechanism relies on matrix operations, which require matrices with regular shapes. This limitation makes it incompatible with the PyG batches containing PCs of varying cardinality, as used in the DeepTree model (Section 5.3.5). To process a batch containing PCs of varying cardinalities with matrices of fixed dimensions, a workaround is necessary.

In the models described in Section 4.6, this is achieved by padding the PCs with dummy points to match the cardinality of the largest PC in the batch. The attention mechanism is then applied with a mask to exclude the dummy points from the computation. This approach is used in MDMA (Section 4.6.3) during the cross-attention calculation in its update block. However, padding introduces computational overhead, which increases with the variance in the cardinality of PCs within a batch. In contrast to this approach, the bipartite MPL approach circumvents this issue and offers greater flexibility by allowing the application of any message passing layer.

5.2.2. Architecture

The critic architecture is shown in Fig. 5.5. It features three subcritics, that are trained simultaneously. Each subcritic is applied at a different level of the aggregation. The first subcritic is applied directly to the input PC. The subsequent subcritics are applied after the pooling operations. The expectation is that the first subcritic provides feedback on lower-level features, while the latter two operate on more aggregated data, providing feedback on higher-level features. Before pooling, an embedding layer is applied. This layer maps the points to the `Point Dimension` for the current level using an FFN. The following transformation of the points is carried out through a "Central Node Update" (CNU) block, which is also used in the subcritics. In this block, the PC is aggregated into a single vector

[3] GATmConv is a variant of GATv2Conv with modifications to the calculation of the attention weights (see Section 3.6.5). The impact of reverting this modification is explored in Section 9.5 (configuration `62-crit.bbp.gatplus`).

that then transforms the points individually. This corresponds to two message-passing steps: one from the points to a central node, and another from the central node back to the points. The following sections will detail the components of the critic.

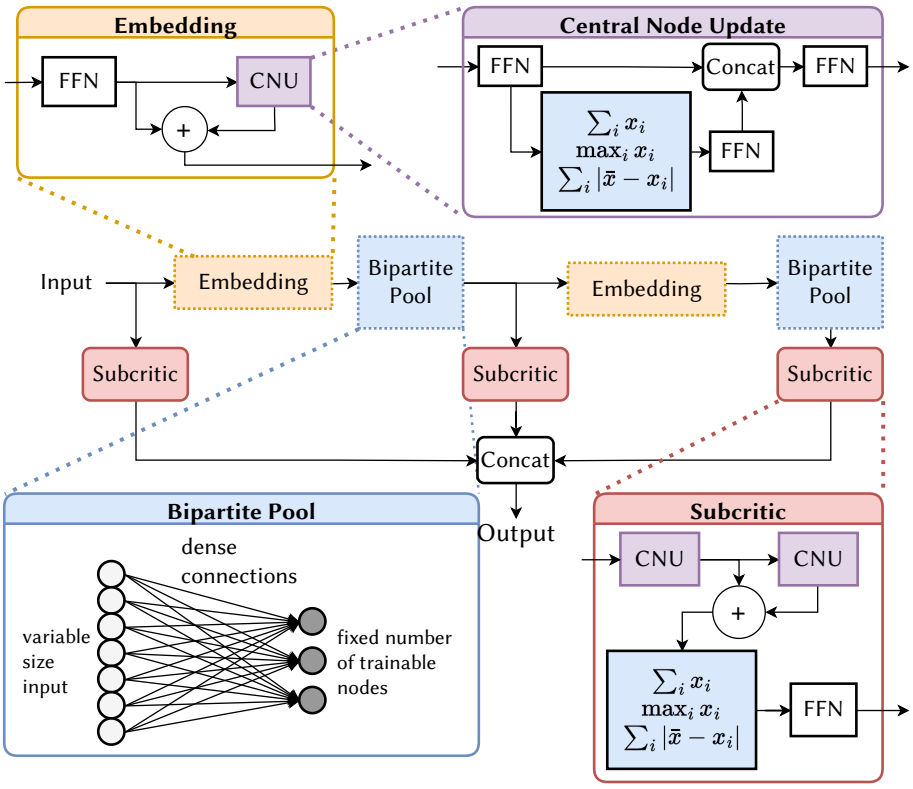


Figure 5.5. | The Critic and its Components.|
See Section 5.2 for the discussion.

Multi-Aggregation In the CNUs and subcritics, the points are aggregated into a single vector. This aggregation is typically achieved by summing over the points. To provide additional information about the distribution, the vector is computed by concatenating the sum, the maximum, the cardinality C, and the width. The width of the distribution is estimated by calculating the mean absolute deviation:

$$\frac{1}{c}\sum_i |\mathbf{x}_i - \frac{1}{c}\sum_j \mathbf{x}_j|.$$

Central Node Update For the following description of the embedding layers and the subcritics, a CNU block is defined as follows: First, the input PC is mapped to the latent dimension n_l using an FFN. Next, a central node is computed by "multi-aggregating" the transformed PC. The aggregated vector is then passed through a second FFN that maps it to n_g. This vector is appended to each of the transformed points. Finally, the points are mapped back to their input dimension using a third FFN. In the embedding layer, both n_g

and n_l of the CNU are set to the `Latent Features` (Table 5.1). In the subcritics, n_g is set to the `Central Node Dimension`, while n_l corresponds to the dimension of the incoming points.

Embedding Layers

An embedding layer is positioned before each bipartite pool. First, an FFN maps the points to a fixed dimension for each aggregation level (`Point Dimensions by level`). The transformed PC is then passed through a CNU block. A residual connection ('+') sums each point with and without the transformation by the CNU block. The resulting PC is returned as the output.

Subcritics

Each subcritic is constructed using two CNUs with a residual connection and an FFN. The input PC is first transformed through two subsequent CNUs. A residual connection ('+') sums the outputs of both CNUs. This PC is subsequently aggregated using the multi-aggregation scheme. The aggregated vector is concatenated with the `conditioning variables` and passed through an FFN, which maps the vector to a scalar. When three subcritics are employed, the critic produces three scalars for each input PC.

Let $L_C(C, G)$ and $L_G(C, G)$ be the critic and generator losses (Section 3.4), such as $-\mathbb{E}_{x\sim p_{\text{data}}}[C(x)] + \mathbb{E}_{z\sim p_z}[C(G(z))]$ and $-\mathbb{E}_{z\sim p_z}[C(G(z))]$ for the WGAN (Eq. 3.23). These loss terms are evaluated separately for each subcritic C_i and then summed, yielding:

$$\sum_{i\in\text{Subcritics}} L_C(C_i, G) \text{ as the loss for the critic, and}$$
$$\sum_{i\in\text{Subcritics}} L_G(C_i, G) \text{ as the loss for the generator.}$$

In this way, the batch is evaluated across three different critics simultaneously, with backpropagation taking place for all of them in parallel.

Permutation Invariance of the Critic

As argued in Section 4.5, the critic should be permutation invariant. All operations used in this critic are either permutation equivariant or invariant: The edges in a Bipartite Pool densely connect the input and output points, making the MPL permutation invariant. The multi-aggregation is inherently invariant; the numerous FFNs operate on all points independently and are thus equivariant. The output FFN of the CNUs takes an equivariant input (from the first FFN) and an invariant input (from the multi-aggregation). Therefore, the CNUs are equivariant. As compositions of equivariant and invariant functions, the subcritics themselves are permutation invariant (Section 4.5).

5.3. Training and Implementation

The generator and critic are trained using an alternating series of one gradient step for the generator followed by one for the critic (referred to as `Ratio Gradient Steps`). The composition of the loss is detailed in the following section. Adam with `Weight Decay` is employed as the `Optimizer` (Section 3.2.3). A study of the learning rates revealed that the optimal rates were very low, specifically 10^{-5} for the generator and $3 \cdot 10^{-5}$ for the critic[4].

5.3.1. Loss

Feature Matching Loss

In addition to the Hinge loss (Eq. 3.28), the generator is also trained to minimize the feature matching loss [53]. For this loss, the activations of the layers within the critic are compared between a generated batch and a batch from the training dataset. The feature matching loss cannot directly operate on an unordered PC. Therefore, the input PC and the output of the Embedding and Pooling layers are multi-aggregated (Section 5.2.2) into vectors, which are then concatenated into a single vector $\mathbf{r}$. To ensure that all features in $\mathbf{r}$ are on a comparable scale, the mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$ are computed for each feature i on the simulation batch. $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are then used to rescale each feature of $\mathbf{r}$:

$$\mathbf{r}'_i = \frac{\mathbf{r}_i - \boldsymbol{\mu}_i}{\boldsymbol{\sigma}_i}.$$

This yields the following loss term[5]:

$$\mathbb{E}_{FM} = \mathbb{E}_{z\sim p_z, x\sim p_{\text{data}}} \left[\|\mathbf{r}'(x) - \mathbf{r}'(G(z))\|_1 \right]. \tag{5.1}$$

Model Loss

By default, Hinge Loss (Eq. 3.28) is used as the GAN objective, and the feature matching loss is scaled by a factor of 10 (Table 5.1). Altogether, this yields the following loss functions L_C and L_G for the critic and the generator:

$$L_C = - \sum_{i\in\text{Subcritics}} \mathbb{E}_{x\sim p_{\text{data}}}[\min(0, -1 + C_i(x))] + \mathbb{E}_{z\sim p_z}[\min(0, -1 - C_i(G(z)))], \tag{5.2}$$

$$L_G = - \sum_{i\in\text{Subcritics}} \mathbb{E}_{z\sim p_z}[C_i(G(z))] + 10 \cdot \mathbb{E}_{z\sim p_z, x\sim p_{\text{data}}} \left[\|\mathbf{r}'(x) - \mathbf{r}'(G(z))\|_1 \right]. \tag{5.3}$$

5.3.2. Selection of the Best Parameter Set

To monitor the training process, the metrics m are recorded at fixed `Validation intervals` on a validation set. The score $s_m(t)$ represents the fraction of recorded values that are higher (worse) than the value recorded at validation step t for a given metric

[4] Parts of this study have been repeated in Section 9.2.2, confirming the results.

[5] It should be noted that an L_1 normalization, rather than the L_2 normalization proposed by the authors, is employed to improve robustness against outliers.

m:

$$s_m(t) = \frac{1}{n} \sum_i^n \mathbb{1}_{m_i \geq m_t}.$$

This score maps the values for each metric to a range between 0 and 1. By averaging the score across all metrics, a qualitative ranking of the model configuration at the respective validation steps is obtained. If this averaged score is 1, the model configuration at this validation step achieves the best metrics across all validation steps. The model configuration with the highest averaged score is saved and used for evaluation.

The dataset-specific metrics are introduced in Section 6.3.

5.3.3. Early Stopping

To avoid unnecessary training, the process is halted when performance gains stagnate (Section 3.3.1). An "early stopping criterion" is introduced to terminate training when the criterion is met. For the DeepTree model, this criterion is defined as:

$$\bigwedge_m \left(f \cdot \min_{t \notin \text{window}} m_t < \min_{t \in \text{window}} m_t \right).$$

This criterion compares the minimum value of each metric inside and outside a window of recent validation times (`Early Stopping Window` in Table 5.1). If the smallest value of the metric is among the most recent values, i.e., inside the window, it indicates that the metric is improving. If the minimum within the window is greater for all metrics than the minimum outside, training is terminated. To avoid continuing the training for vanishingly small gains, the minimum outside the window is scaled down by a `Minimum Improvement` factor $f \in (0, 1)$. By default, this factor is 0.95, and the early stopping window comprises 1000 values, computed every 2000 critic steps. This window size ensures a sufficient number of gradient steps to prevent premature termination of training.

5.3.4. Hyperparameters of the Feed-Forward Neural Networks

Both the generator and the critic iteratively change the cardinality of the PC. Due to this iterative structure, both parts require numerous mappings with small input and output spaces, typically between 10 and 100. These mappings are provided by FFNs (Section 3.1). FFNs are sequences of linear (matrix) and non-linear (activation) functions, interleaved with normalization and dropout layers[6]. The first linear layer maps from the input space to the `Hidden Nodes`, and the last linear layer maps from the `Hidden Nodes` to the output space. The linear layers in between map between the same number of `Hidden Nodes`. `Dropout` (Section 3.3.2), if used, is always applied directly to the output of the linear layers.

5.3.5. Data Representation

Because the number of energy depositions (for particle showers) or constituents (for jets) varies, the cardinality of the PCs also changes (see Chapter 4). As a result, batches of these PCs cannot be represented by regular matrices. The DeepTree model utilizes the batch

[6]The model's performance may significantly depend on the order of these operations. A study on this ordering is presented in Section 9.2.

format implemented in the PyG GNN library [77]. In this format, the points of all PCs are stacked into a 2D matrix, while a 1D vector records, which PC each point belongs to[7]. The memory size of a batch is proportional to the cardinality.

This provides DeepTree with a computational advantage over other PC-based models (see Sections 4.6.1 to 4.6.4). Unlike the sparse representation, where PCs are frequently padded to the largest cardinality in the batch to enable stacking into a regular tensor, this method avoids unnecessary padding. In sparse representations, the added points are later excluded from computation using a mask. If the cardinality varies significantly[8], a large fraction of particles may need to be masked out. Thus, employing the PyG batch format offers computational benefits, while the implementation is more complex and time-consuming.

[7] See pytorch-geometric.readthedocs.io/en/2.5.3/advanced/batching.html, accessed 17.07.2024.
[8] For example, the cardinality in the CaloChallenge datasets varies widely, see Fig. 10.8.

5.4. Run Time and Memory Scaling

Although theoretical considerations can guide the design of a model with the desired run time behavior, determining the actual run time based solely on these considerations is infeasible due to the model's complexity, particularly in the generator. Additionally, the maximum cardinality that the model can produce is constrained by the GPU memory. Thus, run time and memory measurements are necessary[9].

For this measurement, 30 batches containing 10 PCs each are evaluated on an NVIDIA A100 80 GB PCIe GPU. The first three measurements are discarded as warmup runs, and the run time for the remaining 27 batches is averaged.

The cardinality is increased stepwise until the GPU runs out of memory. Unless stated otherwise, the default hyperparameters are used (Table 5.1).

It should be noted that in this generation scenario, the computed values of the nodes in the model were not stored, as they are not required for backpropagation Section 3.2.1. Therefore, in a training scenario, the required GPU memory will increase.

5.4.1. Generator

The cardinality of the PCs produced by the generator is the product of the branching factors. For this measurement, all branching factors are set to the same value, referred to as the *base b*. At each step, the branching factors are expanded by the base ([b], [b,b], [b,b,b], ...), increasing the cardinality by a factor of b (b^1, b^2, b^3, ...), until the GPU runs out of memory. Bases with values from 2 to 5 are investigated. Additionally, the generator is evaluated both with and without the Ancestor MPL (Section 5.1.2). Figure 5.7 presents the resulting run time and memory measurements. Assessing the scaling behavior with cardinality is challenging due to the limited number of data points available for each parameter set. However, the available points suggest a roughly linear scaling for both run time and allocated memory. For cardinalities exceeding 100k, the run time and memory plots align closely.

A lower base leads to a faster increase in run time and allocated memory and vice versa. Removing the Ancestor MPL reduces the median run time and allocated memory by approximately 46% for parameter sets that produce more than 100k points.

5.4.2. Critic

For the critic, the expected run time behavior is straightforward to determine: The number of messages, and therefore the run time in the first bipartite pooling, is proportional to the input cardinality (Section 5.2.1). After the first pooling operation, the cardinality becomes fixed, resulting in a constant run time. Thus, a linear run time behavior is expected. This prediction is confirmed by the measurements presented in Fig. 5.7, which demonstrate a linear scaling behavior for both run time and memory.

[9] The memory measurement is obtained using `torch.cuda.max_memory_allocated`, see pytorch.org/docs/2.0/generated/torch.cuda.max_memory_allocated.html. The run time refers to the elapsed time for the Python code ("Wall time"). The time for the transfer to GPU memory is not included.

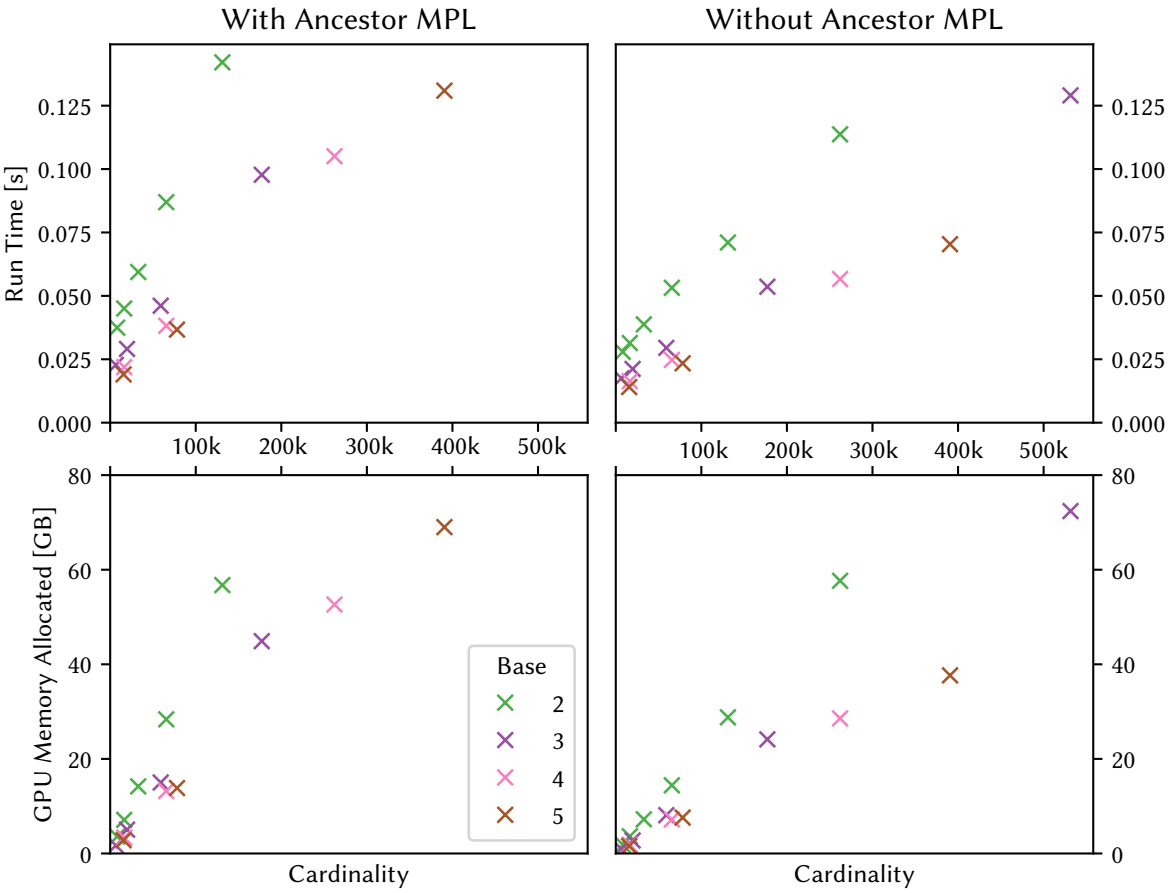


Figure 5.6. | Run Time and Allocated GPU Memory of the Generator for different Cardinalities.|
See Section 5.4.2 for the discussion. The base refers to the factor by which the cardinality is increased in each step. If the marker is a dot / cross, the generator is constructed with / without the Ancestor MPL (Section 5.1.2). Note, that 10 PCs are generated at the same time.

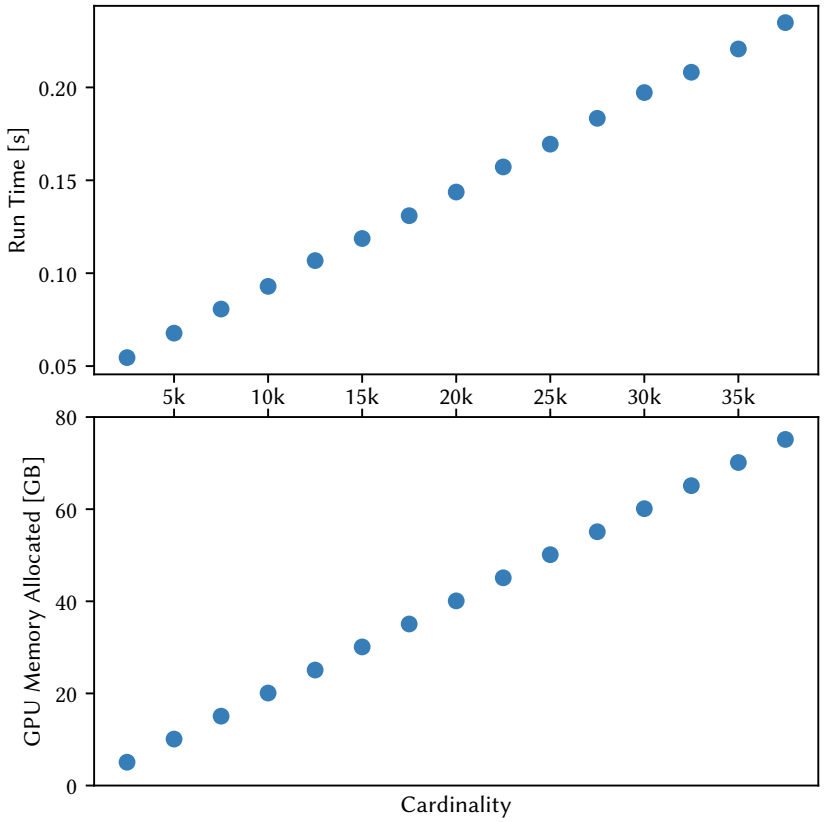


Figure 5.7. | Run Time and Allocated GPU Memory of the Critic for different Cardinalities.|
See Section 5.4.2 for the discussion. Note, that 10 PCs are evaluated at the same time.

5.4.3. Summary

Remarkably, the generator is capable of producing a batch with 10 PCs, each containing 500k points, simultaneously. The critic reaches its limit at just under 40k points. Even for a PC with an extremely high cardinality of 400k, the generator's run time is approximately 14 ms on an NVIDIA A100 GPU with 80 GB of memory. This clearly demonstrates the model's computational scalability, particularly for the generator, which is most critical for potential future deployment.

Chapter Review In this chapter, the DeepTree model, consisting of a generator and a critic, was introduced. The generator's central component is the branching layer, which iteratively upscales PCs in a tree-based manner. This enables the generator to map a single input vector to PCs with high cardinality, i.e., many points, while limiting the number of parameters (Milestone 1). This iterative upscaling in the generator is mirrored by the critic, where the bipartite pool iteratively downscales PCs (Milestone 3). The bipartite pool is not only differentiable but also permutation invariant and has favorable run time complexity. It can be applied not only to this model, but may also serve as a general-purpose embedding for PCs of arbitrary cardinality. The critic consists of multiple subcritics, each applied to a stage of the iterative downscaling. Each subcritic produces a separate output, and the loss is computed for each subcritic and summed. This approach of iteratively increasing/decreasing the cardinality in the generator/critic is a unique feature of the DeepTree model. Other models usually do not change the number of points, as detailed in Section 4.6.

CHAPTER 6

The JetNet Dataset

Chapter Abstract In this chapter, the JetNet datasets are introduced. These datasets exhibit distinct and complex distributions due to the intricate physics processes that generate them. Using sensitive and efficient metrics, these datasets will serve as benchmarks for assessing the fidelity of the DeepTree model.

JetNet [16, 17, 99] is a collection of publicly available datasets of jets, described by their constituents that have been produced through hadronization (Section 2.4). The datasets differ in the jet origin particle and in cardinality, i.e., the number of constituents (30 or 150). They are referred to as g jets for gluons, q jets for light quarks (up, down, or strange), and t jets for hadronically decaying top quarks, depending on the jet origin particle[1]. Each dataset contains approximately 170k individual jets, divided into 110k for training, 50k for testing, and 10k for validation.

Dataset Generation

Details of the dataset generation are provided in Ref. [99, Appendix B]: The events are based on proton-proton collisions with a center-of-mass energy of 13 TeV, matching the conditions of the LHC. The transverse momentum of the parton is constrained within a narrow window around 1 TeV. Initially, parton-level events are generated using MADGRAPH5_aMCATNLO 2.3.1 [118]. Subsequently, the decay and hadronization processes are simulated with PYTHIA 8.212 [119]. The resulting particles are then clustered using the anti-kT algorithm [28] with a distance parameter of $R = \sqrt{(\Delta\eta)^2 + (\Delta\phi)^2} = 0.8$, as implemented in FASTJET 3.1.3 [120]. The initially narrow parton p_T^{jet} spectrum is broadened by the parton shower and by particles not captured by the clustering algorithm. Jets with extreme fluctuations of p_T^{jet} outside the range [0.8 TeV, 1.6 TeV] are removed from the datasets.

[1] The most recent release [16, 17] adds jets initiated by W and Z bosons, though they are not used in this thesis.

6.1. Relative Jet Variables

The jet constituents are considered massless and are thus described by transverse momentum p_T, pseudorapidity η, and azimuthal angle ϕ:

$$\begin{pmatrix} E \\ p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} m_T \cosh y \\ p_T \cos\phi \\ p_T \sin\phi \\ m_T \sinh y \end{pmatrix} \approx \begin{pmatrix} p_T \cosh y \\ p_T \cos\phi \\ p_T \sin\phi \\ p_T \sinh y \end{pmatrix},$$

with the transverse mass $m_T = \sqrt{p_T^2 + m^2} \approx p_T$.

Jet Coordinates

The jet coordinates, denoted with the rel superscript, describe the constituents relative to the kinematic properties of the jet:

$$\begin{aligned} \eta_i^{\text{rel}} &= \eta_i - \eta^{\text{jet}}, \\ \phi_i^{\text{rel}} &= \left(\phi_i - \phi^{\text{jet}}\right) \mod 2\pi, \\ p_{T,i}^{\text{rel}} &= \frac{p_{T,i}}{p_T^{\text{jet}}}. \end{aligned}$$

Effectively, the jet axis is shifted to $\left(\eta^{\text{rel}}, \phi^{\text{rel}}\right) = (0, 0)$, and the jet's transverse momentum is normalized to 1. This adjustment makes jets with different axes directly comparable.

Relative Jet Variables

Based on these jet coordinates for the constituents, relative variables of the jet can be constructed. They are denoted with $\tilde{\square}$.

Mass The mass of a jet can be calculated as follows:

$$\begin{aligned} {m^{\text{jet}}}^2 &= \left(\sum_i E_i\right)^2 - \left(\sum_i p_{x,i}\right)^2 - \left(\sum_i p_{y,i}\right)^2 - \left(\sum_i p_{z,i}\right)^2 \\ &= \left(\sum_i p_{T,i} \cosh\eta_i\right)^2 - \left(\sum_i p_{T,i} \cos\phi_i\right)^2 - \left(\sum_i p_{T,i} \sin\phi_i\right)^2 - \left(\sum_i p_{T,i} \sinh\eta_i\right)^2, \end{aligned}$$

where the last step assumes that the constituents are massless. If the expression is computed for the jet coordinates defined above, this yields the relative mass

$$\begin{aligned} \tilde{m}^2 &= \left(\sum_i p_{T,i}^{\text{rel}} \cosh\eta_i^{\text{rel}}\right)^2 - \left(\sum_i p_{T,i}^{\text{rel}} \cos\phi_i^{\text{rel}}\right)^2 - \left(\sum_i p_{T,i}^{\text{rel}} \sin\phi_i^{\text{rel}}\right)^2 - \left(\sum_i p_{T,i}^{\text{rel}} \sinh\eta_i^{\text{rel}}\right)^2 \\ &= \left(\frac{m^{\text{jet}}}{p_T^{\text{jet}}}\right)^2. \end{aligned}$$

Transverse Momentum The transverse momentum of a jet $p_{\mathrm{T}}^{\mathrm{jet}}$ is given by:

$$p_{\mathrm{T}}^{\mathrm{jet}\,2} = p_x^{\mathrm{jet}\,2} + p_y^{\mathrm{jet}\,2} \qquad = \left(\sum_i p_{\mathrm{T,i}} \cos\phi_i\right)^2 + \left(\sum_i p_{\mathrm{T,i}} \sin\phi_i\right)^2 .$$

If $p_{\mathrm{T}}^{\mathrm{jet}}$ is computed with relative variables, the relative transverse momentum is obtained:

$$\tilde{p}_{\mathrm{T}}^2 = \left(\sum_i p_{\mathrm{T,i}}^{\mathrm{rel}} \cos\phi_i^{\mathrm{rel}}\right)^2 + \left(\sum_i p_{\mathrm{T,i}}^{\mathrm{rel}} \sin\phi_i^{\mathrm{rel}}\right)^2 .$$

In processing the dataset, the coordinates are first translated to the jet coordinates. Subsequently, the constituents are truncated to the 30/150 highest $p_{\mathrm{T}}^{\mathrm{rel}}$ constituents. The jet coordinates correspond to all constituents, while only the truncated jet is provided in the dataset. With each removed constituent, $\tilde{p}_{\mathrm{T}}$ shifts further from 1 towards 0, resulting in a continuous $\tilde{p}_{\mathrm{T}}$ distribution limited to a maximum of 1.

6.2. Analysis of the Dataset Features

Figures 6.1 to 6.5 show the distributions of various jet properties for the investigated datasets. These histograms are based on the full datasets, each containing approximately 170k jets for each jet origin particle.

6.2.1. Cardinality

In Fig. 6.1, the histograms of jet cardinality are shown. For JetNet-30, the overwhelming majority of jets are significantly affected by the truncation to 30 constituents, leading to a large fraction of jets being assigned to the cardinality 30 bin. q jets, compared to t jets and g jets, tend to have lower cardinality. Therefore, they are the least affected by the JetNet-30 truncation, losing on average only 11 (of 38) constituents (29%). The most severely affected are the g jets, which lose on average 37 (of 67) constituents (55%), followed by the t jets, which lose on average 33 (of 63) constituents (52%). In the JetNet-150 dataset, q jets and t jets are almost entirely contained within the 150 constituents limit, resulting in only a minor cutoff peak in the 150 constituents bin. The distribution of constituents for g jets, however, features a heavier tail, leading to a more pronounced cutoff peak.

6.2.2. Transverse Momentum

Constituents The distribution of p_T^{rel} across the constituents is shown in Fig. 6.2a. For all jet types, the JetNet-30 and JetNet-150 histograms coincide for all but the two lowest p_T^{rel} bins. This indicates that each dropped constituent contributes only a small fraction of p_T. A distribution with many constituents at high p_T^{rel} values implies that the momentum is concentrated in fewer constituents. Thus, the momentum is most concentrated in q jets, followed by g jets, and then t jets. All jets contain a large fraction of constituents with p_T^{rel} close to 0, having little influence on the jet properties. The vertical dashed lines indicate the average p_T^{rel} of the 31st constituent, i.e., the largest p_T^{rel} that is dropped for JetNet-30. This value is significantly lower for q jets (≈0.002) compared to g jets (≈0.0055) and t jets (≈0.0059), confirming the observed momentum concentration.

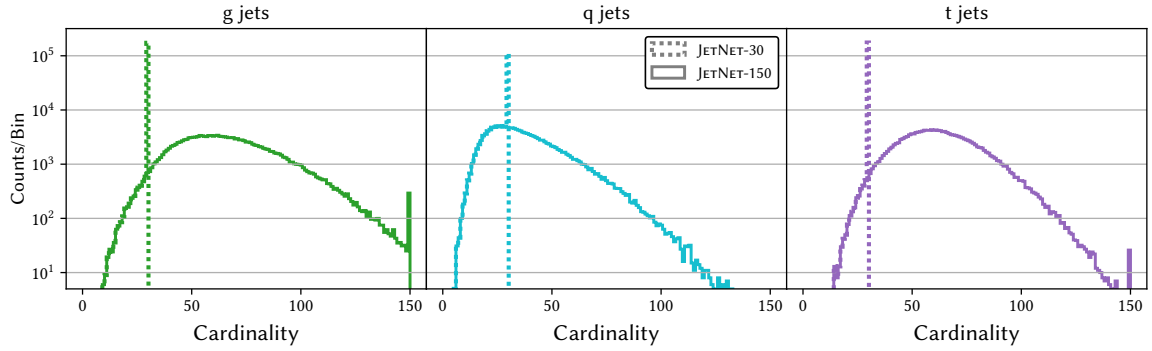
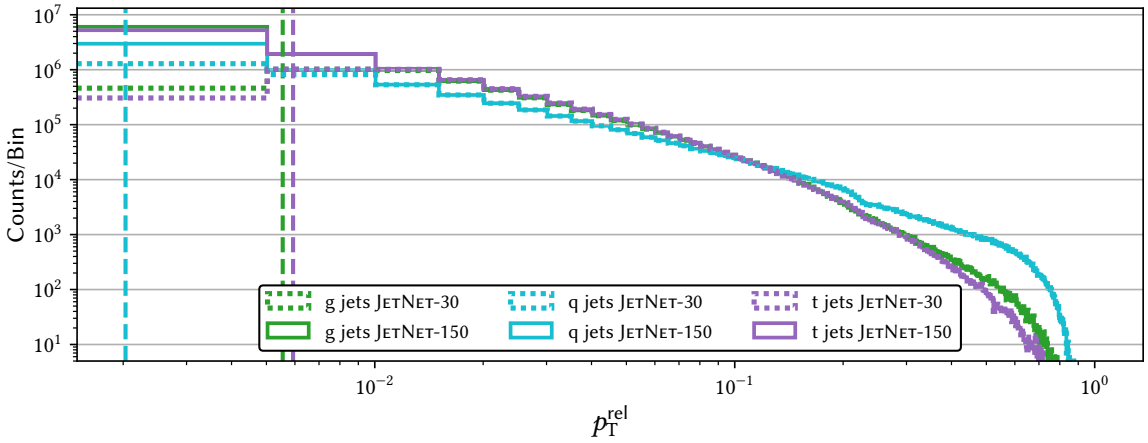
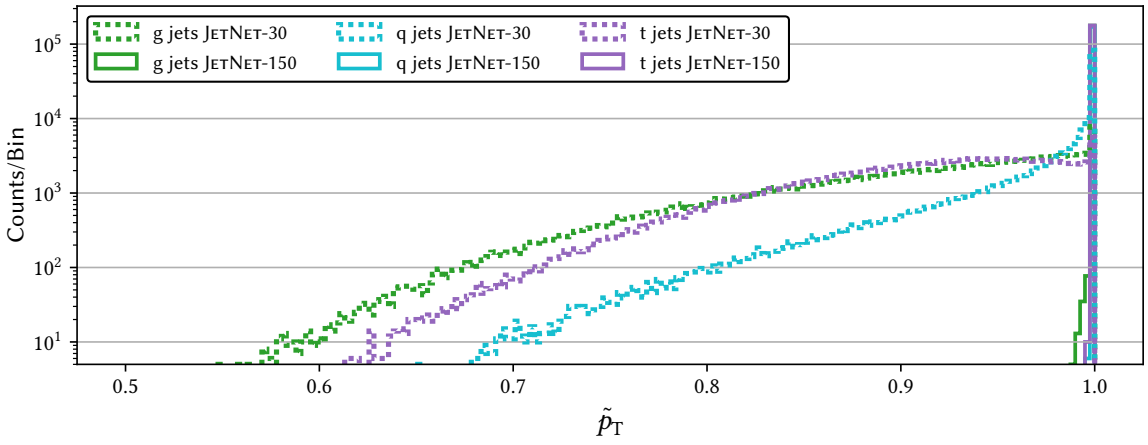


Figure 6.1. | The Cardinality Distribution of the Jets.|
See Section 6.2.1 for the discussion.



(a) Relative transverse momentum of the constituents. Additionally, the dashed horizontal lines show the average p_T^{rel} of the 31st largest p_T^{rel} constituent. For this average, jets with less than 31 constituents are counted as 0.



(b) Relative Transverse Momentum of the Jets.

Figure 6.2. | The p_T^{rel} Distribution of the Constituents and the $\tilde{p}_T$ Distributions of the Jets.| See Section 6.2.2 for the discussion.

Jets Figure 6.2b shows the $\tilde{p}_T$ distributions. Had none of the constituents been dropped, $\tilde{p}_T$ would remain at 1. As increasingly larger p_T^{rel} constituents are removed, $\tilde{p}_T$ shifts away from 1 and moves closer to 0, as explained in the previous chapter. The magnitude of this effect depends on the *number* of constituents dropped, which is determined by the cardinality distribution (Fig. 6.1), and the *amount* of momentum carried by the removed constituents, as determined by the p_T^{rel} distribution (Fig. 6.2a).

This results in a significant difference between the JetNet-30 and JetNet-150 distributions for $\tilde{p}_T$.

When the p_T^{rel} values are more evenly distributed among the constituents, the lowest p_T^{rel} constituents carry a larger fraction of p_T. Thus, their removal has a more substantial impact.

q jets Since q jets generally contain fewer constituents (Fig. 6.1) and their low p_T^{rel} constituents carry a smaller fraction of p_T (Fig. 6.2a), they are least affected by the cut to 30 constituents. As a result, their $\tilde{p}_T$ distribution in JetNet-30 closely resembles the JetNet-150 distribution, which is nearly singular at 1.

g jets In contrast, g jets contain so many constituents that they are significantly impacted not only by the cut to 30 constituents but also by the cut to 150 constituents (Fig. 6.1). Thus, g jets exhibit the largest tail toward 0 in the JetNet-30 distribution among the three jet types, with even some deviations from 1 in the JetNet-150 distribution.

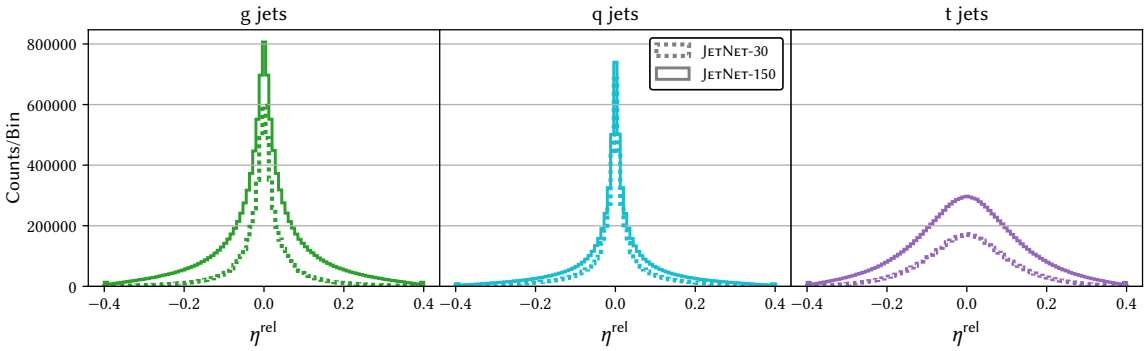
t jets The t jets are nearly fully contained within the 150 constituents, resulting in a $\tilde{p}_T$ distribution concentrated at 1 in JetNet-150. For JetNet-30, their $\tilde{p}_T$ distribution behaves similarly to that of the g jets, though its tail does not deviate as far from 1.

6.2.3. Azimuthal Angle and Pseudorapidity

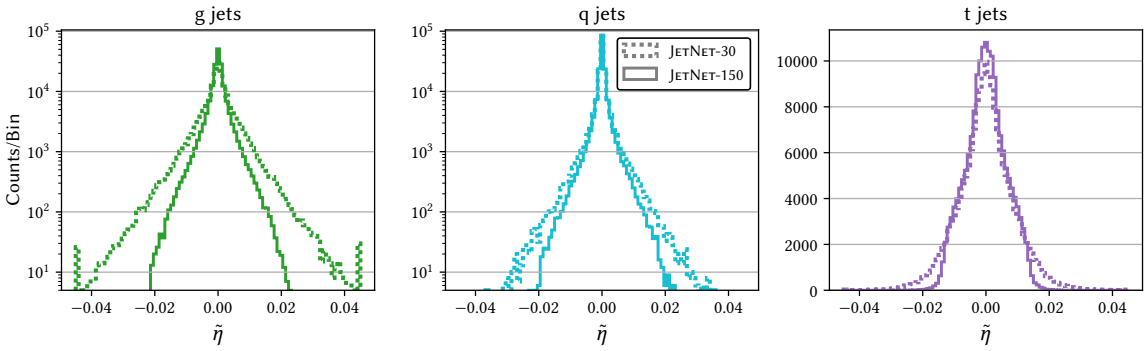
Figures 6.3a and 6.4a show the η^{rel} and ϕ^{rel} distributions for the constituents, while Figs. 6.3b and 6.4b present the resulting $\tilde{\eta}$ and $\tilde{\phi}$ distributions. All distributions are unimodal and symmetric.

Constituents Constituents located farther from the jet axis tend to have a lower p_T^{rel}, making them more likely to be excluded when moving from JetNet-150 to JetNet-30. As a result, the η^{rel} and ϕ^{rel} distributions are more concentrated around the jet axis in JetNet-30 compared to JetNet-150. Consequently, the distributions for JetNet-150 exhibit a sharper peak around 0 than those for JetNet-30.

Jets In contrast, the $\tilde{\eta}$ and $\tilde{\phi}$ distributions show the opposite effect. As constituents are dropped, the jet axis, previously centered at zero, becomes increasingly undetermined. This results in broader $\tilde{\eta}$ and $\tilde{\phi}$ distributions for JetNet-30 compared to JetNet-150, although the broadening is more pronounced in the $\tilde{\eta}$ distributions than in the $\tilde{\phi}$ distributions. The $\tilde{\phi}$ distributions differ significantly from the $\tilde{\eta}$ distributions. For JetNet-150, they are nearly singular around 0 across all jet types.

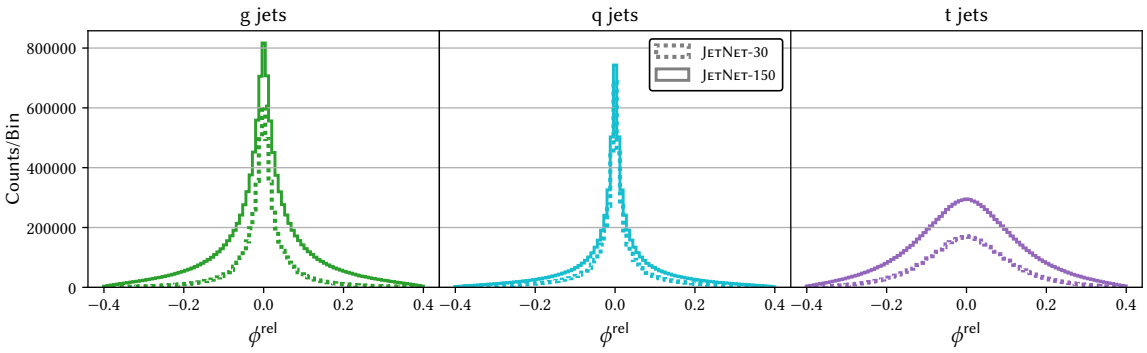


(a) Relative Pseudorapidity of the Constituents η^{rel}.

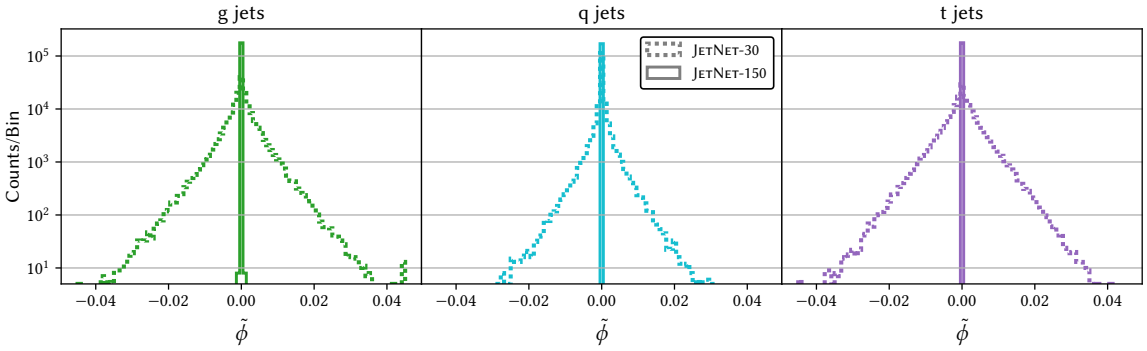


(b) Relative Pseudorapidity of the Jets $\tilde{\eta}$.

Figure 6.3. | The η^{rel} and $\tilde{\eta}$ Distributions of the constituents/jets.|
See Section 6.2.3 for the discussion.



(a) Relative Azimuthal angle of the Constituents ϕ^{rel}.



(b) Relative Azimuthal angle of the Jets $\tilde{\phi}$.

Figure 6.4. | The ϕ^{rel} and $\tilde{\phi}$ Distributions of the constituents/jets.|
See Section 6.2.3 for the discussion.

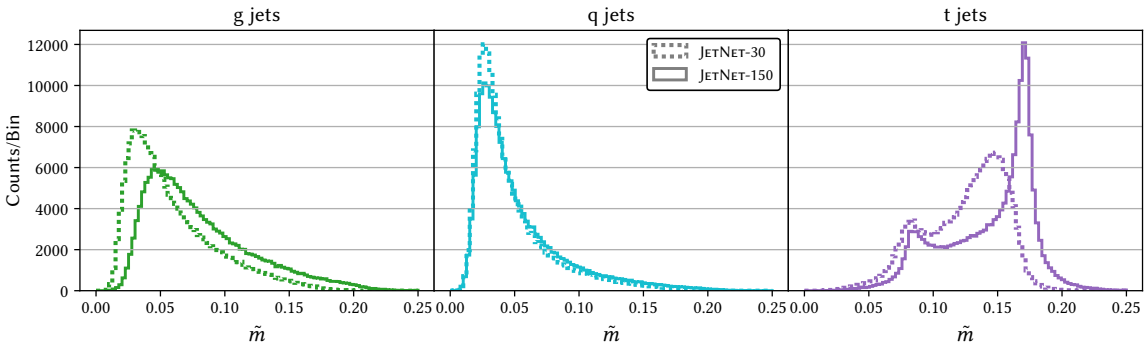


Figure 6.5. | The Distributions of the Relative Jet Mass $\tilde{m}$.|
See Section 6.2.4 for the discussion.

Jet Type Comparison In the η^{rel}, $\tilde{\eta}$, and ϕ^{rel} distributions, q jets show the highest density around 0, followed by g jets, with the t jets distributions being comparatively broader. For JetNet-30, both g jets and t jets show similar, narrow widths, while the distribution of q jets is even more concentrated around zero.

6.2.4. Mass

Figure 6.5 shows the (relative) mass distributions of the jets. The $\tilde{m}$ distributions of g jets and q jets are unimodal and peak at low values, while the t jets show a bimodal distribution. For q jets and g jets, the jet origin particle directly produces the jet, whereas t jets contain subjets produced by the decay products of top quarks. Top quark decay into a bottom quark (producing a subjet) and a W boson with an almost 100% branching ratio. The W boson can then decay into a quark pair, producing two additional subjets. The clustering algorithm may or may not capture all three of these subjets, resulting in the characteristic double peak structure [99]. When transitioning from JetNet-150 to JetNet-30, removing constituents shifts the mass distribution to lower values for all jet types. For q jets and g jets, this leads to a sharper peak as the mass distribution is bounded by 0 on the lower end. In the t jets $\tilde{m}$ distribution, both peaks shift to lower values, but only the higher mass peak is significantly broadened. This sharp double peak structure, as seen in the t jets mass spectrum, is challenging for generative models to reproduce. Thus, the t jet dataset serves as a particularly useful performance benchmark. After establishing the generator on all three datasets in JetNet-30, the development of the critic will focus solely on the t jet JetNet-150 dataset.

6.3. Fidelity Metrics

To evaluate the quality of the generative models, several metrics are employed in this thesis, which are explained in detail below. With the publication of the dataset [99], the W_1^{M}, W_1^{P}, W_1^{EFP}, and FPND metrics were provided, as described in the following section. In Ref. [121], the authors additionally propose FPD and KPD. All metrics have been implemented in the JetNet library [18]. As recommended by the library, a test set of 50,000 jets is used for the computation of these metrics in this thesis.

The lowest achievable value for a given metric can be estimated by computing the metric on a sample of 50,000 jets sampled from the training set. In the presentation of the results, this is referred to as the *Limit*.

Uncertainty Quantification

To quantify the statistical uncertainty arising from the finite training sample size, bootstrapping is applied. The test set and the generated set are sampled five times with replacement to create bootstrapped samples of the same size as the original test/generated set. The metrics are then evaluated for each sample. The results are reported as the mean ± standard deviation of the computed values. For the FPND, the authors of Ref. [99] chose not to compute an uncertainty but rather to compute the metric once on the full test set. To maintain comparability with this baseline, this approach is followed in this thesis.

6.3.1. Wasserstein Distance and Derived Metrics

The Wasserstein-p distance [59, Eq. 6.2] between two distributions P and Q is defined as

$$W_p(P,Q) = \inf_{\gamma \in \Pi(P,Q)} (\mathbb{E}_{(x,y)\sim\gamma}[\|x-y\|]^p)^{1/p}, \tag{6.1}$$

where $\Pi(P,Q)$ is the set of all distributions that have P and Q as marginal distributions. In other words, any[2] joint distribution that minimizes the distance between P and Q is sought. For $p = 1$, this distance is called the Earth Mover's Distance and can be interpreted as the amount of "work" required to shift between the distributions along the shortest path, in the form of the joint distribution γ. The Wasserstein distance fulfills the triangle inequality ($W_p(P,R) \leq W_p(P,Q) + W_p(Q,R)$), is symmetric ($W_p(P,Q) = W_p(Q,P)$), and is finite [122]. These are all desirable properties for defining a metric. In contrast, the commonly used KL divergence (Eq. 3.17) is not symmetric and becomes infinite if the distributions do not overlap[3].

A major downside of the Wasserstein distance is its large computational cost in high dimensions [123]. However, recent substantial advances have been made to achieve acceptable scaling behavior [124]. For one-dimensional variables or multinomial Gaussian distributions, explicit solutions exist, which will be presented in the following paragraphs.

General Approach by Minimizing the Pairing

Let $\underline{\mathbf{x}} = \{\mathbf{x}_1, ..., \mathbf{x}_n\}$ and $\underline{\mathbf{y}} = \{\mathbf{y}_1, ..., \mathbf{y}_n\}$ be two samples of equal size. Instead of finding the joint distribution γ, the distance is minimized over the pairings between the elements of

[2] In general, the optimal joint distribution is not unique.

[3] More precisely, the KL divergence becomes infinite if there is an area with positive measure where the density of Q is 0 and the density of P is not [122].

the samples [125, Lemma 4.2]:

$$W_p(\underline{\mathbf{x}}, \underline{\mathbf{y}}) = \min_{\sigma \in S_n} \left(\frac{1}{n} \sum_i^n \|\mathbf{x}_i - \mathbf{y}_{\sigma(i)}\|^p \right)^{1/p}, \tag{6.2}$$

where S_n is the set of all permutations. For large sample sizes, iterating over all possible pairings becomes computationally infeasible.

One Dimensional Case

If P and Q are real and one-dimensional, the minimization can be achieved by sorting the samples:

$$W_p(\underline{x}, \underline{y}) = \left(\frac{1}{n} \sum_i^n \|x_{(i)} - y_{(i)}\|^p \right)^{1/p}, \tag{6.3}$$

where $x_{(i)}$ and $y_{(i)}$ are the ith-smallest values in the sample.

Multinomial Gaussian Case

For two multinomial Gaussian variables $P \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ and $Q \sim \mathcal{N}(\boldsymbol{\mu}', \Sigma')$, an explicit solution exists for $p = 2$ for any dimension:

$$(W_2(P,Q))^2 = \|\boldsymbol{\mu} - \boldsymbol{\mu}'\|_2^2 + \mathrm{tr}\left(\Sigma + \Sigma' - 2\left(\Sigma^{1/2} \Sigma' \Sigma^{1/2} \right)^{\frac{1}{2}} \right), \tag{6.4}$$

where $\Sigma^{1/2}$ denotes the principal square root of Σ, i.e., the (unique) matrix fulfilling $\Sigma^{1/2}\Sigma^{1/2} = \Sigma$ and ${\Sigma^{1/2}}^T = \Sigma^{1/2}$.

1D Wasserstein Distance for Physics Variables

In this thesis, the Wasserstein-1 distance is applied to one-dimensional physics variables. The W_1^{M} and W_1^{P} are computed for the mass and $p_{\mathrm{T}}^{\mathrm{rel}}$ distributions, respectively. The W_1^{P} is a weighted average of the W_1 distances for $p_{\mathrm{T}}^{\mathrm{rel}}$, η^{rel}, and ϕ^{rel}. The standard deviation σ_κ is used to weight the means μ_κ and return a weighted average[4]. For $\mathrm{K} = \{\eta^{\mathrm{rel}}, \phi^{\mathrm{rel}}, p_{\mathrm{T}}^{\mathrm{rel}}\}$, the W_1^{P} is defined as

$$W_1^{\mathrm{P}} = \frac{\sum_{\kappa \in \mathrm{K}} \mu_k w_\kappa}{\sum_{\kappa \in \mathrm{K}} w_\kappa} \pm \left(\sum_{\kappa \in \mathrm{K}} w_\kappa \right)^{-1/2}, \tag{6.5}$$

with

$$w_\kappa = \sigma_k^{-2} = \frac{N}{\sum_i^N (x_i - \bar{x})^2}.$$

This weighted average not only combines the three metrics into one, but also improve robustness: As a higher standard deviation leads to a lower weight, a W_1^κ with high variance has less influence on the W_1^{P} than one with low variance.

[4] Proposed by Benno Käch.

6.3.2. Energy Flow Polynomials

Energy Flow Polynomials (EFPs) [126] provide a complete linear basis of infrared and collinear safe [127] jet observables. Achieving a good match on a set of EFPs indicates a well-performing model. For a graph G with edges $E(G)$ and N nodes, the EFP is defined as:

$$\mathrm{EFP}_G = \sum_{i_1=1}^{M} \cdots \sum_{i_N=1}^{M} z_{i_1} \cdots z_{i_N} \prod_{(k,l)\in E(G)} \theta_{i_k i_l},$$

where M is the jet cardinality, the z_i represent the momenta or energies of the constituents, and the $\theta_{i_k i_l}$ denote the angular distances between the constituents.

Each EFP corresponds to a graph[5] G, where each node in the graph is assigned to one of the M jet constituents. Each edge is then assigned the angular distance θ_{ij} between the nodes it connects. In the case of a hadron collider simulated in JetNet, they are given by

$$\theta_{ij} = (|\eta_i - \eta_j|^2 + |\phi_i - \phi_j|^2)^{\beta/2} \text{ for a } \beta > 0 \text{ and } z_i = p_{\mathrm{T},i}^{\mathrm{rel}},$$

where β is chosen as 1. The product of these angular distances is then multiplied by the energy (or $p_{\mathrm{T}}^{\mathrm{rel}}$) fraction of each of the nodes. Finally, the sum of this product is computed for each possible assignment of nodes to jet constituents. In this way, each EFP is permutation invariant and includes all particles.

Due to the high computational costs, the EFPs evaluated in the JetNet library for the computation of W_1^{EFP} are restricted to connected graphs[6] with 4 nodes and 4 edges ($4 \leq |E(G)|, 4 \leq N$). For the resulting EFPs, the Wasserstein-1 distances are computed and combined using the weighted average (Eq. 6.5), as for W_1^{P}.

6.3.3. Metrics based on the Fréchet Inception Distance

Fréchet ParticleNet Distance

The Fréchet ParticleNet Distance (FPND) applies the concept of the Fréchet Inception Distance [128] (FID) to jets instead of images. FID compares high-dimensional images by passing them through a classifier, such as the Inception classifier [129], and analyzing the activations in a selected linear layer. These activations are assumed to follow a multinomial Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}, \Sigma)$. The Fréchet distance or Wasserstein-2 distance is then calculated using Eq. 6.4. To adapt this principle for jets, a pretrained ParticleNet [130] model is utilized. ParticleNet is a permutation-invariant regression or classification network. In ParticleNet, the input PC is processed through a sequence of `EdgeConv` [131] blocks, followed by a summation of the points and a mapping to the output classes via an FFN. The activations from the first linear layer after the pooling layer are used to compute $\boldsymbol{\mu}$ and Σ. This model has been trained to classify t, g, and q jets on JetNet-30. Since ParticleNet accepts a maximum of 30 constituents, the PCs of JetNet-150 must be truncated to 30 constituents to compute this metric.

[5] More specifically, a multigraph without self-loops, i.e., a graph that allows multiple edges between the same nodes but no edge from a node to itself.

[6] I.e., a graph where there exists a sequence of edges that connects any two nodes in the graph.

Fréchet Physics Distance

The Fréchet Physics Distance (FPD) calculates the Fréchet distance based on a set of 36 EFPs, rather than the activations of layers in ParticleNet. All EFPs of graphs with four or fewer edges are included, which is a superset of the EFPs used to compute W_1^{EFP}. The distribution of these EFPs is assumed to follow a multinomial Gaussian distribution, allowing the Fréchet distance to be computed using Eq. 6.4. Studies in Ref. [121] indicate that the FPD may be even more sensitive than the FPND.

However, the FPD would exhibit a bias that decreases with increasing sample size. To obtain an unbiased metric for any sample size, the value of the metric is extrapolated to an infinite sample size [132]. Specifically, the FPD is evaluated for bootstrapped samples of varying sizes[7], and a linear fit is performed on the inverse of the sample size. In Ref. [121], it is demonstrated that the resulting intercept provides an unbiased metric.

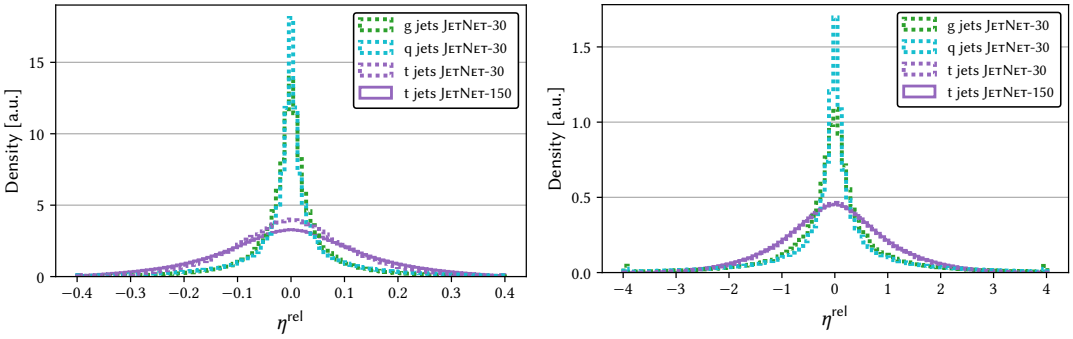
Further Metrics

In addition to the metrics presented here, the Kernel Physics Distance, Maximum Mean Discrepancy, and coverage were proposed in Ref. [121]. As these did not prove to be as sensitive as the other metrics, they are omitted from the studies in this thesis.

[7] Using 10 samples with sizes between 20k and 50k.

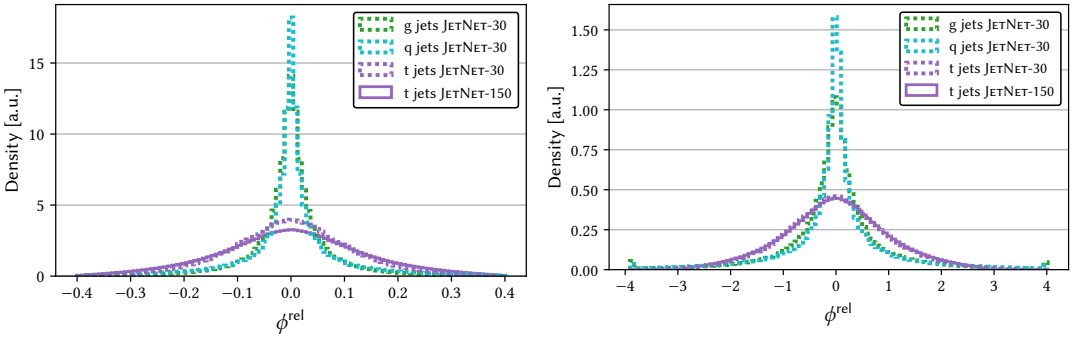
6.4. Pre- and Postprocessing

Without any scaling, the input variables to a NN may span many orders of magnitude. For effective convergence of NNs, it is advantageous to apply a transformation that shifts the mean of all input variables to 0 and scales the variance to 1 [76]. To generate samples, the inverse transformation is applied to the output of the generator. The transformations employed here are implemented in Scikit-learn [133].



(a) Without the Transformation. **(b)** With the Transformation.

Figure 6.6. | The η^{rel} Distribution of the Simulation Datasets.|
See Section 6.4 for the discussion.



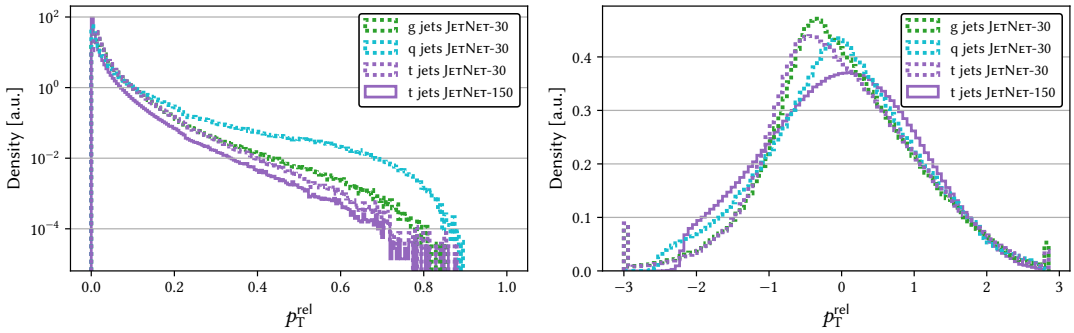
(a) Without the Transformation. **(b)** With the Transformation.

Figure 6.7. | The ϕ^{rel} Distribution of the Simulation Datasets.|
See Section 6.4 for the discussion.

The Figs. 6.2a, 6.3a and 6.4a show the input and output distributions for the $p_{\text{T}}^{\text{rel}}$, ϕ^{rel}, and η^{rel} variables. For the training, η^{rel} and ϕ^{rel} are scaled separately to mean 0 and standard deviation 1 using the `StandardScaler` from sklearn [133].

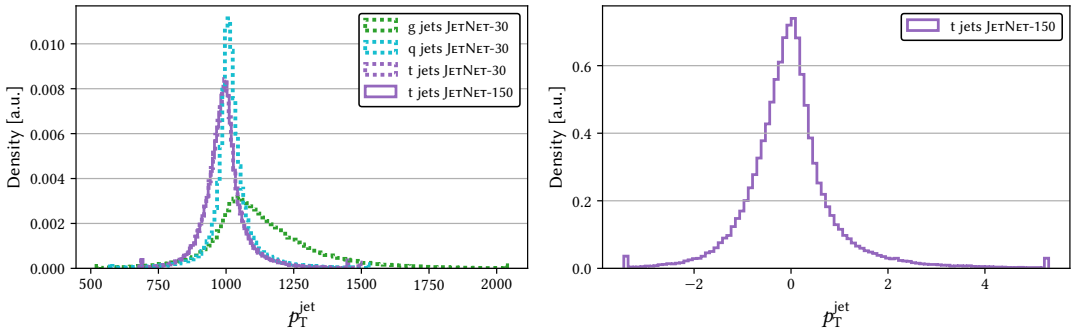
$$x_i \to \frac{x_i - \mu}{\sigma} \text{ with } \mu = \frac{1}{N}\sum_i x_i \text{ and } \sigma = \sqrt{\frac{1}{N}\sum_i |x_i - \mu|^2}. \tag{6.6}$$

The distribution of $p_{\text{T}}^{\text{rel}}$ is strictly positive and steeply falling. To transform it to a symmetric unimodal distribution, it is first transformed using a Box-Cox transformation [134] and then scaled to a standard normal distribution, that is shown in Fig. 6.8.



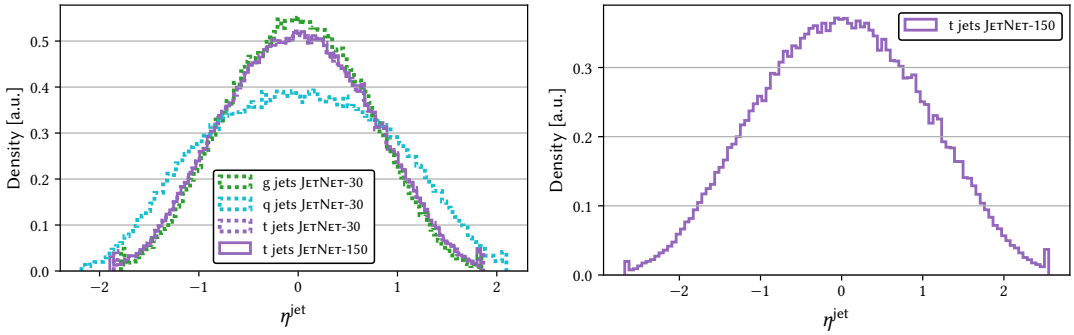
(a) Without the Transformation. **(b)** With the Transformation.

Figure 6.8. | The p_T^{rel} Distribution of the Simulation Datasets.|
See Section 6.4 for the discussion.



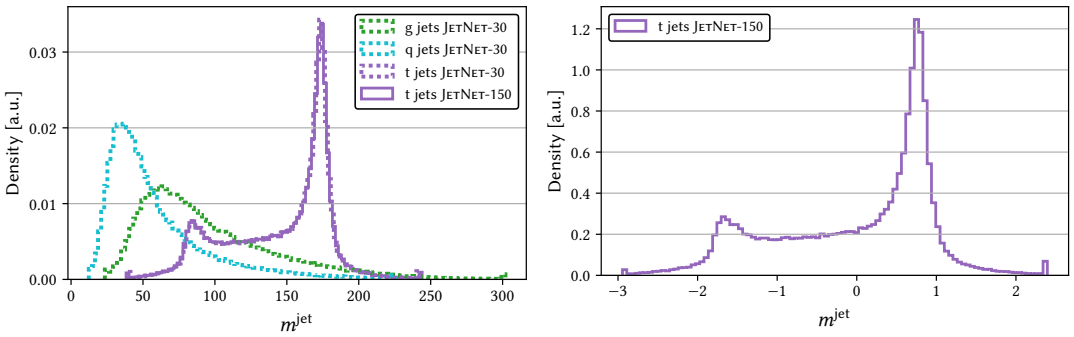
(a) Without the Transformation. **(b)** With the Transformation.

Figure 6.9. | The p_T^{jet} Distribution, used as a Conditioning Variable, before and after the Transformation.|
See Section 6.4 for the discussion. As the transformation for the conditional variables was only introduced for the training on JetNet-150 t jets, the right-hand plot only shows this dataset.



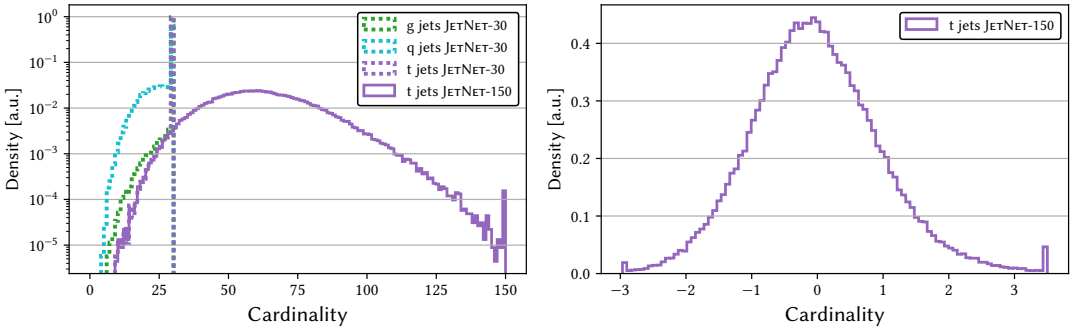
(a) Without the Transformation. **(b)** With the Transformation.

Figure 6.10. | The η^{jet} Distribution, used as a Conditioning Variable, before and after the Transformation.|
See Section 6.4 for the discussion. As the transformation for the conditional variables was only introduced for the training on JetNet-150 t jets, the right-hand plot only shows this dataset.



(a) Without the Transformation. **(b)** With the Transformation.

Figure 6.11. | The m^{jet} Distribution, used as a Conditioning Variable, before and after the Transformation.|
See Section 6.4 for the discussion. As the transformation for the conditional variables was only introduced for the training on JetNet-150 t jets, the right-hand plot only shows this dataset.



(a) Without the Transformation. **(b)** With the Transformation.

Figure 6.12. | The Cardinality Distribution, used as a Conditioning Variable, before and after the Transformation.|
See Section 6.4 for the discussion. As the transformation for the conditional variables was only introduced for the training on JetNet-150 t jets, the right-hand plot only shows this dataset.

Conditioning Variables

The Figs. 6.9 to 6.12 show the distributions of $p_{\mathrm{T}}^{\mathrm{jet}}$, η^{jet}, m^{jet}, and the cardinality. If the generator is provided with critical jet variables, such as $\tilde{m}$, it does not need to learn the distribution of these variables. Instead, it merely needs to learn how to produce jet constituents that add up to a jet with the given properties.

This approach reduces the significance of metrics computed on these jet variables and complicates comparisons to other models. However, the generator must be provided with the cardinality to produce a PC of the correct size. Thus, the generator is conditioned only on the cardinality, as shown in Fig. 6.12. The critic, on the other hand, is not used during generation and may be conditioned on any variable.

JetNet-30 For the training on JetNet-30, the jet type, $p_{\mathrm{T}}^{\mathrm{jet}}$, η^{jet}, m^{jet}, and the cardinality are provided to the MP-GAN discriminator, as described in Ref. [99]. Following the training procedure of MP-GAN, no transformations are applied to the conditioning variables in JetNet-30. Therefore, the transformations of these variables are not shown in the respective plots.

JetNet-150 In the transition from the JetNet-30 dataset to JetNet-150 t jets, transformations are introduced to the conditioning variables. The Figs. 6.9 to 6.12 show the transformed distributions of $p_{\mathrm{T}}^{\mathrm{jet}}$, η^{jet}, m^{jet}, and cardinality provided to the DeepTree critic. For the transformations of $p_{\mathrm{T}}^{\mathrm{jet}}$, η^{jet}, and m^{jet}, standard scaling (Eq. 6.6) is applied. For cardinality, the "Index Transformation", developed in Section 10.3.1, is employed. These transformations move the distributions much closer to the shape of a standard normal distribution.

6.5. Jet Momentum Rescaling

At the beginning of this section, the rescaling of the variables of the constituents is described. Through this rescaling, $\tilde{p}_{\mathrm{T}}$ can be approximated by $\sum_i p_{\mathrm{T,i}}^{\mathrm{rel}}$:

$$\tilde{p}_{\mathrm{T}}^2 = \left(\sum_i p_{\mathrm{T,i}}^{\mathrm{rel}} \cos\phi_i^{\mathrm{rel}}\right)^2 + \left(\sum_i p_{\mathrm{T,i}}^{\mathrm{rel}} \sin\phi_i^{\mathrm{rel}}\right)^2 .$$

Thus, for $\phi_i^{\mathrm{rel}} \approx 0$

$$\tilde{p}_{\mathrm{T}} \approx \sum_i p_{\mathrm{T,i}}^{\mathrm{rel}}.$$

For JetNet-150, this yields an almost singular $\sum_i p_{\mathrm{T,i}}^{\mathrm{rel}}$ distribution, as shown in Fig. 6.13. While the two nearly singular $\sum_i p_{\mathrm{T,i}}^{\mathrm{rel}}$ and $\tilde{p}_{\mathrm{T}}$ distributions are easily recognized by the critic, they present a significant challenge for the generator to model. To facilitate the generator's task, its output should be normalized in the same manner. To compute $p_{\mathrm{T}}^{\mathrm{rel}}$, the four-momentum vectors of the constituents would need to be summed. By rescaling $\sum_i p_{\mathrm{T,i}}^{\mathrm{rel}}$ instead of $\tilde{p}_{\mathrm{T}}$, this calculation and the backpropagation through the η^{rel} of each component can be avoided. The effect of rescaling $\sum_i p_{\mathrm{T,i}}^{\mathrm{rel}}$ to 1 is shown in Fig. 8.2. To

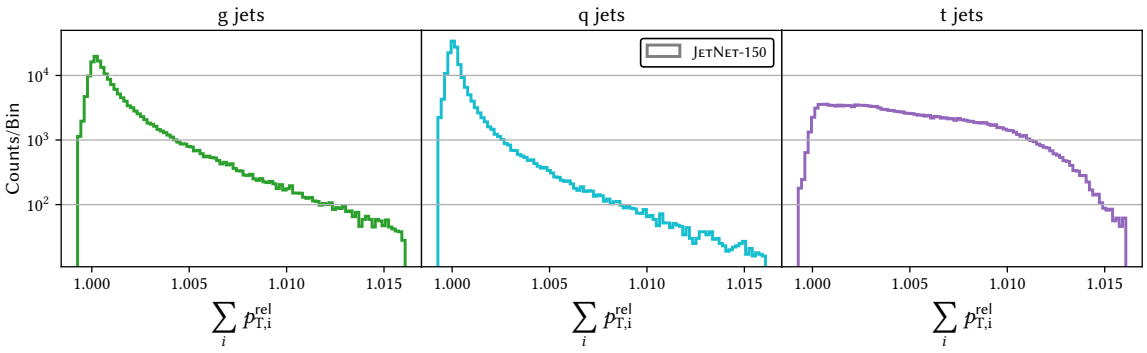


Figure 6.13. | The $\sum_i p_{T,i}^{rel}$ Distribution of the Jets in JetNet-150.|
See Section 6.5 for the discussion.

perform this rescaling, the inverse transformation must first be applied to the p_T^{rel} values, after which they need to be rescaled and transformed again. All these computation steps must be differentiable, so that the gradient from the critic can pass through them and be used to update the generator. The PyTorch implementation of this rescaling is described in Appendix D.1.

Chapter Review In this chapter, the JetNet datasets, along with the metrics for these datasets, are introduced. Moreover, the necessary pre- and postprocessing steps are presented. With these sensitive metrics and available benchmarks, the JetNet datasets serve as an ideal proxy task for generative models for particle showers.

CHAPTER 7

Generator Development on JetNet-30

Chapter Abstract In this chapter, the proof-of-concept for the DeepTree generator is described, utilizing a novel, tree-based upscaling method for PCs (Milestone 2). To achieve this, the generator is combined with an established critic and applied to JetNet-30. The model demonstrates performance that is competitive with the MP-GAN baseline and shows a clear advantage over tree-GAN.

This study was published in Ref. [106].

As a first step in model development, it is essential to establish that the generator is capable of modeling complex dependencies between particles. The points in the PC represent the constituents of the jet, and thus their properties determine the distributions of these points. Therefore, the ability to model complex dependencies between constituents can be tested by investigating the modeling of jet properties, such as jet mass. The generator's ability to scale to higher cardinality is established in the following chapters.

7.1. Study Setup

For each of the three jet types, the model was trained separately.

Choice of the Critic Since the viability of the tree-based PC generation approach needed to be established first, the DeepTree critic was developed after this study. Therefore, the critic from the MP-GAN baseline model [99] was used, as it had already demonstrated its effectiveness. Numerous previous attempts to use other critics, including PointNet [135], ParticleNet [130], and the r-GAN [108] critic, were not successful.

Cardinality Cut For this study, the cardinality cut described in Section 5.1 was omitted. As a result, the produced PCs always contain 30 constituents. Given that only a small fraction of jets has fewer than 30 constituents (see Fig. 6.1), the impact of these surplus constituents on performance is expected to be minimal.

Hyperparameters Other deviations from the default hyperparameters (Table 5.1) are summarized in Table 7.1. For this study, the `CyclicLR` [136] learning rate scheduler was employed. However, since it did not significantly benefit the training, the scheduler was removed in subsequent studies. It should also be noted that the generator in this study employs the permutation-equivariant variant of the branching mechanism, which is further detailed in Appendix C.1.

Table 7.1. | Hyperparameters Different from the Default for Training DeepTree on JetNet-30.|
See Section 7.2 for the discussion. For the default hyperparameters, see Table 5.1.

Common		Ratio Gradient Steps	1:2
Generator (DeepTree)		Loss	Hinge (Eq. 3.28)
		Node Features by level	[64,33,20,10,3]
		Branching Factor by level	[1,2,3,5,5] ($\prod b_i = 30$)
		Branching Mechanism	Equivariant (Appendix C.1)
	Scheduler	Name	`CyclicLR` [a]
		`base_lr`	10^{-5}
		`cycle_momentum`	`false`
		`max_lr`	10^{-4}
		`step_size_up`	10^{4}
Critic (MP-GAN)		Optimizer	Adam ($\beta_1 = 0.0, \beta_2 = 0.9$)
		Learning rate	$3 \cdot 10^{-4}$
		Condition	[jet type, cardinality, p_T^{jet}, η^{jet}, m^{jet}][b]

a As provided by PyTorch: pytorch.org/docs/2.0/generated/torch.optim.lr_scheduler.CyclicLR.html, accessed 10.07.2024.

b The critic is trained for each jet type separately. Therefore, jet type is a constant value that could have been omitted, but this would have deviated from the original critic architecture.

7.2. Generated Distributions

Distribution of the Coordinates of the Constituents

In Fig. 7.1, the distributions of the jet coordinates of the constituents — p_T^{rel}, η^{rel}, and ϕ^{rel} — generated by DeepTree are compared to those from the test dataset. Overall, the distributions show a good match across the datasets, though some deviations are observed in the tails. These disagreements will be quantified later in this section (Table 7.2).

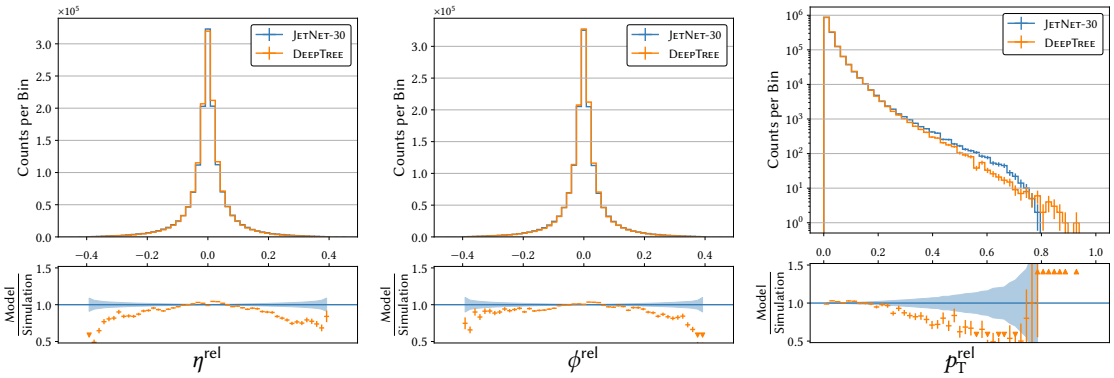
Distribution of the Jet Variables

The jet variables, shown in Fig. 7.2, are computed by summing the four-momentum vectors of the massless constituents and calculating $\tilde{\eta}$, $\tilde{p}_T$, and $\tilde{m}$, as described in Section 6.1. As these variables combine all constituents, they are more sensitive to mismodeling than the marginal variables η^{rel}, ϕ^{rel}, and p_T^{rel}.

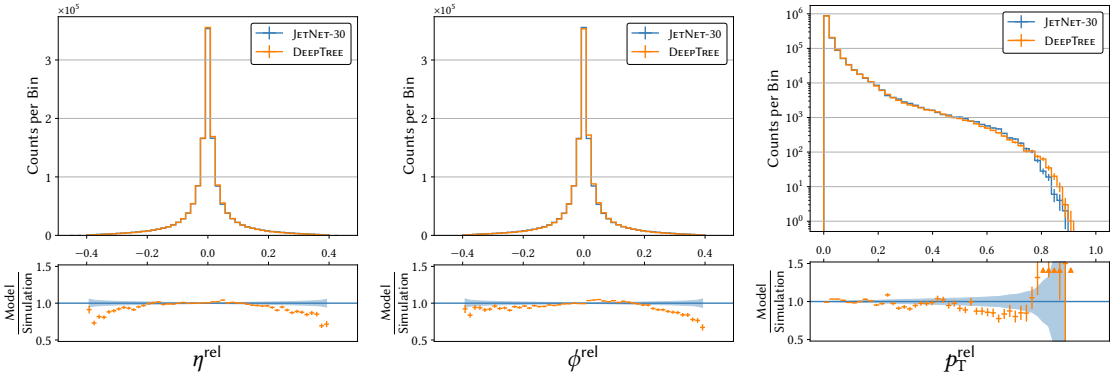
Pseudorapidity Overall, the model reproduces the distributions of the jet variables well. For all three jet types, the $\tilde{\eta}$ distribution is unimodal and symmetric. In the case of g jets and q jets, the $\tilde{\eta}$ distribution features a sharp peak at 0, which the model struggles to replicate precisely. However, the modeling is more accurate for t jets, where the peak is broader.

Transverse Momentum Due to the scaling described in Section 6.1, $\tilde{p}_T$ is normalized to 1. The truncation of constituents reduces the momentum, shifting the $\tilde{p}_T$ of the affected jets from 1 to a value less than 1 (Fig. 6.2a). The resulting $\tilde{p}_T$ distribution is characterized by a slowly rising shape, which is abruptly cut off at 1. While the model successfully reproduces the rising part of the distribution, it struggles to model the cut-off peak accurately. The tail of the distributions is accumulated in an overflow bin, indicated by the framed bar at the right end of the distribution. This issue is particularly pronounced for the q jet distribution, which shows the sharpest cut-off peak. This cut-off feature was artificially introduced by first converting to relative jet variables and then truncating the jets' constituents to 30 (Section 6.1). Distributions that combine a continuous and a singular distribution are especially challenging to model and are unlikely to be encountered in a real-world setting. Therefore, modeling this variable does not serve as a critical performance test of the model.

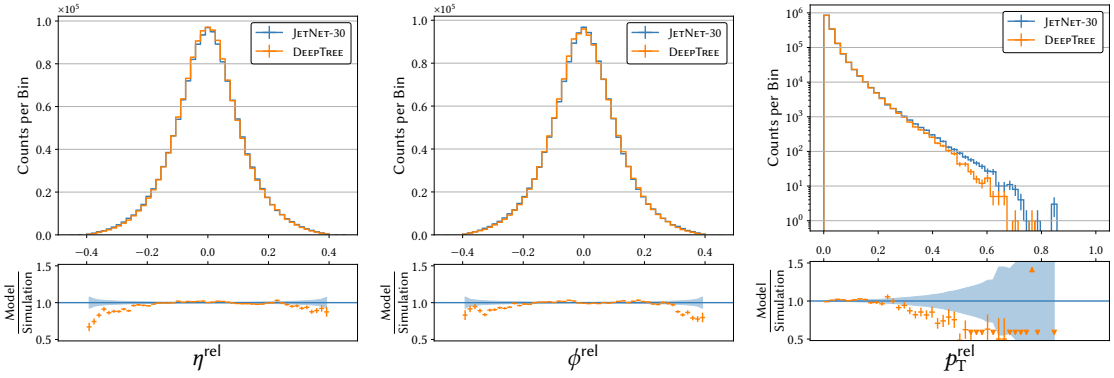
Mass In contrast, the jet mass, represented by $\tilde{m}$, is a crucial physics variable as it reconstructs the mass of the jet's origin particle. The $\tilde{m}$ distributions for g jets and q jets show a sharp turn-on followed by an exponentially falling tail. Meanwhile, the t jets distribution presents a double peak spectrum, as described in Section 6.2.4. The model successfully reproduces all of these jet mass distributions.



(a) g jets

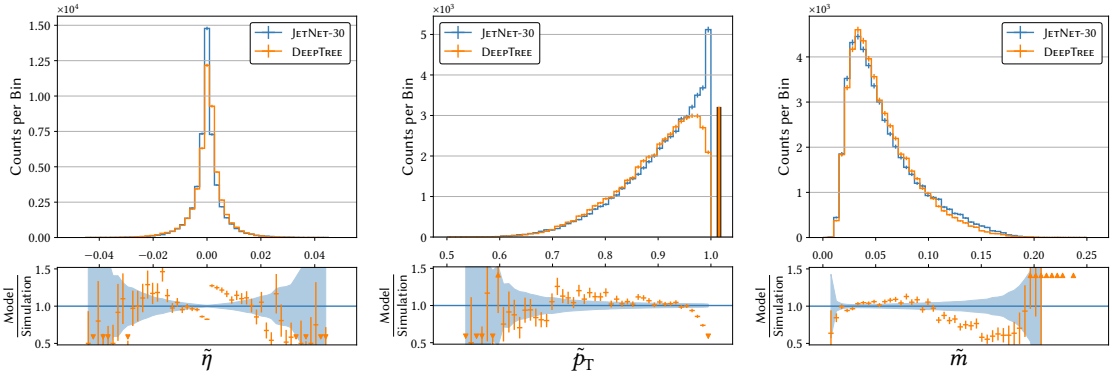


(b) q jets

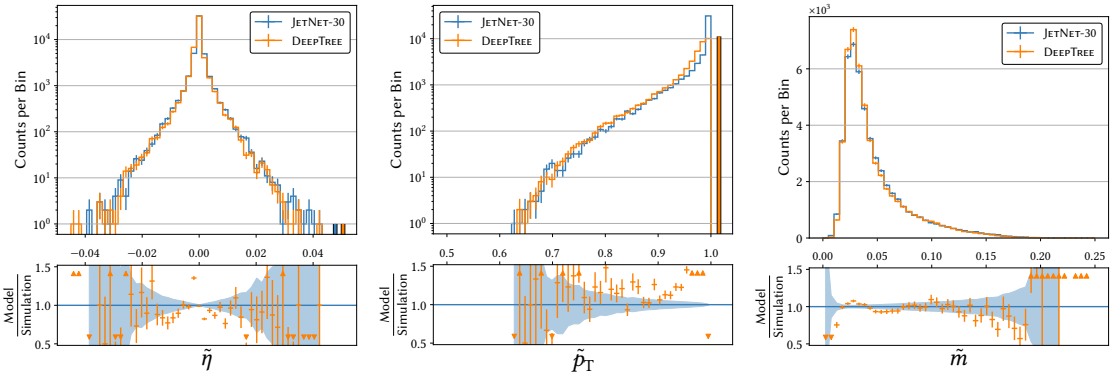


(c) t jets

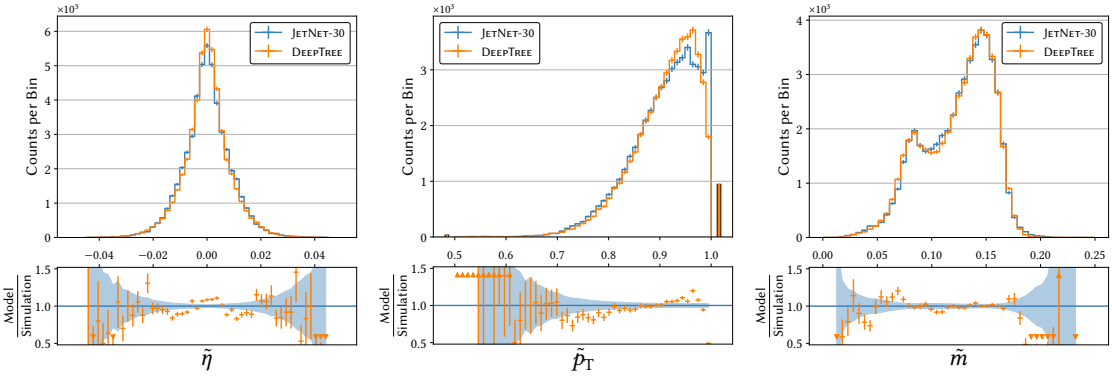
Figure 7.1. | Distributions of the Kinematic Variables of the Jet Constituents for the three Jet types in the JetNet-30 Test Datasets.|
See Section 7.2 for the discussion. The plot design is explained in Appendix A.1. Tables 5.1 and 7.1 contain the hyperparameters for these trainings.



(a) g jets



(b) q jets



(c) t jets

Figure 7.2. | Distributions of the Jet Variables for the three Jet types in the JetNet-30 Test Datasets.|
See Section 7.2 for the discussion. The plot design is explained in Appendix A.1. Tables 5.1 and 7.1 contain the hyperparameters for these trainings. Note that the $\tilde{p}_T$ and $\tilde{\eta}$ plots for q jets have been logarithmically scaled for better visibility.

7.3. Achieved Metrics and Benchmark

A quantitative comparison using the introduced metrics is provided in Table 7.2, where DeepTree is compared to the MP-GAN and tree-GAN baselines. Overall, the DeepTree generator performs similarly to the MP-GAN generator: While DeepTree shows higher fidelity, indicated by lower metrics, on the top quark dataset and at least comparable performance on the light quark dataset, its performance is worse for the gluon dataset. The DeepTree generator demonstrates a significant advantage over the tree-GAN generator across all metrics for each jet type with any of the two critics investigated. The sole exception is the W_1^{M} for the g jets, where the models perform similarly. At this stage, the generator always produces the full 30 constituents for each of the jets. Therefore, the performance for jets with fewer than 30 constituents is limited, as is frequently the case for the q jets (Fig. 6.1). In subsequent studies, the generator will produce a variable cardinality sampled from the dataset (Section 5.1).

The model weights and code for this study are available on GitHub[1].

[1] github.com/DeGeSim/chep23DeepTreeGAN

Table 7.2. | Comparison of the Proposed DeepTree to the MP-GAN Baseline.|
See Section 7.3 for the discussion. See Section 4.6.1 for the description of MP-GAN. The metrics and their computation are presented in Section 6.3. Lower values indicate better performance for all metrics. The uncertainty is reported as the standard deviation of the metrics calculated on the bootstrapped samples. The best central value of a model for a given metric is highlighted in bold. The "Limit" row represents the in-sample distance measured by sampling datasets from the training dataset and calculating the metrics. To ensure comparability with [99], bootstrapping is omitted in the computation of FPND. Thus, no uncertainty is provided. Note that the MP-GAN and tree-GAN models were selected for W_1^{M}, and the testing sample was used to compute the given metrics. The values for MP-GAN and tree-GAN are taken from Ref. [99, Fig. 3].

Dataset	Generator /Critic	$W_1^{\mathrm{M}}(\times 10^3)$	$W_1^{\mathrm{P}}(\times 10^3)$	$W_1^{\mathrm{EFP}}(\times 10^5)$	FPND
Gluon	Limit	0.7 ± 0.2	0.44 ± 0.09	0.63 ± 0.07	0.01
	MP-GAN/MP-GAN	$\mathbf{0.7 \pm 0.2}$	$\mathbf{0.9 \pm 0.3}$	$\mathbf{0.7 \pm 0.2}$	**0.12**
	DeepTree/MP-GAN	1.8 ± 0.2	1.6 ± 0.6	2.1 ± 0.2	0.34
	tree-GAN/ParticleNet	1.7 ± 0.1	4.0 ± 0.4	4 ± 1	84
	tree-GAN/MP-GAN	2.4 ± 0.2	12 ± 7	18 ± 9	69
Light Quarks	Limit	0.5 ± 0.1	0.5 ± 0.1	0.46 ± 0.04	0.01
	MP-GAN/MP-GAN	$\mathbf{0.6 \pm 0.2}$	4.9 ± 0.5	$\mathbf{0.7 \pm 0.4}$	0.35
	DeepTree/MP-GAN	0.9 ± 0.2	$\mathbf{1.7 \pm 0.3}$	$\mathbf{0.9 \pm 0.2}$	**0.15**
	tree-GAN/ParticleNet	10.1 ± 0.1	5.7 ± 0.5	4.1 ± 0.3	11
	tree-GAN/MP-GAN	4.8 ± 0.2	33 ± 6	10 ± 2	148
Top Quarks	Limit	0.51 ± 0.07	0.55 ± 0.07	1.1 ± 0.1	0.01
	MP-GAN/MP-GAN	$\mathbf{0.6 \pm 0.2}$	2.3 ± 0.3	2 ± 1	0.37
	DeepTree/MP-GAN	$\mathbf{0.6 \pm 0.1}$	$\mathbf{1.1 \pm 0.5}$	$\mathbf{1.3 \pm 0.3}$	**0.14**
	tree-GAN/ParticleNet	5.19 ± 0.08	9.1 ± 0.3	16 ± 2	17
	tree-GAN/MP-GAN	13.4 ± 0.4	45 ± 7	50 ± 30	66

Chapter Review In this chapter, a proof-of-concept for the PC upscaling approach used in the generator is presented on a jet dataset containing up to 30 jet constituents. The achieved metrics demonstrate that the DeepTree generator has a significant advantage over the tree-GAN generator and is competitive with the MP-GAN baseline. Thus, Milestone 2 has been achieved.

CHAPTER 8

Critic Development on JetNet-150

Chapter Abstract In this chapter, the DeepTree model is applied to the JetNet-150 dataset, which contains up to 150 jet constituents. At the same time, the capabilities of the novel DeepTree critic are demonstrated. This critic was not available during the training on JetNet-30 in the previous chapter. The model demonstrates competitive fidelity compared to other benchmark models (Milestone 4).

This study was published in Ref. [107].

8.1. Study Setup

As described in Section 6.2.4, the t jets have a complex structure and are particularly challenging to model. This provides highly sensitive metrics for benchmarking the model's capabilities. Thus, this study focuses exclusively on t jets.

The hyperparameters are set to match the defaults given in Table 5.1.

Generator The generator now employs branching factors of 2, 3, 5, and 5, producing a total of 150 points. Additionally, the generator now uses the default branching method instead of the equivariant method[1] used in the previous study on JetNet-30. Furthermore, the cut on cardinality is now applied, as described in Section 5.1.

Critic Due to the $\mathcal{O}(n^2)$ time and memory complexity of the MP-GAN critic, employing it on a dataset with such increased cardinality is infeasible. It is replaced by the DeepTree critic (Section 5.2). The DeepTree critic is conditioned on $\tilde{p}_T$, $\tilde{\eta}$, and $\tilde{m}$.

[1] In the default branching method, each parent is first mapped to b times its size and then split into b children (Section 5.1).
In the equivariant branching method, each parent is first split into b children. An FFN then maps the children to the size of their parents (Appendix C.1).

8.2. Generated Distributions

Distribution of the Coordinates of the Constituents

The distribution of the variables of the constituents is shown in Fig. 8.1. As in the previous study on JetNet-30, the generated distributions closely match the dataset distributions, with some deviations observed in the tails. While the η^{rel} and ϕ^{rel} distributions show a Gaussian-like shape, the $p_{\text{T}}^{\text{rel}}$ distribution is steeply falling, making it more challenging to model. Therefore, the successful modeling of $p_{\text{T}}^{\text{rel}}$ is particularly encouraging.

Distribution of the Jet Variables

Figure 8.2 shows the distributions of the jet variables.

Pseudorapidity Compared to the η^{rel} distribution, the $\tilde{\eta}$ distribution has a much narrower width around 0. Note the change in the range of the plot from [-0.4, 0.4] for η^{rel} to [-0.02, 0.02] for $\tilde{\eta}$. Thus, the observed shift in the generated distribution is still significant, but occurs on a small scale.

Transverse Momentum In the test dataset, $\tilde{p}_{\text{T}}$ is normalized while all constituents are available (Section 6.1). Due to this normalization, the $\tilde{p}_{\text{T}}$ distribution effectively forms a delta distribution at 1 for JetNet-150, as shown in Fig. 6.2b. To calculate the gradients, $\sum_i p_{\text{T,i}}^{\text{rel}}$ is scaled to 1 after the generator, as described in Section 6.5. As expected, the figure shows that this scaling still results in a $\tilde{p}_{\text{T}}$ distribution concentrated very close to 1, but with a finite width. Although the figure may suggest a drastic difference between the generated and data distributions, the normalization of $\tilde{p}_{\text{T}}$ deviates from 1 by at most 2% for both distributions.

Mass Compared to JetNet-30, the t jet mass distribution in JetNet-150 shows a much sharper top mass peak. This, combined with the potential contribution of up to 120 additional constituents to the mass, presents a significant modeling challenge. In the generated mass distribution, the double peak structure is clearly visible, though the model does not produce a peak as sharp as that observed in the test dataset.

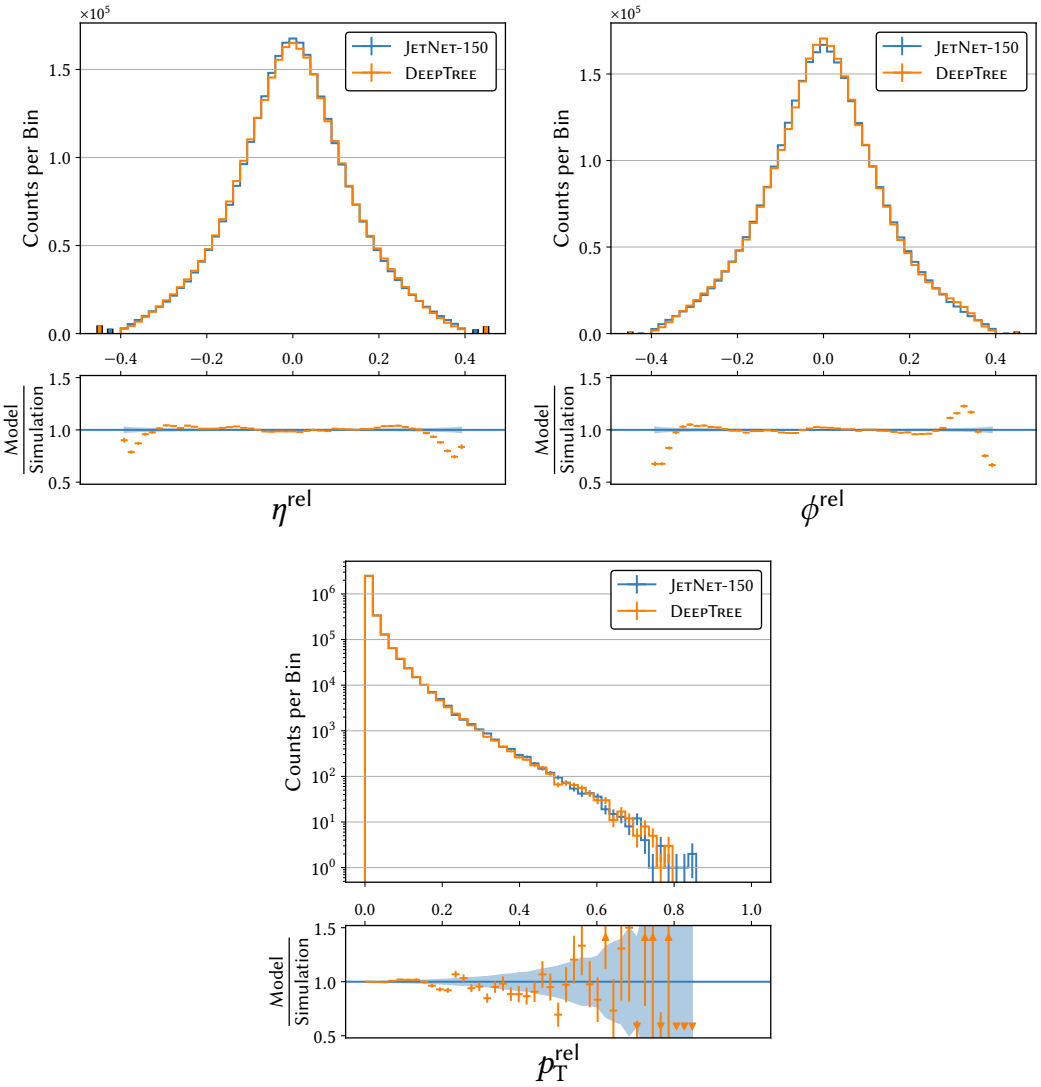


Figure 8.1. | Distributions of the Variables of the Constituents for the t jets in the JetNet-150 Test Dataset.|
See Section 8.2 for the discussion. The plot design is explained in Appendix A.1. The generated distributions match the dataset distribution well, but for some deviations in the tails.

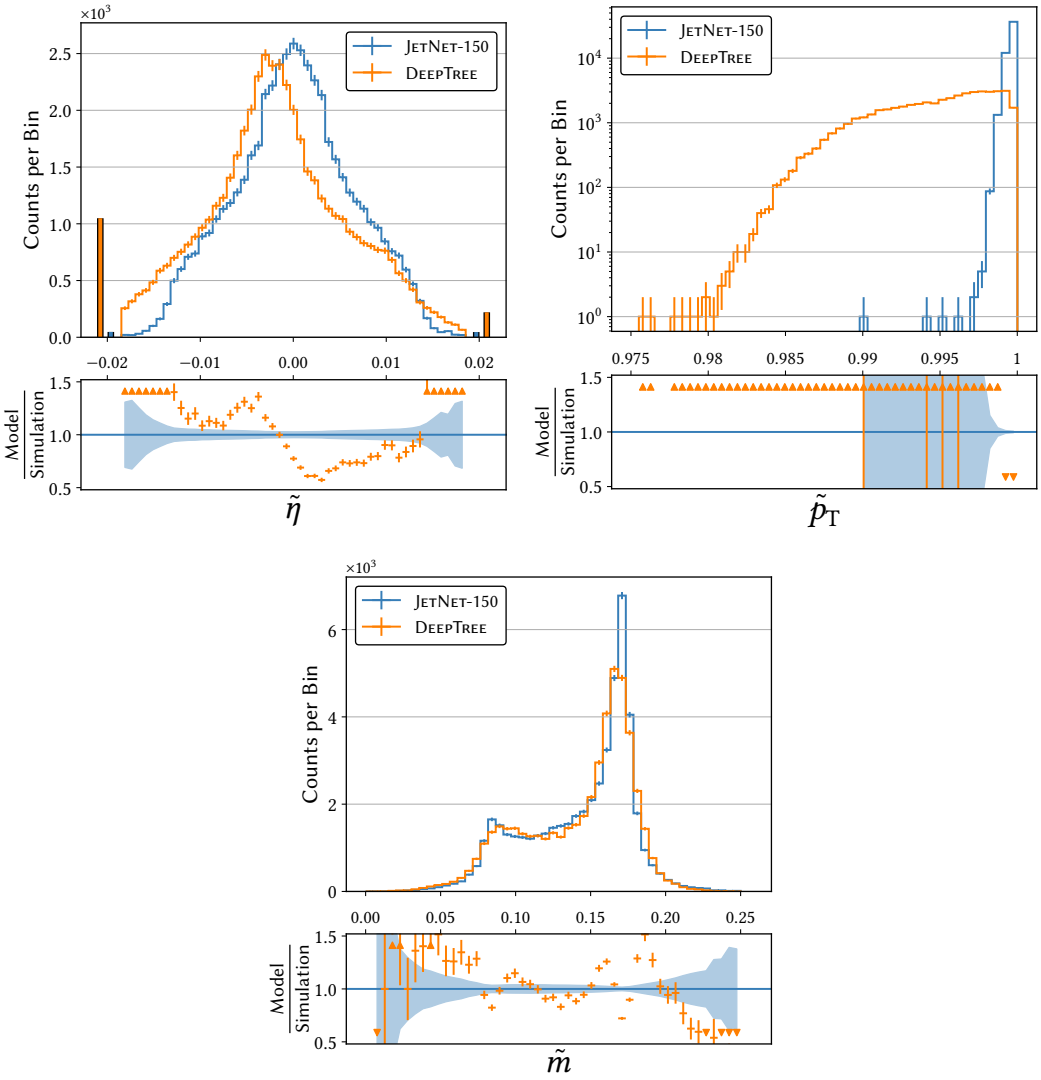


Figure 8.2. | Distributions of the Jet Variables for the t jets in the JetNet-150 Test Dataset.| See Section 8.2 for the discussion. The plot design is explained in Appendix A.1. Each variable has been computed from the sum of the four-vectors of the constituents, making them highly sensitive to small deviations in the modeling of these constituents. For example, in the distribution of $\tilde{p}_T$, the subtle differences in normalization approaches can be clearly visualized, especially given the very fine scale.

8.3. Achieved Metrics and Benchmark

In Table 8.1, DeepTree is compared to other state-of-the-art GANs, introduced in Section 4.6.3 and Section 4.6.2. While DeepTree achieves the best FPD, MDMA achieves the best W_1^{M}, W_1^{EFP}, and – by a narrow margin – W_1^{P}. Thus, DeepTree is shown to be competitive in this context. Within the ablation study in the following chapter, a model configuration with minor hyperparameter adjustments was found that further narrows the gap between MDMA and DeepTree (Table 9.5). Diffusion and flow-matching based models [137, 138] have recently provided state-of-the-art results for the JetNet-150 dataset, but sample production is drastically slower compared to the GAN approaches. Therefore, they were not considered in this comparison. The DeepTree code and weights for this study are available on GitHub[2].

Table 8.1. | Comparison of the Proposed DeepTree to EPiC-GAN and MDMA.|
See Section 8.3 for the discussion. See Sections 4.6.2 and 4.6.3 for the description of EPiC-GAN and MDMA. The metrics and their computation are presented in Section 6.3. Lower values indicate better performance for all metrics. The uncertainty is reported as the standard deviation of the metrics calculated on the bootstrapped samples. The best central value of a model for a given metric is highlighted in bold. The "Limit" row represents the in-sample distance measured by sampling datasets from the training dataset and calculating the metrics. Values for EPiC-GAN and MDMA taken from Ref. [102, Table 1].

Model	$W_1^{\mathrm{M}}(\times 10^3)$	$W_1^{\mathrm{P}}(\times 10^3)$	$W_1^{\mathrm{EFP}}(\times 10^5)$	FPD$(\times 10^4)$
Limit	0.42 ± 0.09	0.12 ± 0.04	1.22 ± 0.32	1.2 ± 0.6
EPiC-GAN	0.69 ± 0.08	0.65 ± 0.03	2.67 ± 0.39	22 ± 1
MDMA	$\mathbf{0.57 \pm 0.09}$	$\mathbf{0.10 \pm 0.02}$	$\mathbf{2.12 \pm 0.64}$	5.3 ± 0.9
DeepTree	1.49 ± 0.04	0.13 ± 0.02	5.01 ± 0.08	$\mathbf{3.4 \pm 0.7}$

[2] github.com/DeGeSim/nips23DeepTreeGANv2

Chapter Review In this chapter, the DeepTree model is applied to a jet dataset containing up to 150 jet constituents. Instead of the established critic used in the preceding study, the DeepTree critic, which has better time complexity, is employed. This critic features a novel PC downscaling method, whose capabilities are tested with this study. The DeepTree model demonstrates its fidelity by achieving competitive metrics within the benchmark. Thus, Milestone 4 has been achieved.

CHAPTER 9

Ablation Study

Chapter Abstract In this chapter, the design choices of the DeepTree model (Milestone 5) are reviewed. A systematic review is necessary to understand, which of these choices are advantageous. To quantify the impact of these choices, a method was developed to express the metrics on a common, meaningful scale. For this purpose, a baseline configuration is trained multiple times. Each modified configuration is trained once. It is then tested whether the metrics produced by the modified configuration can be excluded from the population of baseline trainings.

9.1. Evaluation Method

Ideally, all possible combinations of choices would be evaluated to select the optimal configuration, but this is computationally infeasible. Instead, a single change is introduced to the model at a time. If many of the introduced modifications result in either worse or similar performance, it suggests that the model configuration is already well optimized. In addition to improved performance, more stable training (e.g., through regularization) and a less complex architecture are also desirable. Modifications that achieve these goals are therefore considered more favorable.

Dataset and Metrics Although this model is ultimately designed for calorimeters, conducting an ablation study on the CaloChallenge dataset, introduced in the following chapter, would demand excessive resources and lacks the sensitive metrics available for JetNet-150. Therefore, the performance is evaluated using the JetNet-150 t jets dataset. The configuration in Table 5.1 is used as the baseline.

Computational Considerations As described in Section 5.3.2, the model parameters with the best performance on the validation set are used for testing. To improve the stability of parameter selection, the size of the validation set is increased from 10k to 25k jets. Consequently, the training set is reduced by 15k jets, resulting in a total of approximately 95k jets, which may negatively impact performance. For this study, stability of

the results is prioritized over absolute performance. To manage the significant computational demands, training is terminated at epoch 2500, unless stopped earlier by Early Stopping (Section 5.3.3). For comparison, the model presented in Chapter 8 was trained for approximately 4000 epochs. Therefore, the metrics produced in this study cannot be directly compared to those in Section 8.3.

Run Time Measurements The run time of the generator and critic was measured on an NVIDIA Tesla A100 GPU. The post-processing time of approximately 0.1 ms per jet was omitted. The same batch size used during training was maintained for these measurements. The memory required per jet differs between training and jet generation. As described in Section 3.2, during training, the activations of each layer must be stored in memory to calculate the gradients during the backward pass. Additionally, the critic is not evaluated during jet generation. In a generation scenario, the batch size could potentially be increased by a factor of about 8[1]. This increase could reduce the overhead fraction and result in more significant changes in the observed run time.

9.1.1. Quantifying the Significance of Changes in the Metrics

The training of these investigated models — and, to some extent, the computation of the metrics — is a stochastic process. It is important to quantify whether the metric $\tilde{x}$ from a modified training indicates a significant change in performance compared to the baseline trainings. The distribution X of the metric for baseline trainings is inherently unknown. If the distribution were known, the cumulative distribution function (CDF) could be used to determine how likely the metric is to take a value $\leq \tilde{x}$, assuming the metric was produced using the baseline configuration:

$$\Pr(X \leq \tilde{x}) = \mathrm{CDF}(\tilde{x}) \tag{9.1}$$

A value close to 0 (1) would suggest that it is unlikely to obtain such a low (high) metric with the baseline configuration, indicating an improvement (deterioration). Since the CDF is not available, it can be approximated by the empirical CDF (eCDF) [139, Sec. 15.4]. To achieve this, the model is trained multiple times using the baseline configuration. Given the sample of metrics produced by these baseline trainings $\bar{x}$, the eCDF is computed as follows:

$$\mathrm{eCDF}(\tilde{x}) = \frac{1}{n} \sum_{x \in \bar{x}} \mathbb{1}_{x \leq \tilde{x}}. \tag{9.2}$$

A major advantage of eCDFs is that they map any distribution to the same [0,1] scale. In the following ablation studies, eCDFs will be applied to the metrics to make them directly comparable. Because eCDFs are discontinuous functions, the metrics will be evaluated on a linear interpolation of the eCDFs to obtain a continuous performance measure.

Distribution of Metrics for the Baseline Trainings The baseline configuration is trained 13 times. The eCDFs for all metrics from these trainings are presented in Fig. 9.1. The metrics are plotted on the x-axis, with the corresponding eCDF on the y-axis. The median of the metric for the baseline trainings is indicated by the blue vertical line, which

[1] This factor was estimated by taking the ratio of the peak GPU memory usage during generation to that during training.

intersects the eCDF at $y = 0.5$. The ratio between the worst and best recorded metric is substantial, exceeding a factor of 5 for FPD.

9.1.2. Ablation Result Figures

In the following sections, the results of the ablation studies are displayed, for instance, in Fig. 9.3. The performance metrics evaluated are W_1^{M}, W_1^{P}, W_1^{EFP}, FPD, and FPND, as detailed in Section 6.3. Additionally, the run time of the generator, the critic, or the full model is provided in milliseconds, depending on which part of the model is affected by the change. For each configuration, a single training is conducted, and the metrics are computed. These metrics are then normalized to the [0,1] range using the interpolated eCDFs derived from the baseline trainings. Configurations are listed in the rows, with metrics given in the columns. Each cell in the ablation figures contains two numbers: the first is the value mapped with the eCDF, and the second is the actual metric value. The first column is an exception, where the cells display the average eCDF for the respective row/configuration. The cells are color-coded according to the eCDF value, ranging from blue (0, better than the baseline trainings) through white (0.5, median) to red (worse than the baseline trainings). eCDF values within the [0.2, 0.8] range are assumed to indicate no significant change in training performance. Therefore, it is expected that 20% of the trainings will yield an eCDF above 0.8 and below 0.2 purely by chance. In the last column, the run time for the part of the model affected by the change is provided for a single PC[2]. The median of the baseline trainings is presented in the first row.

Average of the eCDFs of the Metrics

Since the mapped values are all within the [0,1] range, the five performance metrics can be easily averaged (shown in the "$\overline{\text{Metrics}}$" column in the ablation figures). Averaging provides a more robust measure against fluctuations compared to relying on a single performance metric. However, because the performance metrics are correlated, this average can only mitigate fluctuations in the performance measurement, not fluctuations inherent in the training process. If the metrics from a training were independent, $\overline{\text{Metrics}}$ would represent the sum of five independent variables, and the distribution of $\overline{\text{Metrics}}$ would approximate a Gaussian distribution. A low value in one metric typically indicates successful training and is expected to correspond with low values in the other metrics for the same training, and vice versa. Therefore, a positive correlation between the metrics is expected. Table 9.1 shows the Pearson correlation coefficients between the metrics. All but the $W_1^{\mathrm{P}}/W_1^{\mathrm{EFP}}$ coefficient are positive. Thus, $\overline{\text{Metrics}}$ is not the sum of independent variables, and cannot be assumed to be Gaussian.

In Fig. 9.2, the histogram of $\overline{\text{Metrics}}$ for the baseline trainings is shown. Ten values fall within the range of 0.5 to 0.71. The remaining three trainings have $\overline{\text{Metrics}}$ values of 0.17, 0.17, and 0.27. The range from the second-lowest to the second-highest value is [0.17, 0.68]. It is assumed that a $\overline{\text{Metrics}}$ within the range of [0.17, 0.68] indicates no substantial change in the performance of the training.

[2]Since the baseline timing distributions have a narrow width for both generator and critic, minor speed changes often result in large shifts in the eCDF.

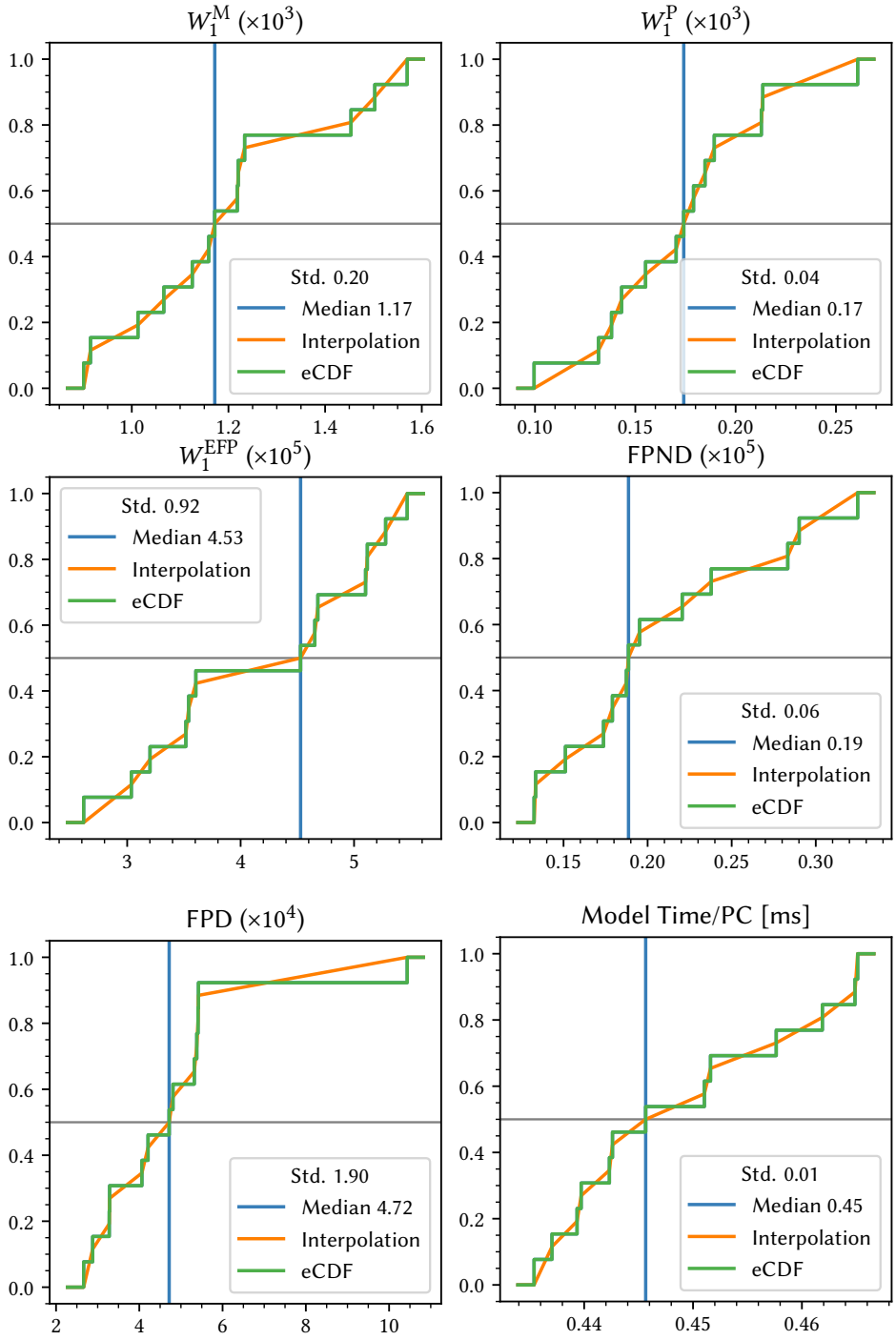


Figure 9.1. | The eCDFs and their Linear Interpolations of the metrics of the Baseline Trainings.|
See Section 9.1 for the discussion.

	W_1^{M}	W_1^{P}	W_1^{EFP}	FPD	FPND
W_1^{M}	1	0.19	0.12	0.26	0.36
W_1^{P}		1	-0.44	0.04	0.22
W_1^{EFP}			1	0.21	0.48
FPD				1	0.31
FPND					1

Table 9.1. | The Pearson Correlation Coefficients for the eCDFs of the Baseline Trainings.| See Section 9.1.2 for the discussion.

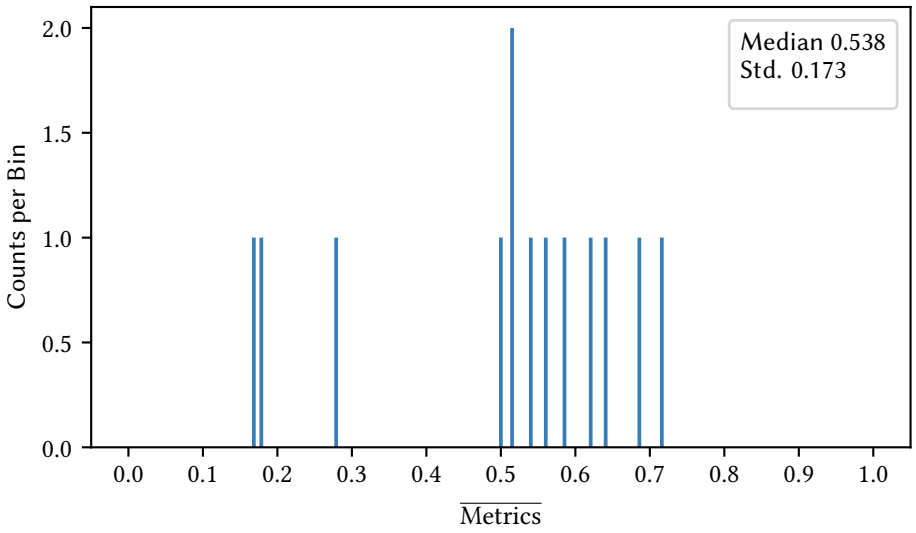


Figure 9.2. | The Histogram of $\overline{\text{Metrics}}$ of the Baseline Trainings.| See Section 9.1.2 for the discussion.

Nomenclature The median of the baseline trainings is referred to as `0-base`. Each training with a modified configuration is assigned a running number `i` and is labeled as `i-gen`, `i-crit`, or `i-common`, depending on whether the modification affects the generator, the critic, or both.

9.2. Simultaneous Modifications of Generator and Critic

In this section, changes affecting both parts of the GAN are investigated.

9.2.1. Loss

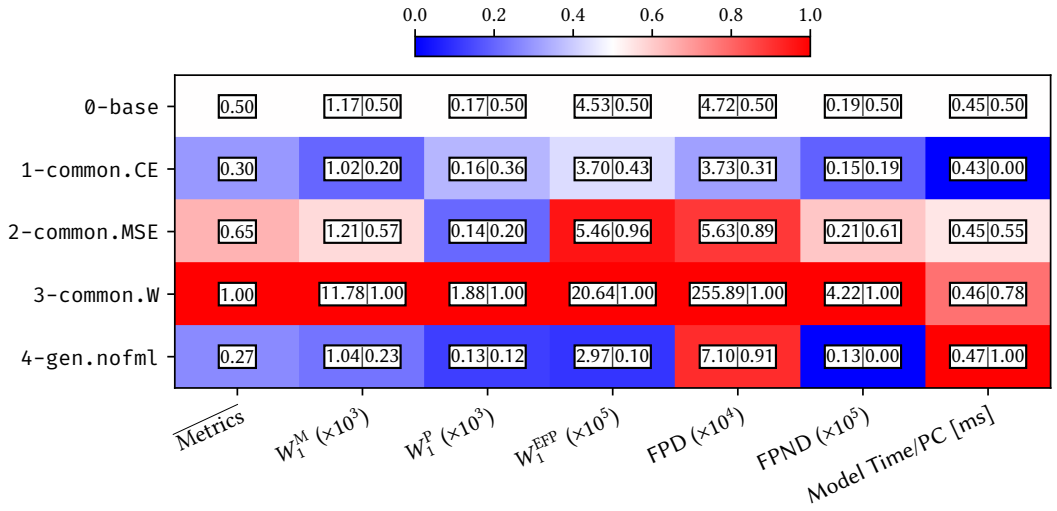


Figure 9.3. | Ablation Results for Modifications to the Loss Terms.|
See Section 9.2.1 for the discussion and Section 9.1.2 for the explanation of the figure. Blue is better, red is worse, and white means unchanged. The cells are labeled with the metric and with the eCDF of the metric (1st and 2nd number).

GAN Objectives The baseline `0-base` was trained using the Hinge Loss (Eq. 3.28), while `1-common.CE`, `2-common.MSE`, and `3-common.W` were trained using non-saturating cross-entropy (Eq. 3.21), least squares (Eq. 3.26), and Wasserstein (Eq. 3.23) loss, respectively. For the `3-common.W` configuration, a gradient penalty loss[3](Eq. 3.25) with a factor of 1 was applied.

The configuration with cross-entropy loss (`1-common.CE`) shows a small advantage in W_1^{M} and FPND. In contrast, the configuration with least squares loss (`2-common.MSE`) shows a significant disadvantage in W_1^{EFP} and FPD. The configuration using Wasserstein loss (`3-common.W`) with gradient penalty effectively fails to converge. The run time shown in the figure reflects the time taken by the components during evaluation and does not account for backpropagation. During training, each gradient step is significantly slower

[3]The gradient penalty is computed by calculating the gradient of the critic with respect to an "interpolated sample" multiple times. These samples are typically interpolated linearly between one sample from the generator and one from the dataset. For a PC-based GAN, constructing an "interpolated PC" raises two issues: First, since PCs are inherently unordered, it is unclear how to match the points from the two PCs to each other. Second, the cardinality of the PCs could differ. In the DeepTree model, the cardinality is taken from the dataset during training, ensuring that the cardinalities match by construction. In this experiment, the points were matched in the arbitrary order produced by the generator. However, it is plausible that ordering both PCs by $p_{\mathrm{T}}^{\mathrm{rel}}$ before interpolation could yield better performance. This complex problem lies outside the scope of this thesis.

than the baseline (approximately 40 ms vs. 57 ms per batch on an NVIDIA V100), likely due to the computation-intensive evaluation of the gradient penalty. While fidelity might improve with a different choice of the loss factor, the training speed cannot.

Feature Matching Loss By default, the generator is trained with the Hinge loss (Eq. 3.28) and a Feature Matching Loss (Eq. 5.1) intended to stabilize the training. However, removing this loss term improves the W_1^{P}, W_1^{EFP}, and FPND metrics (`4-gen.nofml`). Conversely, the FPD significantly worsens. As indicated by the wide plateau for FPD in the range $[6, 10]$ in Fig. 9.1, it is a volatile metric. Therefore, these results still likely suggest a performance improvement.

9.2.2. Other Modifications

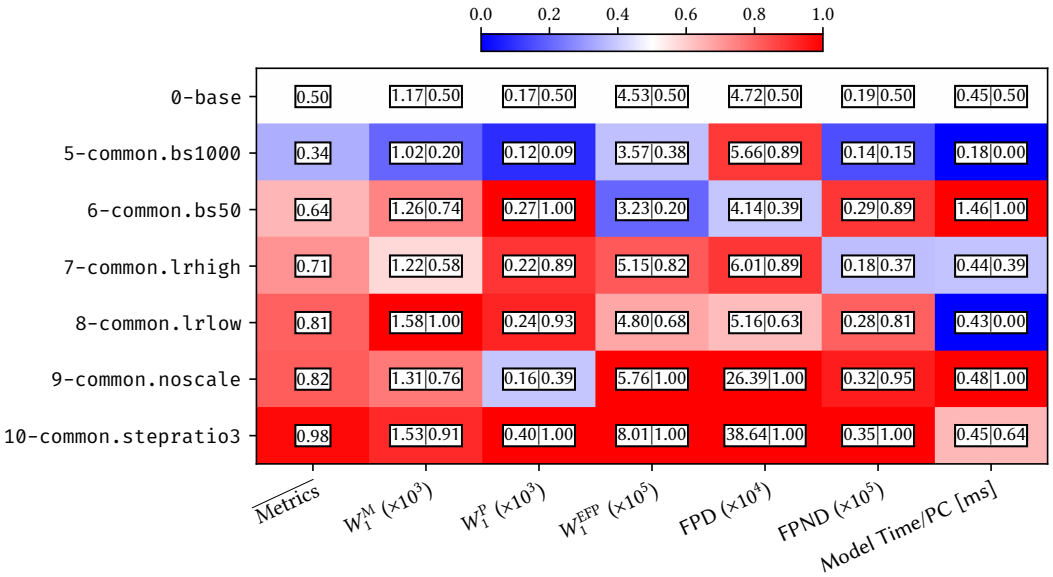


Figure 9.4. | Ablation Results for other Modifications Affecting Both Model Parts.| See Section 9.2.2 for the discussion and Section 9.1.2 for the explanation of the figure. Blue is better, red is worse, and white means unchanged. The cells are labeled with the metric and with the eCDF of the metric (1st and 2nd number).

Batch Sizes To make the configurations with different batch sizes comparable, the number of epochs is adjusted to achieve similar training times using an NVIDIA Tesla V100[4]. The configuration with batch size 1000 / 50 (`5-common.bs1000`/`6-common.bs50`) was trained for a total of 9200 / 900 epochs, corresponding to 255 / 330 hours of training time. As expected, this results in a significant speedup for the batch size 1000 configuration and a slowdown for the batch size 50 configuration. The `5-common.bs1000` configuration shows an improvement, particularly for W_1^{P} and FPND, but performs worse for FPD. FPND and W_1^{EFP} are significantly degraded for `6-common.bs50`.

[4]Due to computational constraints, the trainings take place on V100 GPUs, but the run time of the model is measured on A100 GPUs.

Learning Rate Scaling the learning rate of the generator and critic up (or down) by a factor of 5 in the configuration `7-common.lrhigh` (`8-common.lrlow`) leads to a significant deterioration in performance across three of the five metrics.

Jet Momentum Rescaling For the JetNet datasets, the output of the generator is rescaled such that $\sum_i p_{\mathrm{T,i}}^{\mathrm{rel}}$ equals one, as discussed in Section 6.5. This rescaling occurs during training, with gradients propagated from the critic through the scaling into the generator. While this improves the modeling of momenta, it is uncertain whether it disturbs the gradient. The configuration `9-common.noscale` demonstrates that the model can converge without this rescaling; however, the performance is significantly degraded on W_1^{EFP}, FPD, and FPDN.

Ratio of Critic to Generator Steps In the default configuration, the critic and generator are trained in alternating steps. GANs are often trained with more critic steps per generator step, as seen in Ref. [58]. In the `10-common.stepratio3` configuration, the critic was updated three times for each generator step. The performance of this configuration is significantly worsened across all metrics.

9.3. Modifications to the Feed-Forward Neural Networks

The numerous small FFNs within the DeepTree model are critical to its performance. In this section, modifications to these FFNs are discussed.

Number of Nodes & Number of Layers In the default configuration, the FFNs of the generator use 100 hidden nodes across 3 layers. Reducing the number of hidden nodes to 50 (`11-gen.ffn.hn50`) drastically reduces performance. Doubling the number of hidden nodes (`12-gen.ffn.hn200`) improves W_1^{P} and W_1^{EFP}, but significantly degrades W_1^{M} and FPD. Increasing the number of layers from 3 to 5 (`13-gen.ffn.nl5`) negatively impacts W_1^{P} and FPD for the generator.

Bias By default, the FFNs do not include a bias term. Adding a bias term to the generator (`14-gen.ffn.bias`) leads to an improvement in W_1^{P}, while other changes remain insignificant. For the critic, the addition of a bias term significantly deteriorates W_1^{P} and FPD (`15-crit.ffn.bias`).

Activation The use of GELU [39] and SELU [140] activations instead of LeakyReLU is explored for the generator FFNs in configurations `16-gen.ffn.gelu` and `17-gen.ffn.selu`. Both modifications worsen most metrics, particularly the configuration with SELU activation.

Normalization The choice of normalization is crucial for the convergence of a model. In the generator, Batch Normalization is applied to the FFNs, while the critic utilizes Spectral Normalization.[5] Removing Batch Normalization from the generator leads to a decline in performance, most notably in W_1^{EFP} and FPD (`18-gen.ffn.nonorm`). Adding Spectral Normalization to Batch Normalization (`19-gen.ffn.snbn`) significantly improves W_1^{M} but worsens FPD. For the critic FFNs, removing Spectral Normalization unexpectedly results in a substantial performance improvement (`21-crit.ffn.nonorm`), particularly in W_1^{M}, W_1^{EFP}, and FPD. However, eliminating normalization could negatively affect training stability. Replacing Spectral Normalization with Batch Normalization improves W_1^{M} but significantly degrades W_1^{EFP}, FPD, and FPND (`20-crit.ffn.bn`). Combining both normalizations for the critic improves W_1^{P} but considerably worsens W_1^{EFP}, FPD, and FPND (`22-crit.ffn.snbn`). It should be noted that the normalizations for the branching layers in the generator and the embedding layer in the critic are investigated separately in the respective sections (Section 9.4 and Section 9.5).

Dropout In the default configuration, the FFNs of the generator are trained without dropout, while those of the critic are trained with a dropout rate of 0.5. Introducing a dropout rate of 0.1 or 0.3 to the generator effectively prevents the model from converging (`23-gen.ffn.dr1` and `24-gen.ffn.dr3`). Reducing the dropout rate for the critic FFNs to

[5] Note that the embedding layer in the critic and the branching layer in the generator also use Batch Normalization. The normalization of these components is tuned separately (see Sections 9.4 and 9.5).

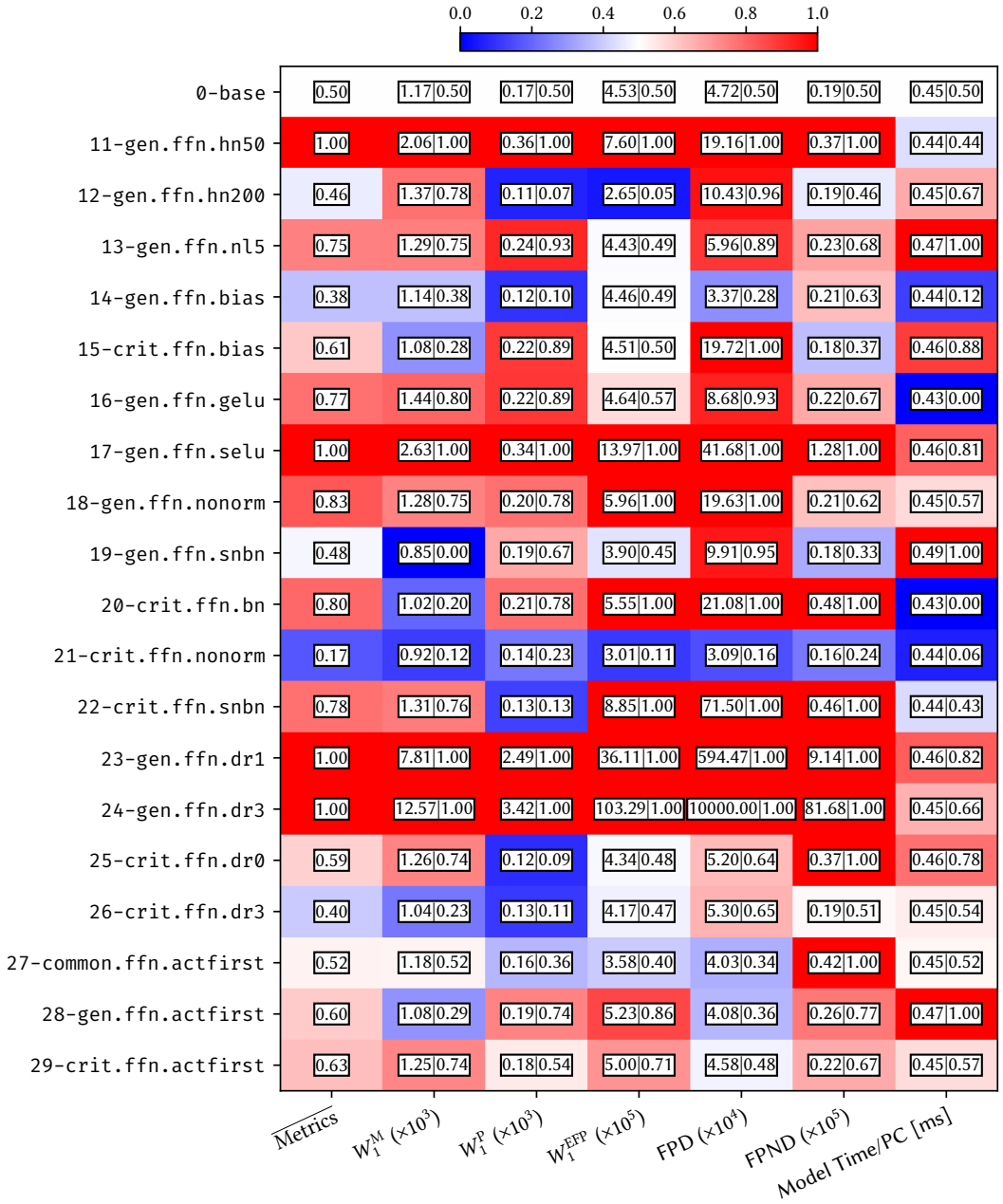


Figure 9.5. | Ablation Results for the FFNs.|
See Section 9.3 for the discussion and Section 9.1.2 for the explanation of the figure. Blue is better, red is worse, and white means unchanged. The cells are labeled with the metric and with the eCDF of the metric (1st and 2nd number).

0.3 or 0.0 has a minor impact on the average eCDF $\overline{\text{Metrics}}$ (`25-crit.ffn.dr0` / `26-crit.ffn.dr3`). For both dropout rates, W_1^{P} improves, but for 0.0, FPND deteriorates.

Ordering of Linear Layer, Normalization, and Activation Configurations `27-common.ffn.actfirst` / `28-gen.ffn.actfirst` / `29-crit.ffn.actfirst` explore an alternative approach to constructing the FFNs. The order of layers within the FFNs is crucial, as discussed in Appendix C.3. Instead of starting with a linear layer followed by Batch Normalization and LeakyReLU activation, the FFNs in these configurations begin with normalization and activation before the linear layer. This reordering, combined with residual connections, introduces significant architectural changes: The residual connections now link the outputs of two linear layers instead of two activations. Additionally, the resulting sum is passed through Batch Normalization and an activation function before reaching a linear layer.

Applying this construction method to both model parts / the generator results in a notable deterioration of FPND / W_1^{EFP} (`27-common.ffn.actfirst` / `28-gen.ffn.actfirst`). For the critic, this modification does not produce any significant change in performance.

9.4. Modifications to the Generator

9.4.1. Tree

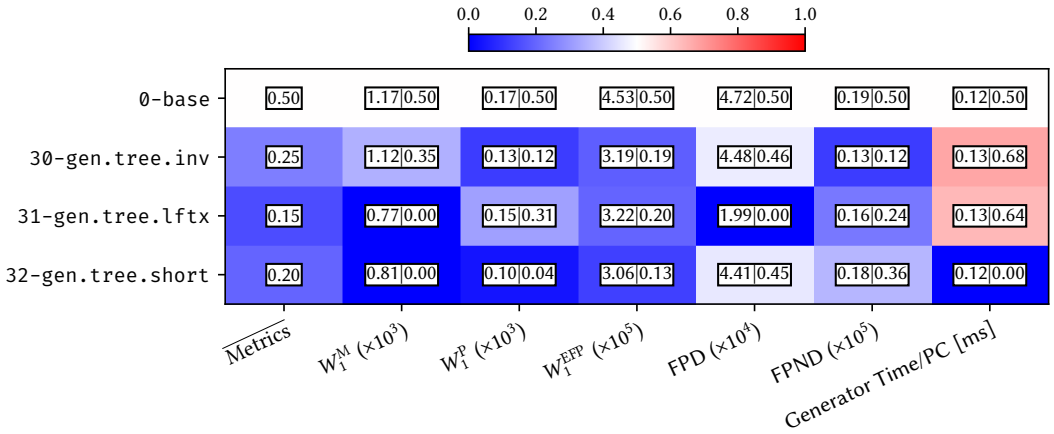


Figure 9.6. | Ablation Results for Modifications to the Trees of the Generator.| See Section 9.4.1 for the discussion and Section 9.1.2 for the explanation of the figure. Blue is better, red is worse, and white means unchanged. The cells are labeled with the metric and with the eCDF of the metric (1st and 2nd number).

In the generator, a single point is mapped through a series of branching layers to a PC with the desired cardinality. The tree structure, including the branching fractions and the size of the points at each level, are considered hyperparameters. In Table 9.2, the tree structure of three modified configurations are compared to the baseline. So far, the prime factors of the desired cardinality have been used as branching fractions in order from lowest to highest ([1,2,3,5,5], similar to the parameters used in Ref. [105]). In the configuration `30-gen.tree.inv`, the order of the branching fractions is inverted to [1,5,5,3,2]. The point dimensions remain unchanged. This results in a performance improvement, especially for W_1^{P}, W_1^{EFP}, and FPND. For the last layer, the authors of Ref. [105] use a greatly increased branching fraction, thus avoiding an additional level of the tree. In the configuration `32-gen.tree.short`, the branching fractions of the last two levels (5 and 5) are combined into one (25). The point dimensions remain unchanged, and the last level is skipped. This configuration shows significantly improved performance, especially for W_1^{M}, W_1^{P}, and W_1^{EFP}. A similar gain in performance, especially for W_1^{M} and FPD, is observed when doubling the point dimensions to [128,64,40,20,3] in the `31-gen.tree.lftx` configuration. Here, the branching fractions remain unchanged. In summary, every single change to the generator tree yields significant improvements. Therefore, the generator tree is a promising area for further optimization.

Configurations ↓	Levels →	0	1	2	3	4
0-base	Branching Fractions	1	2	3	5	5
	Cardinality	1	2	6	30	150
	Point Dimension	64	33	20	10	3
	Tensor Size	64	66	120	300	450
30-gen.tree.inv	Branching Fractions	1	5	5	3	2
	Cardinality	1	5	25	75	150
	Point Dimension	64	33	20	10	3
	Tensor Size	64	165	500	750	450
32-gen.tree.short	Branching Fractions	1	2	3	25	
	Cardinality	1	2	6	150	
	Point Dimension	64	33	20	3	
	Tensor Size	64	66	120	450	
31-gen.tree.lftx	Branching Fractions	1	2	3	5	5
	Cardinality	1	2	6	30	150
	Point Dimension	128	64	40	20	3
	Tensor Size	128	128	240	600	450

Table 9.2. | The Branching Fractions and Point Dimensions for the Configurations Modifying the Tree of the Generator.|
See Section 9.4.1 for the discussion. The "Tensor Size" is the number of floats that represent the PC at the given level.

9.4.2. Branching Layer

The most central components of the generator are the branching layers that map each parent to b new leaves. The default mode of operation is detailed in Section 5.1.1. An FFN is used to map from the feature dimension of the parent f to $b \cdot f$. The resulting vector is then divided into the b new leaves. After this, a second FFN is applied to all nodes in the tree, reducing their dimension.

Equivariant Branching Mechanism In the configuration 33-gen.br.eqv, an alternative branching method (Appendix C.1) is explored. This method requires that the number of features at each level l must be divisible by the branching factor of the subsequent level $l + 1$. In the default configuration, this requirement is already met, with the number of features set to [64,33,20,10,3] and the branching factor to [1,2,3,5,5]. However, this alternative branching method demonstrated significantly worse performance across all metrics compared to the default branching layer.

Noise Branching Mechanism Another alternative branching method (Appendix C.2) was evaluated in the configuration 34-gen.br.noise. In this approach, each parent is concatenated to a noise vector before being mapped to its children. The concatenated noise vector is different for each of its children. However, this method resulted in a significant deterioration in performance across all metrics.

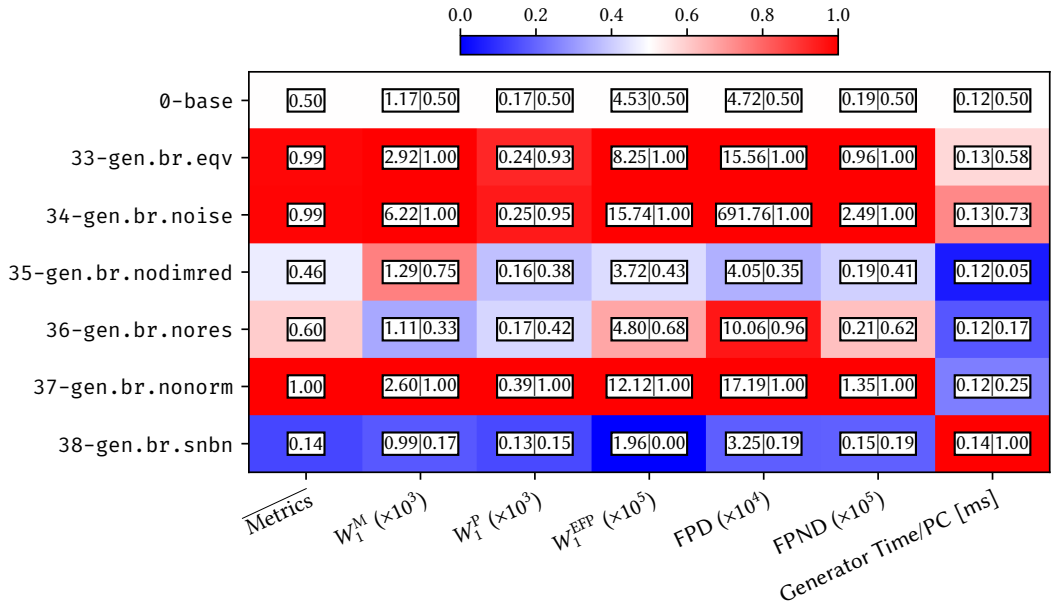


Figure 9.7. | Ablation Results for the Branching Layer of the Generator.|
See Section 9.4.2 for the discussion and Section 9.1.2 for the explanation of the figure. Blue is better, red is worse, and white means unchanged. The cells are labeled with the metric and with the eCDF of the metric (1st and 2nd number).

Dimensionality Reduction in Branching Layer *vs.* in Ancestor MPL In the default configuration, children of the branching FFN are produced with the same size as their parent. The dimension of all nodes is then reduced by the subsequent dimensionality reduction FFN. Alternatively, the FFN within GINConv in the ancestor MPL (Section 5.1.2) may be utilized for reducing node dimensions. In this case, the dimensionality reduction FFN in the branching layer is removed. This results in a dimensional change in the nodes before and after the ancestor MPL. To implement the residual connection for the MPL, the output of the MPL is added to the input, truncated to match the same dimension. The resulting configuration `35-gen.br.nodimred` produces no significant change in performance.

Residual Connections The branching layers contain a residual connection, where each parent is added to its newly produced child (Fig. 5.2). When this residual connection is removed, a degradation in the FPD metric is observed (`36-gen.br.nores`).

Normalization Batch Normalization is employed to regularize the branching layer, including both the branching FFN and the dimensionality reduction FFN. In the trainings `37-gen.br.nonorm` and `38-gen.br.snbn`, Batch Normalization is either removed or Spectral Normalization is added. The removal of Batch Normalization results in a drastic deterioration across all metrics, whereas the addition of Spectral Normalization leads to a clear improvement in all metrics.

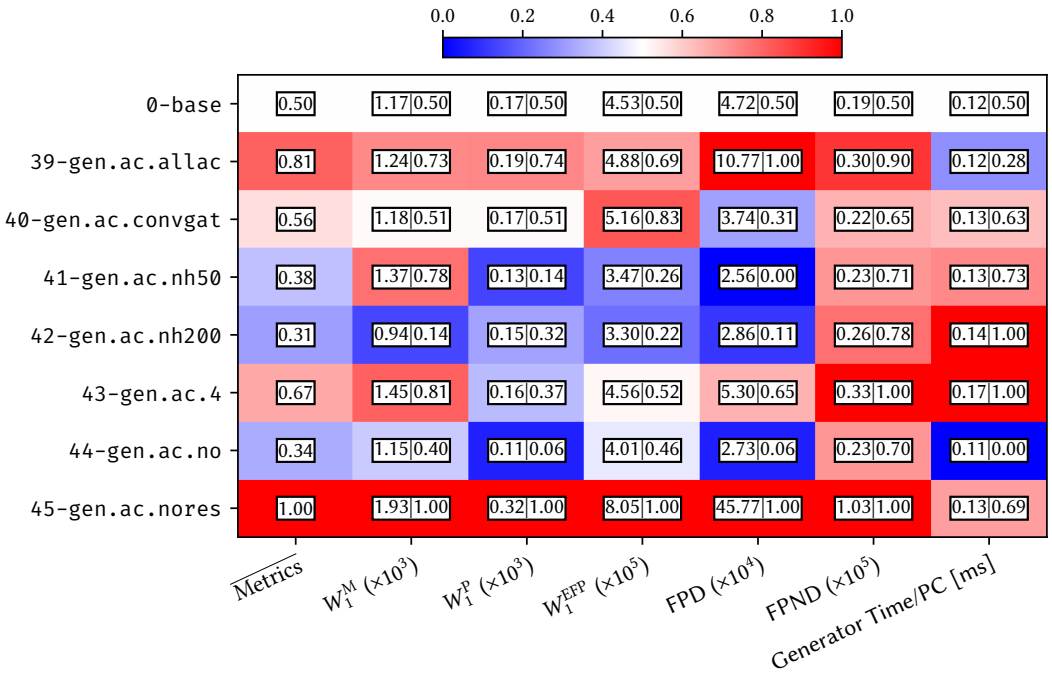


Figure 9.8. | Ablation Results for the Ancestor MPL of the Generator.|
See Section 9.4.3 for the discussion and Section 9.1.2 for the explanation of the figure. Blue is better, red is worse, and white means unchanged. The cells are labeled with the metric and with the eCDF of the metric (1st and 2nd number).

9.4.3. Ancestor MPL

As described in Section 5.2, the ancestor MPL passes messages from each node to all its descendants.

Parent *vs.* Ancestor Connections If messages are passed by parents only to their direct children, rather than to all descendants, a slight decline in performance is observed, particularly for FPD and FPND metrics (`39-gen.ac.allac`).

Choice of MPL GINConv Section 3.6.4 is utilized as the MPL. Substituting it with GATv2Conv (Eq. 3.47), which is employed in the critic, results in a worse W_1^{EFP}, with no other significant changes in performance (`40-gen.ac.convgat`).

Size of FFN inside the MPL By default, the FFN within GINConv in the ancestor MPL utilizes 100 hidden nodes. Doubling the number of hidden nodes results in a significant improvement for W_1^{M} and FPD (`42-gen.ac.nh200`). Conversely, halving the number of hidden nodes also leads to an improvement in FPD and W_1^{P} (`41-gen.ac.nh50`). However, the results of these changes appear contradictory, rendering the findings inconclusive.

Residual Connection over MPL A residual connection has been added to skip over the MPL, aimed at preventing a vanishing gradient (Section 5.1.2). The removal of this residual connection severely degrades performance across all metrics (`45-gen.ac.nores`).

Multiple MPLs When the ancestor MPL is replaced by a sequence of four MPLs instead of a single one, W_1^{M} and FPND significantly worsen (`43-gen.ac.4`).

Removing the MPL In the `44-gen.ac.no` configuration, the ancestor MPLs are removed from the generator. Surprisingly, this results in a significant performance boost in W_1^{P} and FPD, along with a reduction in run time. Eliminating this component would substantially reduce the model's complexity and could drastically improve run time, particularly when scaling to larger PCs.

9.4.4. Gating the Condition

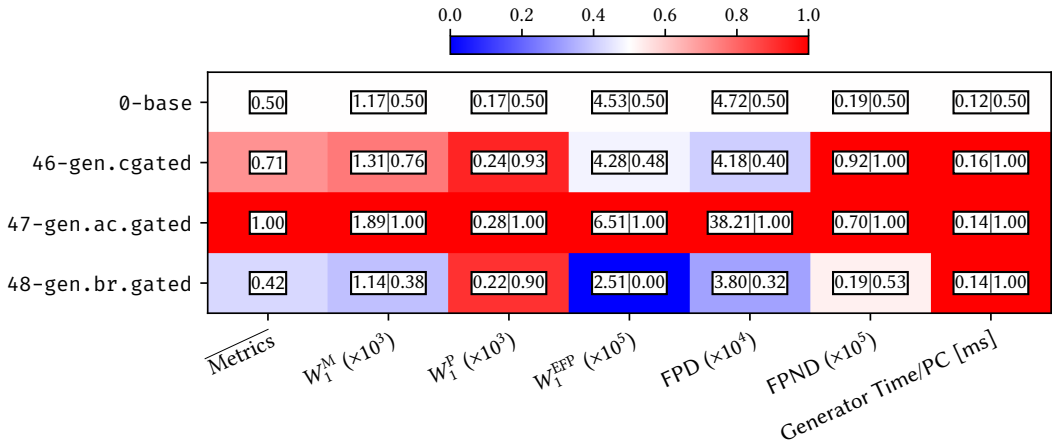


Figure 9.9. | Ablation Results for the Gating in the Generator.|
See Section 9.4.4 for the discussion and Section 9.1.2 for the explanation of the figure. Blue is better, red is worse, and white means unchanged. The cells are labeled with the metric and with the eCDF of the metric (1st and 2nd number).

The branching layer and ancestor MPL both receive the condition and the global feature as additional inputs. In the branching layer, these inputs are concatenated with the parent and passed through the branching FFN to generate the children. In the ancestor MPL, they are concatenated with each node before being passed to the MPL. Instead of concatenation, the "Gated Conditioning Unit," described in Appendix D.3, is used to combine the input vector with the global feature and the condition. This modification could enhance the network's responsiveness to global features and the condition while reducing the input size of subsequent components, such as the branching FFN. With the configuration `48-gen.br.gated`, this gating is applied to the branching layers; with `47-gen.ac.gated`, it is applied to the ancestor MPLs; and with `46-gen.cgated`, it is applied to both. For the branching layers (`48-gen.br.gated`), the impact on performance is mixed: W_1^{P} improves, but W_1^{M} deteriorates. When the gating is applied to the ancestor MPLs alone (`47-gen.ac.gated`), performance declines across all metrics. Similarly, a significant performance deterioration is observed when gating is applied to both types of layers (`46-gen.cgated`), particularly affecting W_1^{P} and FPND.

9.4.5. Other Modifications

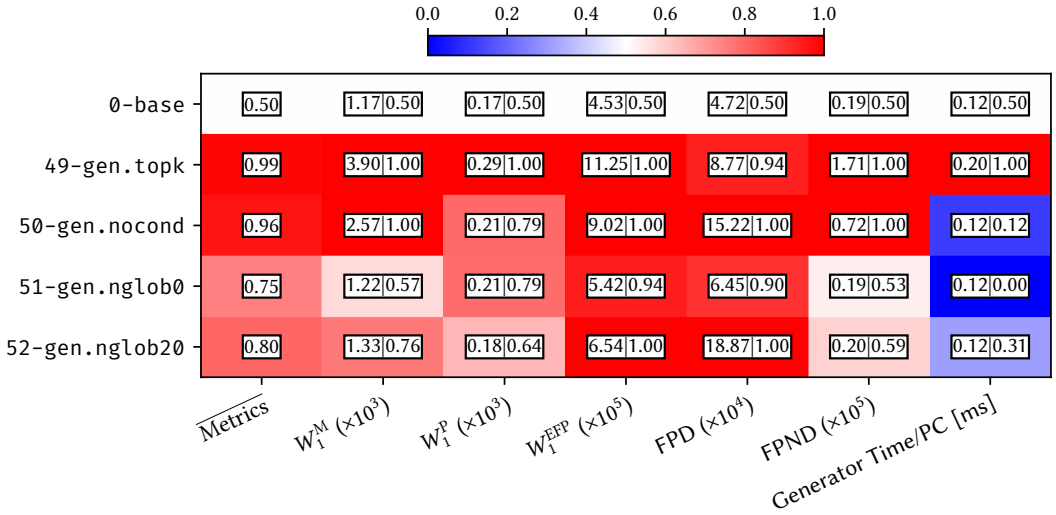


Figure 9.10. | Ablation Results for other Modifications to the Generator.|
See Section 9.4.5 for the discussion and Section 9.1.2 for the explanation of the figure. Blue is better, red is worse, and white means unchanged. The cells are labeled with the metric and with the eCDF of the metric (1st and 2nd number).

Truncating the Output PC Since this generator produces a fixed-size PC, points must be removed before returning the PC. The target cardinality C is sampled from the dataset. The first C points in the output vector of the generator are used, as described in Section 5.1. In the configuration `49-gen.topk`, the C points with the largest p_T^{rel} values are selected instead. This results in not only a drastic performance degradation across all metrics but also a significant increase in the generator's run time.

Conditioning For the generator, the condition consists solely of the target cardinality. Removing this condition significantly degrades performance across all metrics (`50-gen.nocond`). While the generator's reliance on this condition is a minor limitation, many established methods exist for reproducing a one-dimensional distribution.

Size of the Global Feature The global feature layer aggregates the leaves and constructs the global feature, which is used to condition the subsequent branching layer and ancestor MPL. Both doubling the size to 20 (`52-gen.nglob20`) and removing the global feature entirely (`51-gen.nglob0`) result in performance degradation, particularly for W_1^{EFP} and FPD.

9.5. Modifications to the Critic

9.5.1. Tree Structure

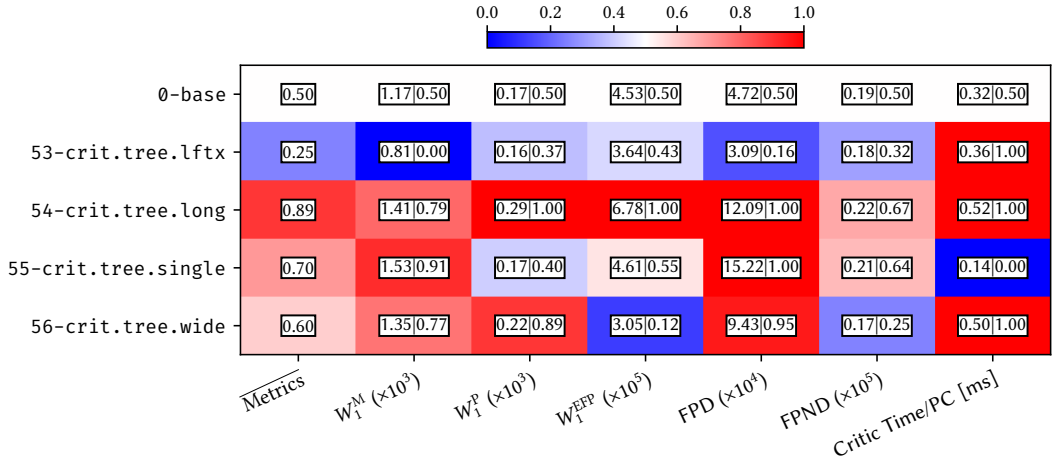


Figure 9.11. | Ablation Results for Modifications to the Tree of the Critic.|
See Section 9.5.1 for the discussion and Section 9.1.2 for the explanation of the figure. Blue is better, red is worse, and white means unchanged. The cells are labeled with the metric and with the eCDF of the metric (1st and 2nd number).

Similar to the generator, the critic possesses its own tree-like[6] structure in which the points are aggregated. The bipartite pool maps a PC of any cardinality to a fixed number of nodes (Section 5.2.1). Thus, instead of fixing the branching factor as in the generator, the cardinality at each level beyond the input is fixed. By default, these cardinalities are set to $[C, 30, 6, 1]$, where C is the input cardinality. In Table 9.3, three configurations with modified tree structures are compared to the baseline.

The critic is composed of multiple subcritics, operating both before and after the pooling operations. The first subcritic operates on the full-size PC and can focus on low-level features, while the subsequent subcritics, which operate on increasingly aggregated PCs, can concentrate on higher-level features. However, the necessity of these additional subcritics is uncertain. The model `55-crit.tree.single` includes only a single pool, which aggregates the PC into one point (cardinalities $[C, 1]$ with point dimensions $[3, 20]$). As a result, it contains only two subcritics. As expected, this leads to significantly poorer performance in FPD and W_1^{EFP}.

Conversely, adding two additional levels, along with two more subcritics, also proved detrimental to performance. In the configuration `54-crit.tree.long`, cardinalities of $[C, 50, 30, 15, 6, 1]$ and point dimensions of $[3, 10, 10, 10, 10]$ were used, causing a significant degradation in W_1^{P}, W_1^{EFP}, and FPD.

Similarly, shifting to higher cardinalities $[C, 75, 25, 5, 1]$ (with point dimensions $[3, 10, 10, 10, 10]$) improved W_1^{EFP} but worsened W_1^{P} and FPD (`56-crit.tree.wide`).

However, shifting to increasing point dimensions $[3, 20, 40, 80]$ showed a substantial advantage for W_1^{M} and FPD (`53-crit.tree.lftx`).

[6] Note that the structure used in the critic is not strictly a tree, as the Bipartite Pool connects each input node with all seed nodes, rather than a single parent.

Configurations ↓	Levels →	0	1	2	3	4	5
`0-base`	Cardinality	C	30	6	1		
	Point Dimension	3	10	10	10		
	Tensor Size	$c \cdot 3$	300	60	10		
`55-crit.tree.single`	Cardinality	C	1				
	Point Dimension	3	20				
	Tensor Size	$c \cdot 3$	300				
`54-crit.tree.long`	Cardinality	C	50	30	15	6	1
	Point Dimension	3	10	10	10	10	10
	Tensor Size	$c \cdot 3$	500	300	150	60	10
`56-crit.tree.wide`	Cardinality	C	75	25	5	1	
	Point Dimension	3	10	10	10	10	
	Tensor Size	$c \cdot 3$	300	60	10	10	
`53-crit.tree.lftx`	Cardinality	C	30	6	1		
	Point Dimension	3	20	40	80		
	Tensor Size	$c \cdot 3$	60	240	80		

Table 9.3. | The Cardinalities and the Point Dimensions for the Configurations that Change the Tree of the Critic.|
See Section 9.5.1 for the discussion. The input cardinality is C. The "Tensor Size" is the number of floats that represent the PC at the given level.

9.5.2. Bipartite Pool

For the MPL layer within the bipartite pool, GATmConv, a variant of GATv2Conv, is employed. These MPLs are described in Section 3.6.5. Reverting to GATv2Conv results in a significant deterioration in W_1^{P} and W_1^{EFP} (`62-crit.bbp.gatplus`). In a prior study, a configuration using GINConv instead (Section 3.6.4) failed to converge.

To ensure a Lipschitz continuous critic, all components must satisfy the Lipschitz condition. By default, Spectral Normalization is not applied to the $\boldsymbol{\theta}_s/\boldsymbol{\theta}_t$ matrices. Applying it produces inconclusive results, as some metrics deteriorate (W_1^{M}, FPD) while others improve (W_1^{P}) (`63-crit.bbp.sn`).

Increasing the number of attention heads from 16 to 32 degrades performance in W_1^{EFP} and FPD (`64-crit.bbp.heads32`). Conversely, reducing the number of attention heads to 4 worsens W_1^{P} and FPND, but improves W_1^{M} and W_1^{EFP} (`65-crit.bbp.heads4`). Additionally, it reduces the runtime of the critic from 32 to 28 ms per PC.

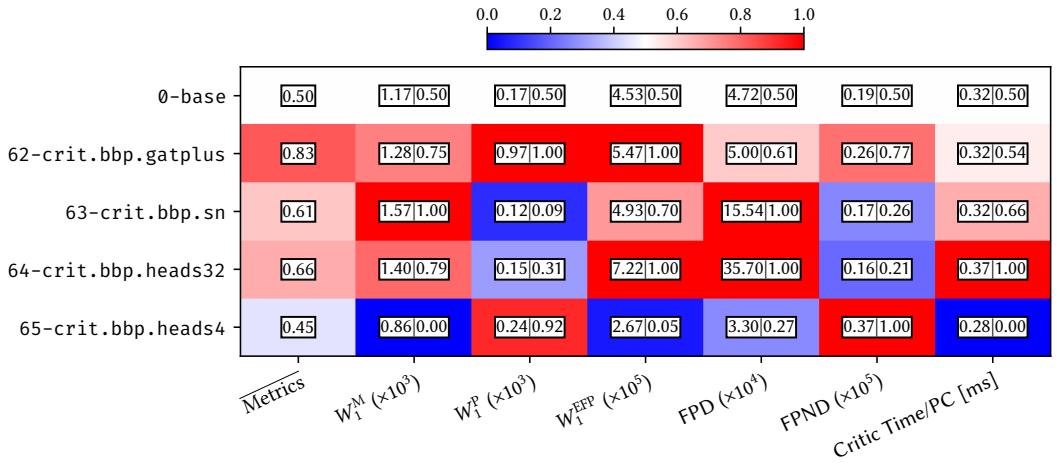


Figure 9.12. | Ablation Results for Modifications to the Bipartite Pool in the Critic.| See Section 9.5.2 for the discussion and Section 9.1.2 for the explanation of the figure. Blue is better, red is worse, and white means unchanged. The cells are labeled with the metric and with the eCDF of the metric (1st and 2nd number).

9.5.3. Embedding Layer

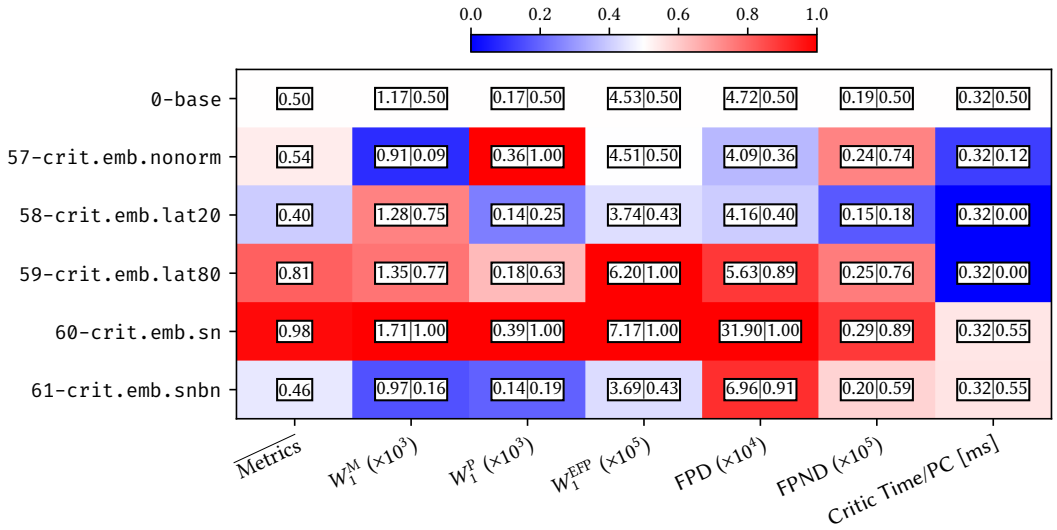


Figure 9.13. | Ablation Results for Modifications to the Embedding Layer in the Critic.| See Section 9.5.3 for the discussion and Section 9.1.2 for the explanation of the figure. Blue is better, red is worse, and white means unchanged. The cells are labeled with the metric and with the eCDF of the metric (1st and 2nd number).

Normalization Previous studies have indicated that the normalization of embedding layers is crucial for the model's performance. As a result of these studies, the embedding layers in the critic are the only components that utilize Batch Normalization, while Spectral Normalization is applied to the remaining parts. However, removing Batch Normalization yields inconclusive results, with W_1^{M} improving but W_1^{P} deteriorating (`57-crit.emb.nonorm`). In the configuration `60-crit.emb.sn`, where Batch Normalization is replaced by Spectral Normalization, a significant degradation in performance is observed across all

metrics. When both normalizations are applied, the results remain inconclusive: W_1^{M} and W_1^{P} improve, but FPD deteriorates (`61-crit.emb.snbn`).

Central Node Dimension The effectiveness of an embedding layer depends on its CNUs (Section 5.2). Halving the Central Node Dimension n_g to 20 improves FPND, but the overall impact on performance is inconclusive (`58-crit.emb.lat20`). Doubling n_g results in degraded performance, particularly for W_1^{EFP} and FPD (`59-crit.emb.lat80`).

9.5.4. Subcritics

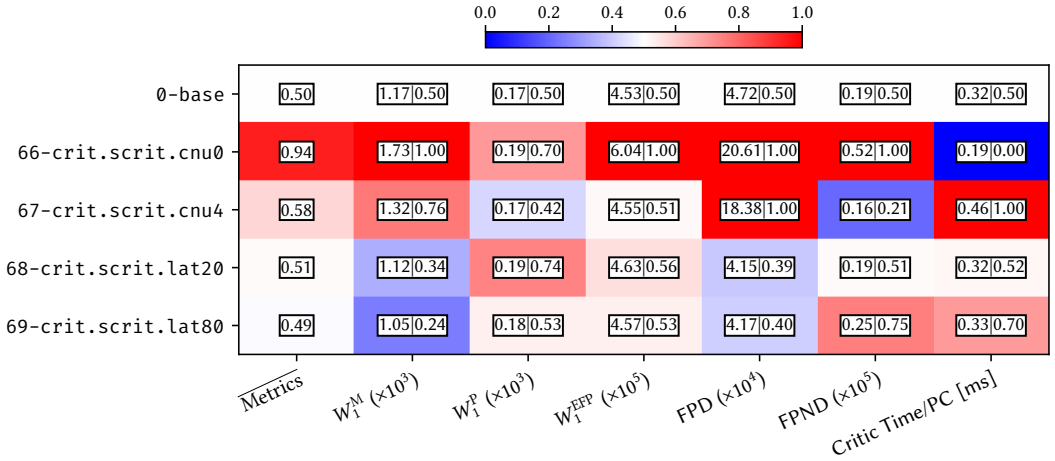


Figure 9.14. | Ablation Results for Modifications to the Subcritics.|
See Section 9.5.4 for the discussion and Section 9.1.2 for the explanation of the figure. Blue is better, red is worse, and white means unchanged. The cells are labeled with the metric and with the eCDF of the metric (1st and 2nd number).

The subcritics consist of two CNUs with a residual connection, multi-aggregation, and an FFN (Section 5.2). Removing the CNUs from the subcritics significantly worsens performance across all metrics except W_1^{P} (`66-crit.scrit.cnu0`). Conversely, doubling the number of CNUs to 4 results in a deterioration in FPD (`67-crit.scrit.cnu4`). Halving or doubling the central node dimension in the CNUs produces no significant change in any metric (`68-crit.scrit.lat20` / `69-crit.scrit.lat80`).

9.5.5. Removing the Condition

The critic is conditioned on $\tilde{\phi}$, $\tilde{\eta}$, and $\tilde{m}$. Removing this conditioning results in a drastic improvement in W_1^{P} and FPD, but also leads to a drastic deterioration in W_1^{M} and W_1^{EFP} (`70-crit.nocond`).

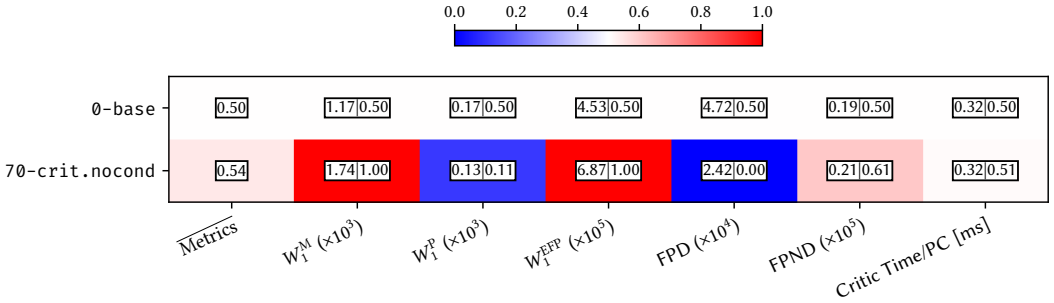


Figure 9.15. | Ablation Results for Removing the Condition from the Critic.|
See Section 9.5.5 for the discussion and Section 9.1.2 for the explanation of the figure. Blue is better, red is worse, and white means unchanged. The cells are labeled with the metric and with the eCDF of the metric (1st and 2nd number).

9.6. Optimizing the Design

As expected, the performance of most configurations in these ablation studies is either worse than or comparable to the baseline configuration. Only 3 out of 70 configurations have an $\overline{\text{Metrics}}$ lower than or equal to the best baseline training (0.17, see Fig. 9.2). Conversely, approximately 68% (48 out of 70) of the modified configurations produce metrics with eCDFs greater than 1, meaning they fall outside the distribution of the baseline trainings.

This suggests that the model is overall well-tuned.
Table 9.4 shows the best-performing configurations that yield at least three metrics with eCDFs ≤ 0.2.

Table 9.4. | The Configurations of the Ablation Study that Yield at Least Three Metrics with eCDFs ≤ 0.2.|
See Section 9.6 for the discussion.

Configuration	$\overline{\text{Metrics}} \downarrow$	No. of Metrics with eCDF ≤ 0.2
`38-gen.br.snbn`	0.14	5/5
`31-gen.tree.lftx`	0.15	3/5
`21-crit.ffn.nonorm`	0.17	5/5
`32-gen.tree.short`	0.20	3/5
`4-gen.nofml`	0.27	3/5
`44-gen.ac.no`	0.34	3/5
`5-common.bs1000`	0.34	3/5
`30-gen.tree.inv`	0.35	3/5

To determine whether these changes genuinely improve performance or are simply due to statistical fluctuations, these configurations would need to be trained repeatedly, and the resulting metric distributions need to be compared to the baseline. Nevertheless, the study provides a set of observations that could be used to improve the model in the future:

- Spectral Normalization, initially intended for the critic, might instead benefit the branching layers of the generator.
- The tree structure shows potential for improvement, particularly in the generator, where all introduced changes lead to significant improvement in three of the five metrics.
- The Feature Matching Loss may deteriorate the performance.
- The ancestor MPLs in the generator, which account for the majority of the generator's complexity, may be unnecessary or even harmful.

This clearly shows that the best possible performance with this model has not yet been achieved, and further studies are likely to yield additional improvements. Beyond improving the metrics, it is also desirable to achieve a more stable training process, such as

through regularization, and to simplify the architecture. Thus, adding Spectral Normalization to the branching layer (`38-gen.br.snbn`) or removing the ancestor MPLs (`44-gen.ac.no`) are particularly promising modifications. While some of these changes can be implemented independently, others may interact or counteract each other. To determine the optimal combination of changes, a more complex and detailed study is required, which is beyond the scope of this thesis.

Table 9.5. | Comparison of the DeepTree– with the `31-gen.tree.lftx` Configuration – to EPiC-GAN and MDMA.|
See Section 9.6 for the discussion. The default and `31-gen.tree.lftx` configurations are described in Table 5.1 and Section 9.4.1. See Sections 4.6.2 and 4.6.3 for the description of EPiC-GAN and MDMA. The metrics and their computation are presented in Section 6.3. Lower values indicate better performance for all metrics. The uncertainty is reported as the standard deviation of the metrics calculated on the bootstrapped samples. The best central value of a model for a given metric is highlighted in bold. The "Limit" row represents the in-sample distance measured by sampling datasets from the training dataset and calculating the metrics. Values for EPiC-GAN and MDMA taken from Ref. [102, Table 1].

Model	$W_1^{\mathrm{M}}(\times 10^3)$	$W_1^{\mathrm{P}}(\times 10^3)$	$W_1^{\mathrm{EFP}}(\times 10^5)$	$\mathrm{FPD}(\times 10^4)$
Limit	0.42 ± 0.09	0.12 ± 0.04	1.22 ± 0.32	1.2 ± 0.6
EPiC-GAN	0.69 ± 0.08	0.65 ± 0.03	2.67 ± 0.39	22 ± 1
MDMA	$\mathbf{0.57 \pm 0.09}$	$\mathbf{0.10 \pm 0.02}$	$\mathbf{2.12 \pm 0.64}$	5.3 ± 0.9
DeepTree (default)	1.49 ± 0.04	0.13 ± 0.02	5.01 ± 0.08	3.4 ± 0.7
DeepTree (`31-gen.tree.lftx`)	0.77 ± 0.06	0.17 ± 0.02	3.46 ± 0.10	$\mathbf{2.5 \pm 0.4}$

Best Configuration Performance

Table 9.5 presents the metrics computed for the `31-gen.tree.lftx` configuration. With the updated hyperparameters, DeepTree performs almost on par with EPiC-GAN and MDMA across all metrics, and even outperforms them in FPD. The distribution of the jet variables is shown in Fig. 9.16. Compared to the previous distributions in Fig. 8.2, the $\tilde{m}$ and $\tilde{\eta}$ distributions show better agreement with the dataset distribution.

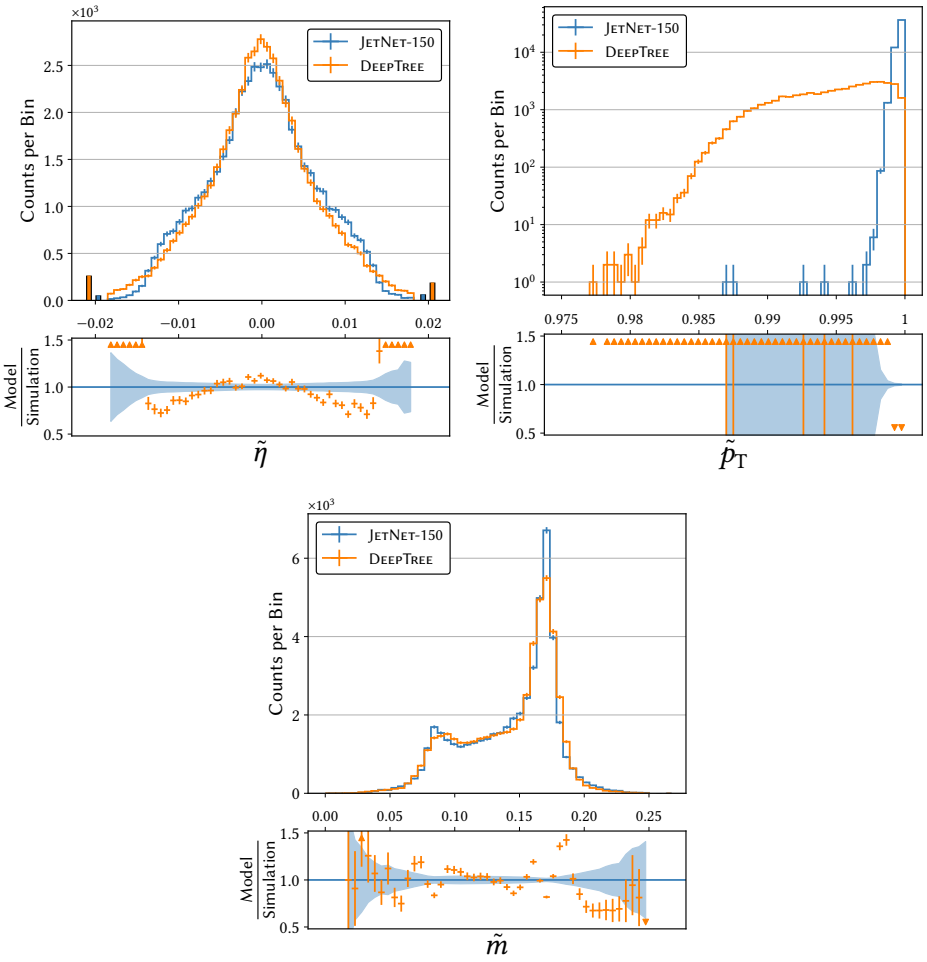


Figure 9.16. | Distributions of the Jet Variables – with the `31-gen.tree.lftx` Configuration – for the t jets in the JetNet-150 Test Dataset.|
Update to Fig. 8.2. The `31-gen.tree.lftx` configuration is described in Section 9.4.1. The plot design is explained in Appendix A.1.

Chapter Review In this chapter, a method was developed to systematically quantify changes to a model with fluctuating performance. This method was then applied to review the architecture and hyperparameter choices for the DeepTree model. While the study confirms the choices at large, it also opens up multiple avenues for further improving the model. Thus, Milestone 5 has been achieved.

CHAPTER 10

CaloChallenge

Chapter Abstract This chapter describes the entry to the CaloChallenge competition using the DeepTree model. The properties of the dataset necessitate changes to the model and the processing steps (Milestones 6 & 7). Generating particle showers requires producing PCs with very high cardinalities. While the studies in the previous chapters focused on achieving high fidelity at mid-range cardinalities, this study demonstrates the model's capability to scale up to even higher cardinalities (Milestone 8).

10.1. Introduction

The "Fast Calorimeter Simulation Challenge 2022" (CaloChallenge) [141] is a public[1] competition aimed at developing generative models for calorimeter simulation. Each of the three provided datasets contains particle showers initiated by a single particle, simulated using GEANT4. The datasets are simulated with an increasing number of calorimeter cells and therefore increasing difficulty:

Dataset 1 [142] is based on the ATLAS GEANT4 open datasets [143]. The dataset is divided into two parts: one containing particle showers initiated by photons, and the other by charged pions. Its irregular geometry comprises 368 cells for photons and 533 cells for pions.

Dataset 2 [144] features a detector with cylindrical geometry. The detector consists of 45 layers along the cylinder axis, made from active (silicon) and passive (tungsten) material. These layers are divided into 9 radial and 16 angular segments, resulting in 144 cells per layer, for a total of 6480 cells. The 100k showers are initiated by electrons, with energies sampled from a log-uniform distribution in the range [1 GeV, 1 TeV]. An additional 100k electron showers are provided for testing.

Dataset 3 [145] differs from Dataset 2 only in the segmentation of the detector. It consists of 45 layers, each divided into 18 radial and 50 angular segments, yielding a total of 40500 cells.

[1] calochallenge.github.io/homepage/

Potentially, each cell can contain an energy deposition. This allows the shower to be represented as a PC, with each energy deposition corresponding to a point. The number of cells determines the maximum cardinality of the PC. Thus, the maximum cardinality of more than 40k for Dataset 3 places an extremely high memory demand on the GPU, resulting in very slow processing times. In contrast, Dataset 1 contains only 368/533 cells, making it less demanding and enabling faster turnaround times.

Entry to the CaloChallenge Competition

Dataset 2 was selected for this study because it contains a realistic shower simulation and has a reduced processing time compared to Dataset 3. The model developed in this study was submitted to the competition and presented at the ML4Jets2023 conference [146], together with the results of other models [147]. A final evaluation is in preparation [141].

Simulated Detector

Figure 10.1 shows the coordinate system of the simulated detector for Dataset 2. The shower initiating particles enter the detector along the axis of the cylinder (z-axis). Energy deposits within the cells of the cylinder are described by the z index (in $\{0 \dots 44\}$), the angular index α (in $\{0 \dots 15\}$), and the radial index r (in $\{0 \dots 8\}$).

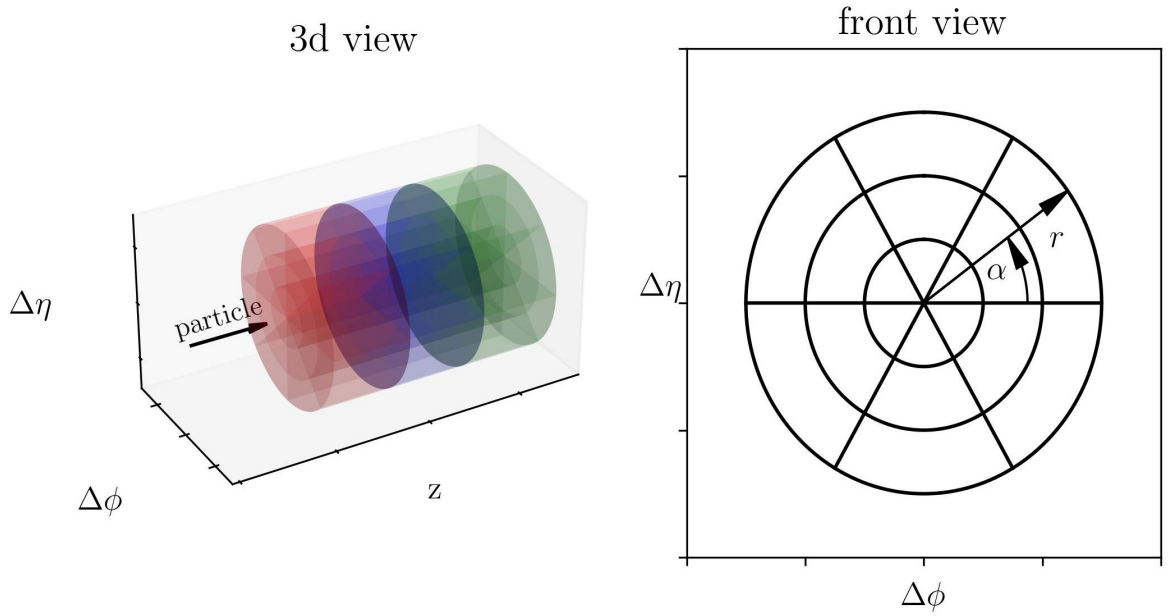


Figure 10.1. | Coordinate System for the Detector in CaloChallenge Datasets 2 and 3 [141].|
See Chapter 10 for the discussion.

Since the detector is invariant to rotations in α and the initial particle enters along the z axis, the entire dataset is symmetric in α. As a result, the α indices are evenly distributed (Fig. 10.3). Most hits occur close to the beam axis. The indexing of the radial segmentations begins with the innermost layer. Many energy depositions occur close to the beam axis, resulting in a falling r distribution (Fig. 10.2). Showers are initiated by a single particle, which produces an increasing number of particles with decreasing energies through decays and interactions, until the particles' energies fall below the threshold for producing new

particles and are absorbed. This process creates a longitudinal (z) hit distribution that peaks rapidly and then gradually decreases (Fig. 10.4).

Point Cloud-based Models for the CaloChallenge

A particle shower can be effectively represented as a PC if the following assumptions (Chapter 4) are satisfied:

- The detector comprises a high number of densely distributed small cells, creating an almost continuous space.
- The geometry is irregular, preventing the shower from being represented on a grid.
- The sparsity, defined as the fraction of cells without a hit, is high.

However, these assumptions are violated in all datasets: Dataset 1 contains a very low number of cells. Although the simulated detectors in Datasets 2 and 3 contain significantly more cells, the sparsity remains low, as discussed later in this section.

The low sparsity and regular geometry in these datasets suggest that a grid-based model, such as a CNN [79] or a VisionTransformer [90], might be more appropriate. Moreover, these properties negate the advantages of a PC-based model and introduce several challenges:

- Representing the shower as a PC assumes a continuous space with a continuous density of points. When the detector consists of large cells, the density of hits may vary rapidly at the boundaries, which is difficult to replicate with a PC-based model.
- A grid representation places adjacent cells next to each other, allowing neighborhood information to be utilized, for example, by a convolutional layer. This neighborhood information is not inherently included in the PC-based representation and cannot be directly evaluated. In a high-sparsity setting, hits in neighboring cells would be rare, limiting the usefulness of this information.
- Each point in the PC representation contains the position and energy of the hit, resulting in a dimension of 3+1. By contrast, a grid representation uses an ordered tensor, where the cell positions are defined by their placement within the tensor (Chapter 4). As a result, the dimension of a single cell is 1 (just the energy). When the fraction of cells with hits, multiplied by the point dimension (4), exceeds 1, the tensor size used to represent the PC becomes larger than that of the tensor in a grid representation.
- As a postprocessing step, points from this continuous space are mapped to discrete cells. If the points are not sparsely deposited, multiple points could be mapped to the same cell.

However, if a PC-based model can produce reasonable results under these suboptimal conditions, this further proves its capabilities.

10.2. Model

The cardinalities required for the CaloChallenge Dataset 2 put DeepTree's scalability to the test. While cardinalities could reach 6480, i.e., depositing energy in every single cell of the calorimeter, the highest cardinality observed in the dataset is 5496. As a compromise between the potential and the observed maximum cardinality, the branching factors were configured to produce a final cardinality of 6000.

Hyperparameters The changes to the model are presented in Table 10.1. In addition to the modifications necessary to produce PCs with the required cardinalities, several changes were introduced that improved results in a preliminary study. The ablation study presented in the previous chapter occurred after this study. Therefore, the suggested improvements could not be included here.

Conditioning Variables The goal of the CaloChallenge was to provide a model capable of generating a shower with a given energy. Thus, the initial energy E_{in}, the cardinality C, and the average energy per hit $\bar{E}$ were used as conditioning variables not only for the critic, but also for the generator.

Avoiding Multiple Hits per Cell By default, the model has no incentive to maintain a distance between two points. However, placing points too close together may result in both hits being mapped to the same cell. In addition to the postprocessing strategy described in the following section, a loss term is introduced to penalize the model when too many points are too close to each other. This '`nndist`' loss term compares the distances of each point to its nearest neighbor and penalizes the model if its generated distributions deviate from the dataset distribution. Further details are discussed in Appendix D.2.

Table 10.1. | Hyperparameters Different from the Default for Training DeepTree on the CaloChallenge Dataset.|
See Section 10.2 for the discussion. For the default hyperparameters, see Table 5.1.

Common		Batch size	50
		Condition	$[E_{\text{in}}, C, \bar{E}]$
	FFNs	Order	[Activation, Linear, Dropout, Normalization]
Generator		Loss	Hinge + $10 \cdot$ `nndist`
		Node Features by level	[64,25,15,10,8,6,4]
		Branching Factor by level	[1,2,3,4,5,5,10] ($\prod b_i = 6000$)
	Ancestor MPL	Gating (Appendix D.3)	True
	Branching Layer	Gating (Appendix D.3)	True
Critic	FFNs	Dropout	0.3
	Embedding Layer	Normalization	Batch + Spectral Normalization
	Bipartite Pool	Normalization	Spectral Normalization

10.3. Pre- & Postprocessing

As outlined in Section 4.3, the pre- and postprocessing steps required to train and evaluate a PC-based model on calorimeter data with discrete cells are quite extensive. The preprocessing of the training dataset involves the following steps, which are described in detail below:

1. Hits are converted into points containing the energy and the r, α, and z indices of the respective cells.

2. The discrete indices are transformed to continuous distributions ("dequantized").

3. The marginal distributions of the points (hit energy and coordinates) are mapped to distributions approximating a standard normal distribution.

Conversely, the postprocessing of the generated showers involves the following steps:

1. The transformations of the marginal distributions are inverted.

2. An algorithm converts the points into energies assigned to cells.

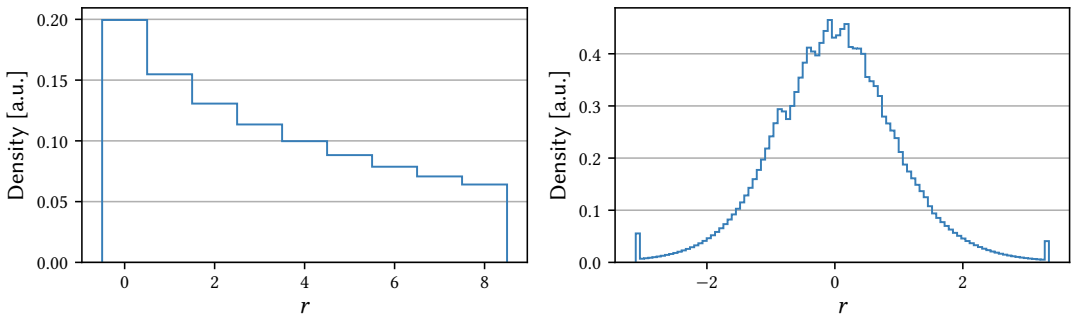
3. Since the showers are invariant under rotation in α, the generated PCs are rotated by a random shift to ensure a flat α distribution.

Choice of the Transformations

The set of transformations described here was chosen to map the coordinates of the points to distributions approximating a standard normal distribution. In this context, approximating a standard normal distribution means that the distribution is unimodal, symmetric, and has a standard deviation close to 1. The output distribution of a newly initialized FFN also approximates a standard normal distribution (Fig. C.4), being unimodal, symmetric, and having a standard deviation roughly around 1, depending on the input and weight initialization. As a result, the coordinate distributions produced by the model immediately after initialization are similar to the target coordinate distributions. This approach simplifies the task for the model.

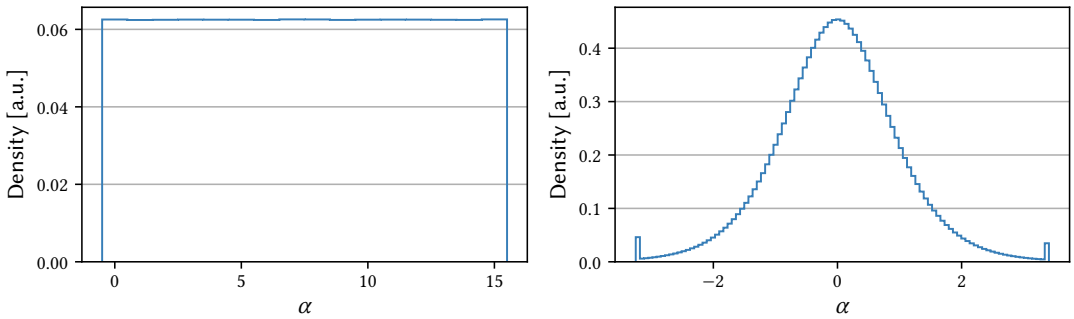
10.3.1. Mapping between the Discrete Cells and Continuous Points

PCs representing particle showers consist of points located at the positions of the cells. Training a generative model directly on this "quantized" PC would require the model to first learn to reproduce the discrete geometry within the continuous space of the PC. To simplify this task, the discrete distributions should be dequantized, transforming them into continuous ones. The exact location of the energy deposition within a cell is inherently unknown. Therefore, each position within the cell is equally likely, and the positions of the hits should be transformed into a uniform distribution spanning the cell. This dataset has a regular grid structure, where the cells can be addressed by r, α, and z indices. Thus, instead of operating on the positions, we operate on the indices.



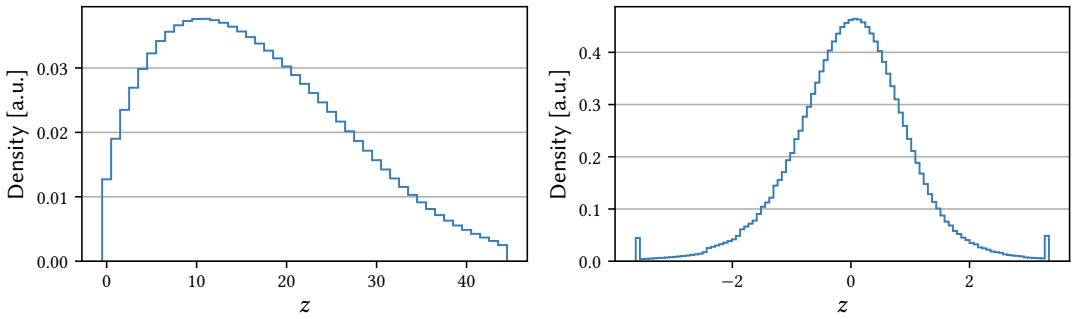
(a) Without the Transformation. **(b)** With the Transformation.

Figure 10.2. | The **r** Distribution of the Simulation Dataset.|
See Section 10.3.1 for the discussion.



(a) Without the Transformation. **(b)** With the Transformation.

Figure 10.3. | The $\boldsymbol{\alpha}$ Distribution of the Simulation Dataset.|
See Section 10.3.1 for the discussion.



(a) Without the Transformation. **(b)** With the Transformation.

Figure 10.4. | The **z** Distribution of the Simulation Dataset.|
See Section 10.3.1 for the discussion.

Cell Index Transformation

The r, α, and z indices are transformed through the following sequence of steps:

1. Uniform noise $\mathcal{U}([0,1))$ is added to these indices to make the distribution continuous. This can be reversed by applying the floor operation.
2. The continuous values are scaled to have a minimum of 0 and a maximum of 1.
3. The distribution is then transformed using the "logit" function (reverse: "expit"), defined as
$$\operatorname{logit} x = \log \frac{x}{1-x}, \quad \text{and} \quad \operatorname{expit} x = \frac{x}{1+\exp x}.$$
4. Finally, the distribution is scaled to have a mean of 0 and a standard deviation of 1, this is referred to as "normal scaling".

If each index contains the same number of points, adding uniform noise and scaling to [0,1] results in a standard uniform distribution. This distribution can then be converted to one resembling the standard normal distribution by applying the "logit"[2] function and normal scaling. The effects of these transformations on the coordinates are shown in Fig. 10.4 (z), Fig. 10.3 (α), and Fig. 10.2 (r). Even for the r and z indices, which are not as uniformly distributed as α, the transformed distributions approximate a standard normal distribution.

Inverting the Transformations

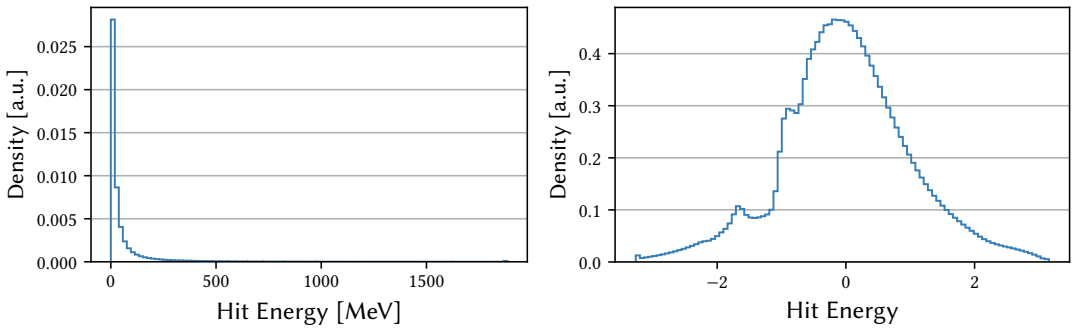
During generation, the applied transformations are inverted, and the floor operation is used to discretize the coordinates into indices. Since the model was trained on this continuous space, it may produce hits close to each other that are mapped to the same cell. The simplest approach to address this issue is to sum the hits assigned to the same cells. However, this method results in fewer hits at higher energies. Under the assumption of high sparsity, this effect would be rare. But given the very low sparsity in this dataset, a specialized algorithm is required to manage these cases. A preceding study demonstrated that the introduced `nndist` loss (Appendix D.2), which has been added to the model, can mitigate this effect by spacing out the points, though it does not fully resolve the issue. In Section 10.3.4, a custom algorithm is presented that shifts low-energy hits to neighboring cells.

10.3.2. Hit Energy Transformation

The hit energies typically follow an approximate power law distribution, $|\mathbf{E}|^{-\varepsilon}$, where $\varepsilon > 0$. Using the Box-Cox transformation, these distributions can be transformed into Gaussian-like distributions[3]. The hit energies are scaled using a Box-Cox transformation [133] and then normal scaled. For evaluation and generation, these transformations are inverted. Applying the Box-Cox transformation and normal scaling to the hit energies E results in a normal-like distribution, although the density is not completely smooth, as shown in Fig. 10.5.

[2] Applying the inverse of the CDF of the standard normal distribution would directly return the standard normal distribution. However, this approach showed no advantage and proved to be numerically unstable.

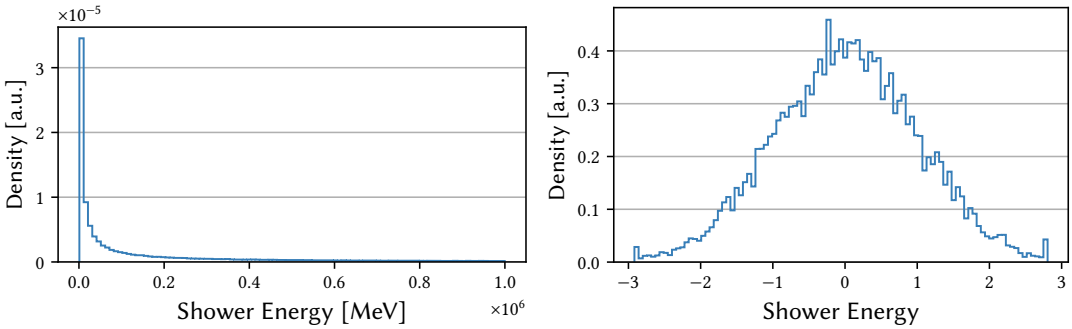
[3] See scikit-learn.org/stable/modules/preprocessing.html, accessed 10.07.2024.



(a) Without the Transformation.

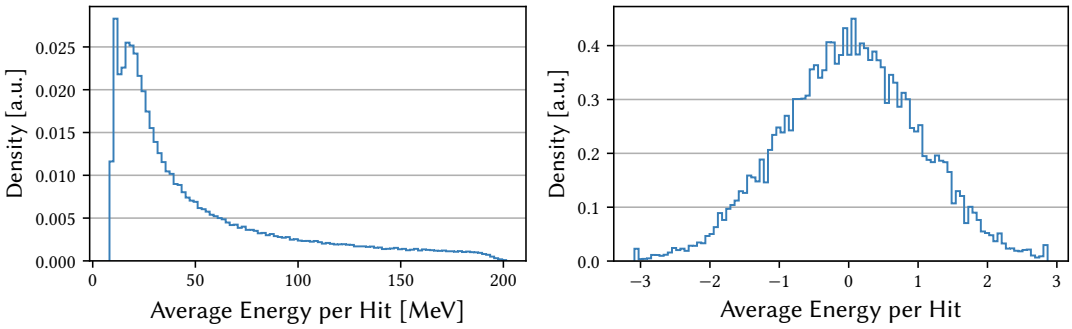
(b) With the Transformation.

Figure 10.5. | The Hit Energy (in MeV) Distribution of the Simulation Dataset.| See Section 10.3.2 for the discussion.



(a) Without the Transformation. **(b)** With the Transformation.

Figure 10.6. | The Shower Energy $\mathbf{E}_{\text{in}}$ Distribution of the Simulation Dataset.| See Section 10.3.3 for the discussion.



(a) Without the Transformation. **(b)** With the Transformation.

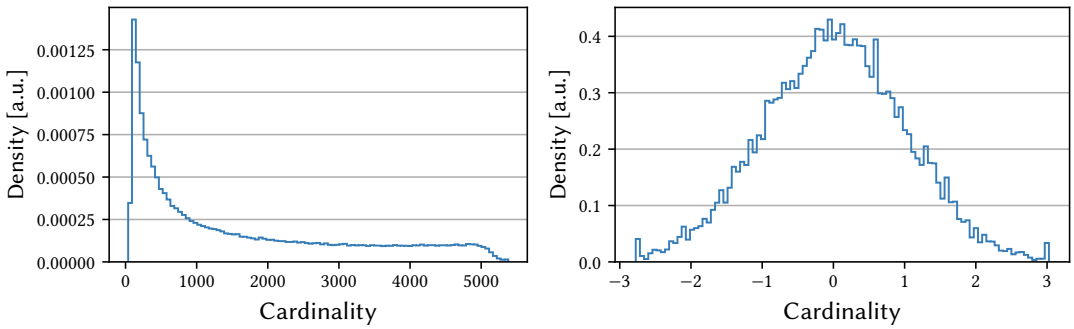
Figure 10.7. | The Average Energy per Hit $\bar{\mathbf{E}}$ Distribution of the Simulation Dataset.| See Section 10.3.3 for the discussion.

10.3.3. Transformation of the Conditioning Variables

Any distribution can be transformed into a standard normal distribution by first applying the inverse CDF, followed by the CDF of the standard normal distribution. If the inverse CDF is unknown, it can be approximated using a sample. This transformation is implemented by SKLEARN's `QuantileTransformer` [133]. While the `QuantileTransformer` produces distributions that closely resemble the desired normal distribution, it can also introduce undesired artifacts (Appendix B) if applied to an independent sample. If the `QuantileTransformer` were applied to the points, its inverse transformation would need to be applied to the model output. To avoid this, the `QuantileTransformer` is applied only to the conditioning variables, leaving the generator output unaffected.

The initial energy E_{in} and the average energy $\bar{E}$ are both transformed using the Box-Cox transformation followed by `QuantileTransformer`. Figures 10.6 and 10.7 show the resulting normal-like distributions.

In Fig. 10.8, the distributions for the cardinality C before and after the transformation are shown. The cardinality distribution peaks quickly at approximately 115 points and has a very long tail extending beyond 5000 points. Due to this long tail, roughly 19% of the showers occupy more than half of the 6480 calorimeter cells. However, instead of normal scaling, `QuantileTransformer` is applied as the final step. The resulting distribution closely approximates a standard normal distribution (Fig. 10.8).



(a) Without the Transformation.

(b) With the Transformation.

Figure 10.8. | The Cardinality **c** Distribution of the Simulation Dataset.|
See Section 10.3.3 for the discussion.

10.3.4. Hit-Shifting

Method

As previously discussed, the generator is not directly aware of the calorimeter cells. It may produce multiple hits for a single calorimeter cell (*multi-hits*, see Section 2.3), especially in showers with very high cardinality. Simply summing the multi-hits for each cell would result in a low cardinality and hits with very high energies. To mitigate this effect, an algorithm is employed to move the multi-hits from "overcrowded" cells to neighboring empty cells (in $r/\alpha/z$). Because higher energy hits are more important, the algorithm moves the multi-hits in order of energy, skipping the highest energy hit. If no adjacent empty cells are available, the remaining multi-hits are summed up.

The underlying assumption is that moving a hit from one cell to the next does not significantly alter any distribution. This assumption will be tested in the following section (Section 10.4.1).

Limitations and Edge Cases

The algorithm is expected to perform less effectively at the borders of a calorimeter, where finding an empty neighboring cell is more challenging. Additionally, if i is the maximum index for a given coordinate, multi-hits can be shifted from i to $i-1$ but not from $i+1$ to i. This introduces a negative bias for cells in the outer layer and a positive bias for the cells adjacent to them. By shifting multi-hits between neighboring bins, the algorithm may smear out distributions and is unsuitable for distributions with extreme density changes from one cell to the next.

Run Time

The algorithm iterates through the neighboring cells of all cells with multiple hits until all multi-hits have been moved to a neighboring empty cell. Thus, its run time scales with both the number of multi-hits and the probability of finding an empty cell next to a cell with a multi-hit. This makes the algorithm fast in a high-sparsity setting, for which the DeepTree model has been designed, but slow in a high-occupancy setting.

However, in the study presented in the following section, these undesired effects are found to be negligible. Furthermore, it is demonstrated that this algorithm can significantly improve the fidelity of a PC-based model in a low-sparsity setting. This algorithm, along with other tools for handling calorimeter data as PCs, has been packaged and released on the Python Package Index [148].

10.4. Evaluation

Assessing the fidelity of a generative model for this dataset is more challenging than for the JetNet datasets. Since no set of sensitive metrics has been established yet, there is no baseline for comparison. Therefore, in addition to evaluating the marginal distributions of the hits, 2D distributions and a set of shower variables are also used to evaluate the performance.

Furthermore, the effects of the hit-shifting will be evaluated in the following sections. By design, the DeepTree model produces the same cardinality as in the dataset, but the aggregation of hits causes the cardinality to deviate. To investigate the effects of aggregation and hit-shifting, the dataset is compared to the model output with hit-shifting and aggregation (labeled "DeepTree (Shift+Sum)"), with summation only (labeled "DeepTree (Sum)"), and without aggregation (labeled "DeepTree (No Aggr.)"). The last option cannot be used to map the PCs to calorimeter cells, and is provided only for comparison.

10.4.1. Cardinality and Hit Variables

Cardinality

The cardinality in a shower is shown in Fig. 10.9. The dataset distribution peaks near zero, then rapidly falls to a plateau that extends beyond 5000 hits. The no-aggregation distribution matches the dataset distribution by design. In the aggregation-only distribution, the near-zero peak shifts closer to zero, whereas the distribution with hit-shifting accurately reproduces this peak.

The dataset distribution reaches a maximum cardinality of 5623 hits. The tail of the aggregation-only distribution extends only to 3546 hits, leading to increased oversampling in the [1000, 3500] hits range and undersampling beyond that. With hit-shifting, the tail extends to 4688 hits, and oversampling becomes noticeable only above 3000 hits. As cardinality increases, the likelihood of hits occupying the same cells also increases, making the effect of hit-shifting more pronounced. Overall, hit-shifting significantly improves the agreement between the generated and dataset distributions.

Hit Energy

Figure 10.10 shows the energy distribution of the hits. Summing hits in the same cells leads to fewer hits but with higher energies. The dataset distribution contains many hits at low energies. The lower end of the distribution (<400 MeV) matches well with the no-aggregation distribution, while the aggregation-only distribution shows fewer hits at these energies due to aggregation. Hit-shifting significantly improves the agreement, though noticeable discrepancies between the generated and dataset distributions persist, particularly in the high-energy tails:

For higher energies, the no-aggregation distribution initially contains too few hits between 800 MeV and 2200 MeV and too many in the tail of the distribution. The aggregation-only distribution contains fewer hits overall but overpopulates the tail of the distribution starting at 1900 MeV. Additionally, it includes hits with energies outside the range of the dataset distribution. Hit-shifting significantly increases the number of generated low-energy hits, resulting in a distribution that lies midway between the no-aggregation

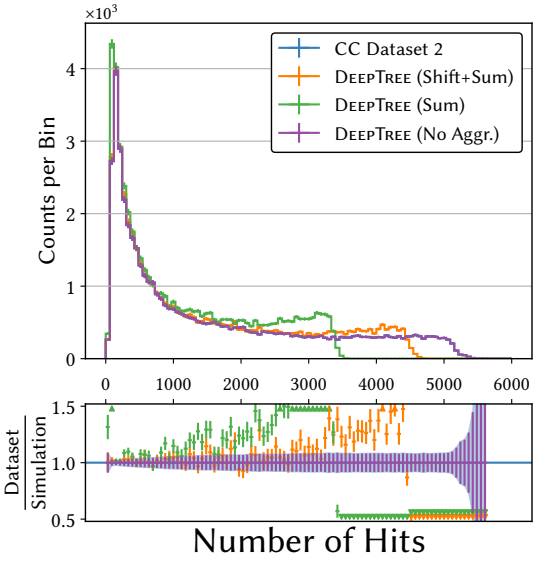


Figure 10.9. | Distributions of the Cardinality per Shower with and without Shifting the Hits.|
See Section 10.4.1 for the discussion. The no-aggregation distribution ('No Aggr.') matches the dataset distribution ('CC Dataset 2') by construction. The plot design is explained in Appendix A.1, with the difference that this histogram contains 100 bins.

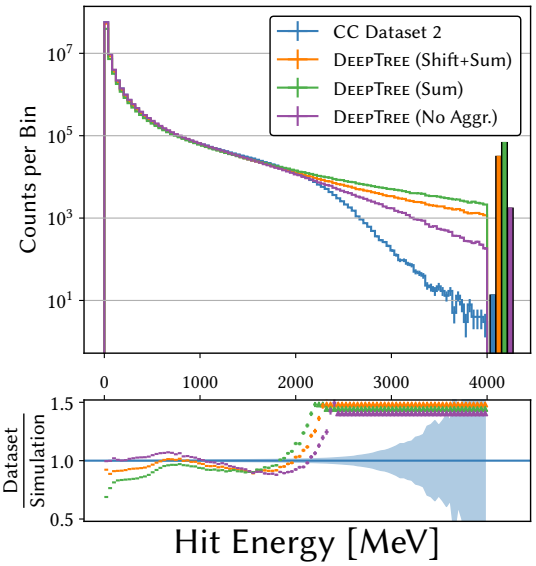


Figure 10.10. | Distributions of the Hit Energies with and without Shifting the Hits.|
See Section 10.4.1 for the discussion. The plot design is explained in Appendix A.1, with the difference that this histogram contains 100 bins.

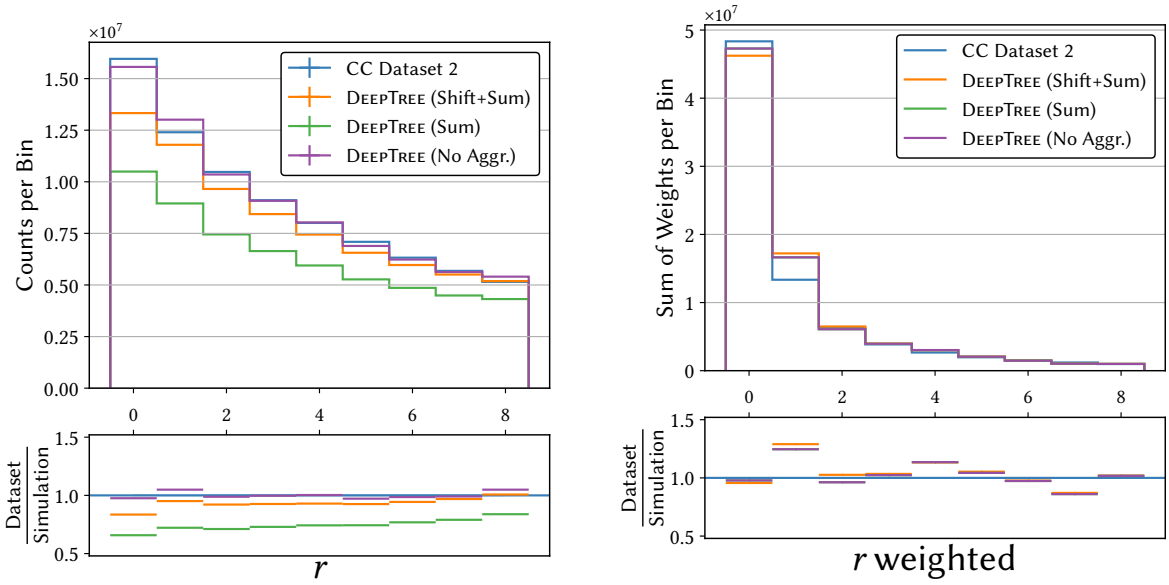


Figure 10.11. | Distribution of r for the Hits with and without Shifting the Hits.| See Section 10.4.1 for the discussion. The histogram on the right is weighted with the hit energy to an average weight per hit of 1. The plot design is explained in Appendix A.1.

and aggregation-only distributions. The remaining discrepancy could be reduced with a transformation that scales the tail of the distribution differently.

Hit Coordinates

The distributions of the coordinates are shown in Figs. 10.11 to 10.13. To fairly assess the fidelity of the model, the random shift in α (Section 10.3) was not applied for these histograms. The observations are similar across all coordinates and can be summarized as follows:

For the *unweighted* histograms, the no-aggregation distribution generally aligns well with the dataset distribution. The aggregation-only distribution produces too few hits overall. Its ratio to the dataset distribution decreases as the number of hits in the dataset distribution increases within the respective bin. Hit-shifting significantly reduces this mismatch, but does not entirely restore the fidelity seen in the no-aggregation distribution. Overall, hit-shifting moves the generated distributions closer to those of the dataset.

In the *energy-weighted* histograms, the differences among the three generated distributions are negligible. By design, the distribution is the same for the no-aggregation and aggregation-only distributions, but may differ for the hit-shifting distribution. Overall, the generated distributions align well with the dataset distribution, as the ratio is mostly in the range $[0.9, 1.1]$. An exception is the r distribution, where the $i = 1$ bin is overpopulated, and hit-shifting slightly worsens the modeling. This distribution may be challenging to model, as most of the energy is concentrated close to the beam axis in the $r = 0$ bin.

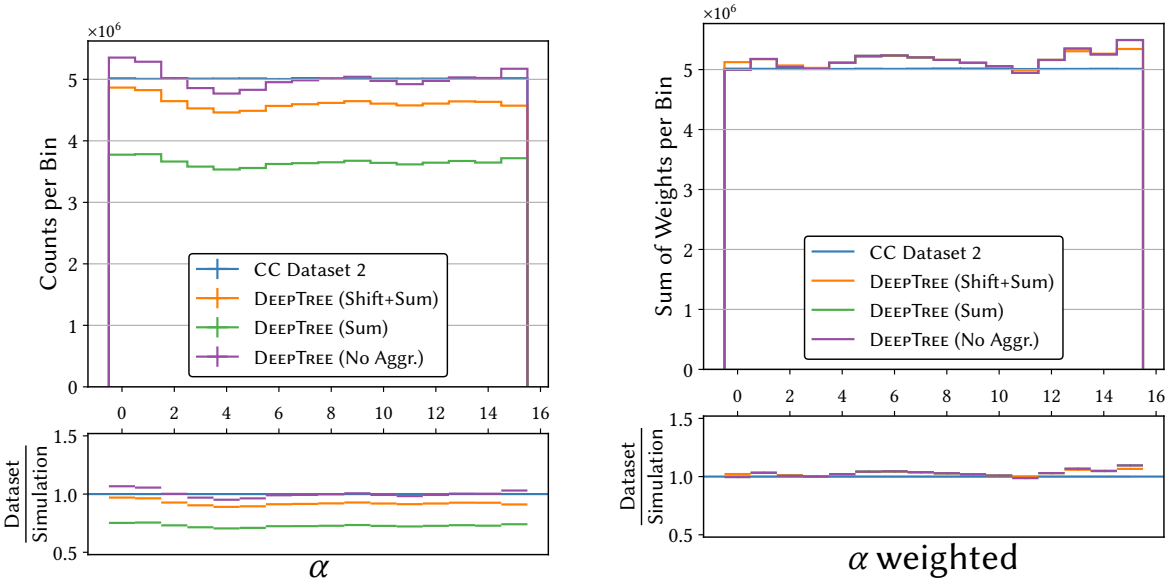


Figure 10.12. | Distribution of α for the Hits with and without Shifting the Hits.| See Section 10.4.1 for the discussion. The histogram on the right is weighted with the hit energy to an average weight per hit of 1. The plot design is explained in Appendix A.1.

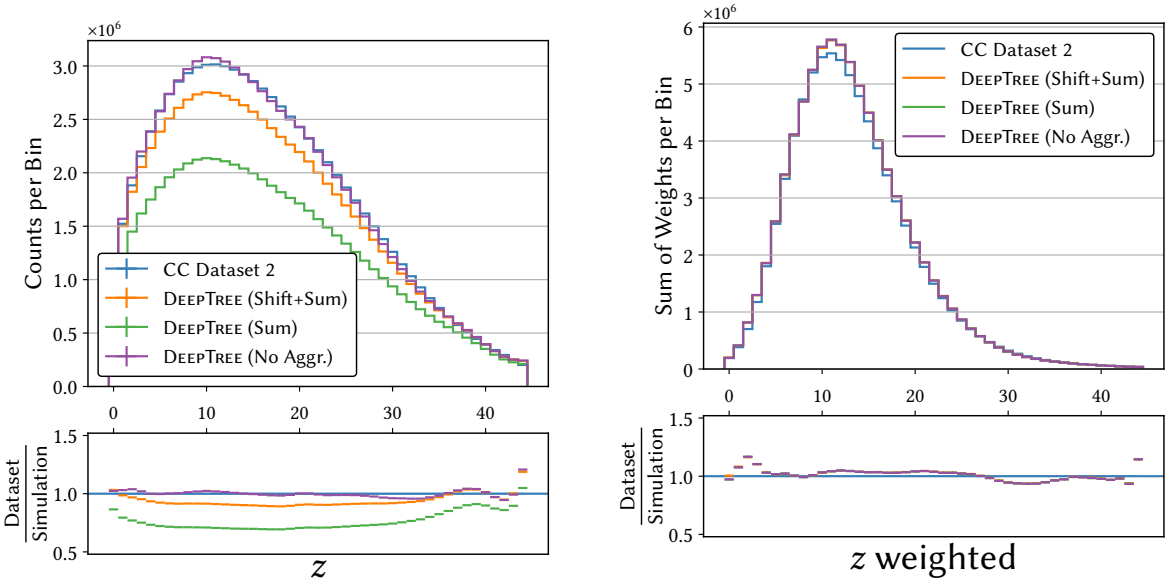


Figure 10.13. | Distribution of z for the Hits with and without Shifting the Hits.| See Section 10.4.1 for the discussion. The histogram on the right is weighted with the hit energy to an average weight per hit of 1. The plot design is explained in Appendix A.1.

10.4.2. 2D Distributions of the Hit Variables

To evaluate the quality of the produced 2D distributions, both hit-shifting (previously labeled 'Shift+Sum') and α rotation are applied again as described in Section 10.3. Since the distributions are flat in α, combinations of variables involving α are omitted. To compare the histograms, the limits of the color scales in the difference histograms, labeled "DeepTree– CC Dataset 2," are set to ± one quarter of the maximum count in the histograms for the dataset distribution.

The hit energy vs. r histogram (Fig. 10.14) shows that high-energy hits are concentrated close to the beam axis ($r = 0$). While the model generates some outlier hits at very high energies, the overall distribution is well-reproduced, with the differences between the histograms being close to zero.

The hit energy vs. z histogram (Fig. 10.15) shows that the highest energy hits are most frequently deposited around layer 11. While the model reproduces this pattern very well, it also generates some outlier hits with very high energies, primarily near the peak of the distribution in layer 11. This issue is likely caused, or at least exacerbated, by the necessary aggregation of hits in this area (see Section 10.3.4).

Figure 10.16 shows the z vs. r histograms. Most hits are concentrated in the $r = 0$ bin and the $z \in \{1...9\}$ range. The model produces too few hits for the $r = 0$ bins (for $z \in \{1...18\}$) and too many in the $r = 1$, $z \in \{0...4\}$ bins. Note that each bin in this histogram combines 16 angular bins in the detector for 100 showers. Thus, the maximum count in each histogram bin is 1600. Many of the histogram bins in this area reach counts over 1400, indicating that all cells in this area are almost always (>87.5%) occupied. This high occupancy is inherently challenging for a PC-based model to reproduce accurately.

In the energy-weighted version of the same z vs. r histograms (Fig. 10.17), nearly all energy content is concentrated in the two innermost r layers and the first 20 z layers. Overall, the model accurately reproduces this distribution. However, it slightly overestimates the energy content in the $r = 1$, $z \in \{7...19\}$ bins.

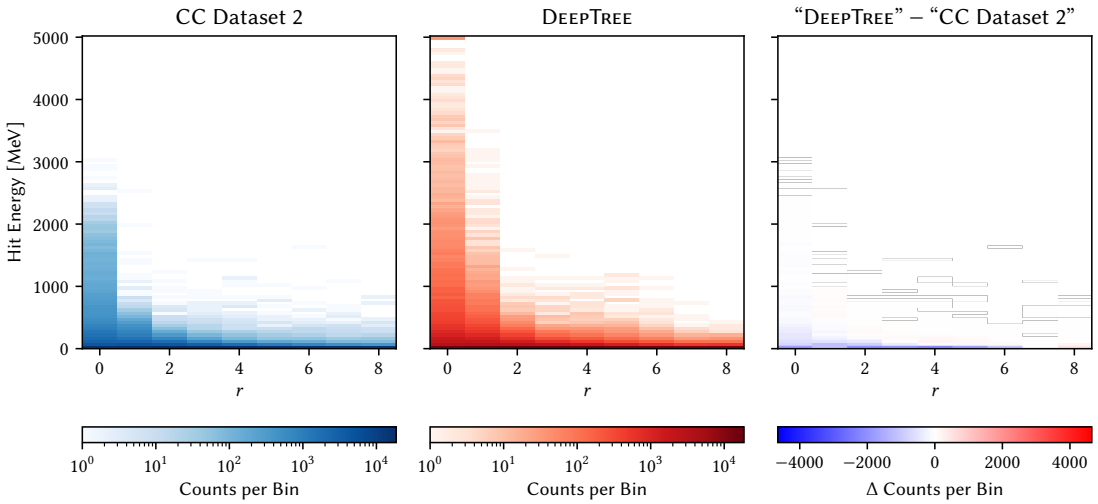


Figure 10.14. | 2D Histogram of the Hit Energy and r for the Dataset Distribution, the Generated Sample and the Difference between them.|
See Section 10.4.2 for the discussion. The histograms contain 100 showers. The plot design is explained in A.2.

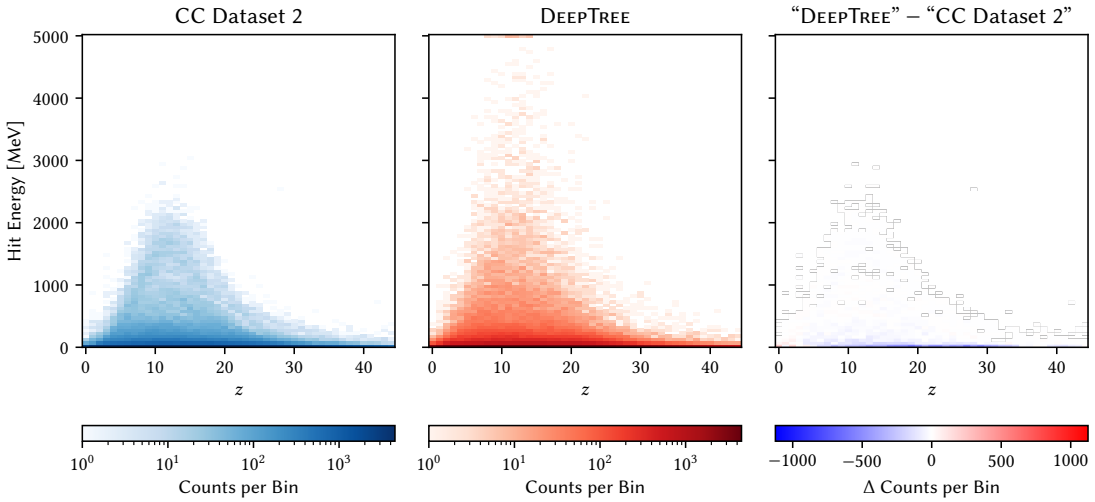


Figure 10.15. | 2D Histogram of the Hit Energy and z for the Dataset Distribution, the Generated Sample and the Difference between them.|
See Section 10.4.2 for the discussion. The histograms contain 100 showers. The plot design is explained in A.2.

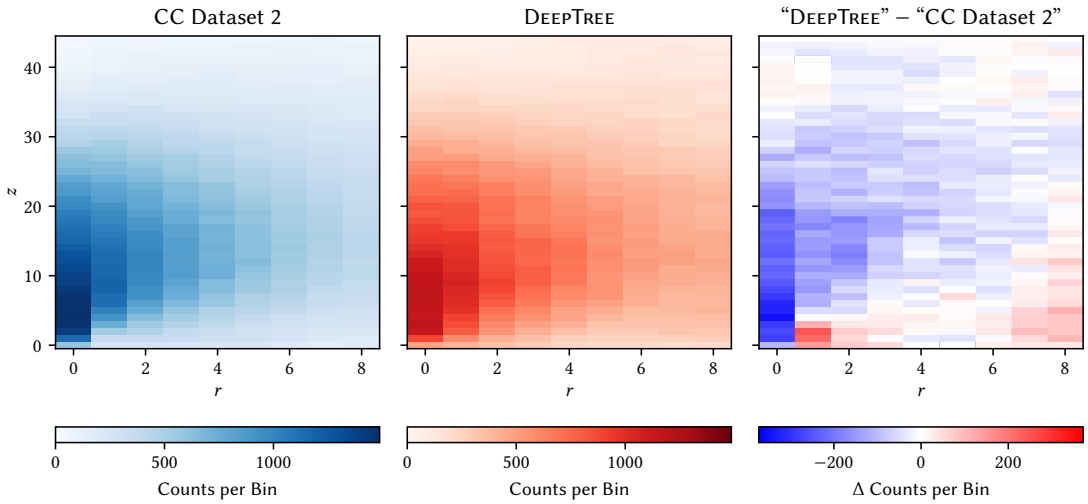


Figure 10.16. | 2D Histogram of z and r for the Dataset Distribution, the Generated Sample and the Difference between them.|
See Section 10.4.2 for the discussion. The histograms contain 100 showers. The plot design is explained in A.2.

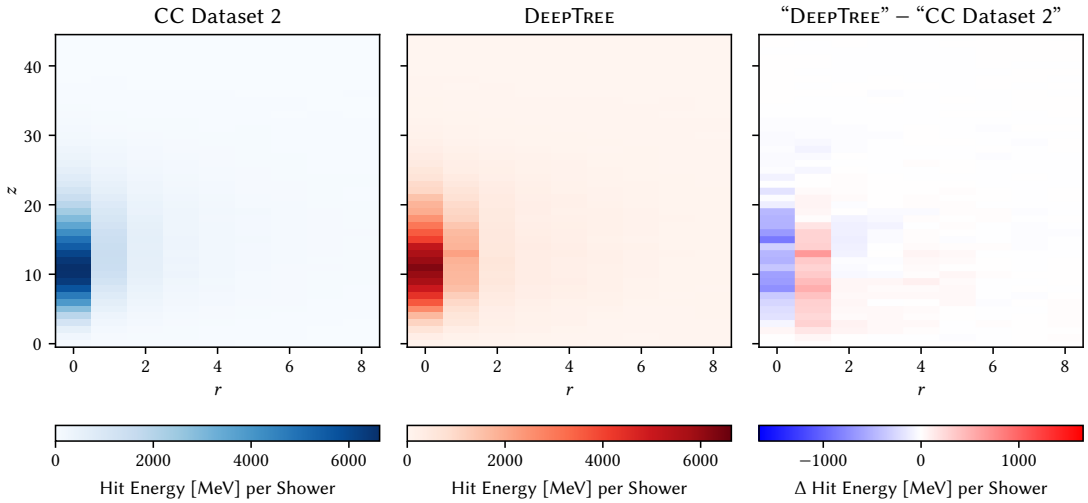


Figure 10.17. | Energy Weighted 2D Histogram of z and r for the Dataset Distribution, the Generated Sample and the Difference between them.|
See Section 10.4.2 for the discussion. The histograms contain 100 showers. The plot design is explained in A.2.

10.4.3. Shower Variables

Finally, the model's performance is evaluated based on shower-level variables instead of hit variables. Figures 10.18 to 10.21 present a set of computed shower variables.

Shower Development

First, the development of the shower along the z-axis is investigated. The distribution of the layer with the highest hit count ("peak layer") is shown in Fig. 10.18. In the model distribution, the peak is slightly too high and shifted to the right compared to the dataset distribution. To further analyze the shower shape, a turn-on/turn-off layer is defined as the first/last z layer to contain at least half of the cardinality of the peak layer. The corresponding histograms are shown in Figs. 10.19 and 10.20. The turn-on layer distribution peaks quickly in the second layer and vanishes by the 8th layer. While the model closely matches most layers to the simulation, the bin for layer 0 is overpopulated, and the highest bin in layer 1 is underpopulated. The turn-off layer distribution is much wider than the two preceding distributions, stretching roughly from layer 8 to layer 35. The generated distribution aligns well with the dataset distribution, with only minor deviations in the tails.

Response

One of the three conditioning variables for the model is the energy of the shower initiating particle, E. Figure 10.21 shows the response, defined as the sum of the produced hit energies divided by E. Both the model and dataset distributions peak at approximately 0.8, but the dataset distribution has a much narrower width. This suboptimal modeling of the response might be attributed to the gating mechanism (Appendix D.3). Although it was introduced following some initial success, the ablation study (Chapter 9) conducted afterward revealed that it was ultimately disadvantageous.

Density of the Shower

To investigate the modeling of energy density in the shower, the fractions of energy deposited in a sphere around the center of the shower are computed. This sphere is centered at the energy-weighted average of the hits in Euclidean coordinates. For each hit, the difference vector to the center is calculated. A hit is included in the sphere if the norm of the difference vector is less than a set "radius" r (here 0.3 and 0.8). To treat all coordinates equally, the standard deviation is computed for each coordinate, and its inverse is used to scale the distance to the center for that coordinate:

$$\left\| \begin{pmatrix} \sigma_x^{-1} \cdot (x - x_{\text{Center}}) \\ \sigma_y^{-1} \cdot (y - y_{\text{Center}}) \\ \sigma_z^{-1} \cdot (z - z_{\text{Center}}) \end{pmatrix} \right\| < r.$$

The histograms for the two spheres with radii $r = 0.3$ and $r = 0.8$ are shown in Fig. 10.22. For the $r = 0.3$ sphere, the tails are well-modeled, but the model distribution shows some deviations in the center of the rising distribution. For the $r = 0.8$ sphere, the ratio of model bins to simulation bins is mostly around 1. However, the peak around 0.87 is overpopulated, and the [0.75, 0.83] range is underpopulated by the model.

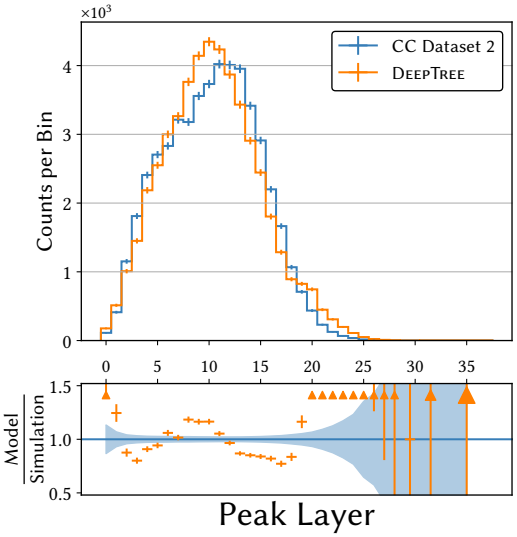


Figure 10.18. | Distribution of the z Layer with the Highest Hit Count.|
See Section 10.4.3 for the discussion. The plot design is explained in Appendix A.1.

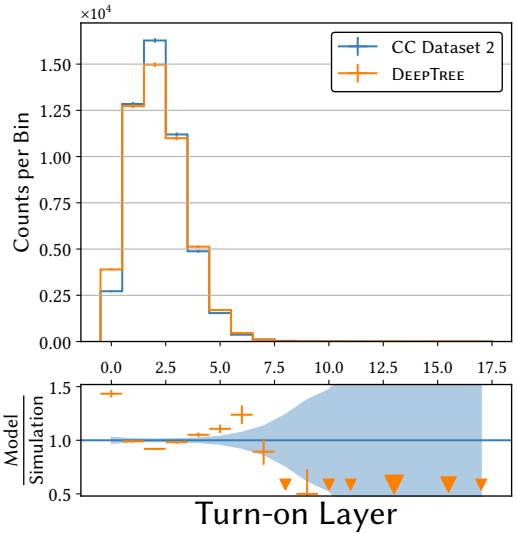


Figure 10.19. | Distribution of the first z Layer with at least half of the Hit Count of the Peak Layer.|
See Section 10.4.3 for the discussion. The plot design is explained in Appendix A.1.

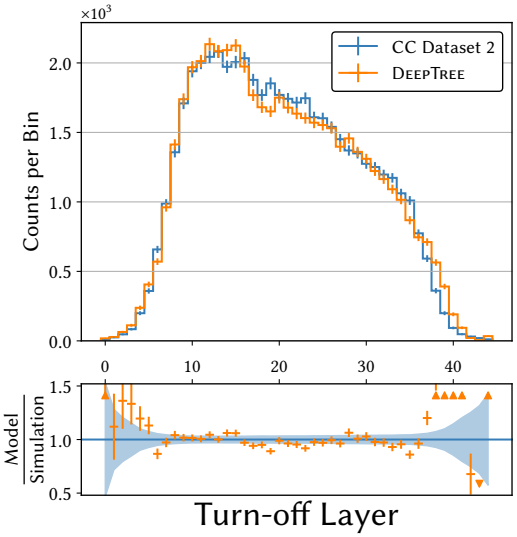


Figure 10.20. | Distribution of the last z Layer with at least half of the Hit Count of the Peak Layer.|
See Section 10.4.3 for the discussion. The plot design is explained in Appendix A.1.

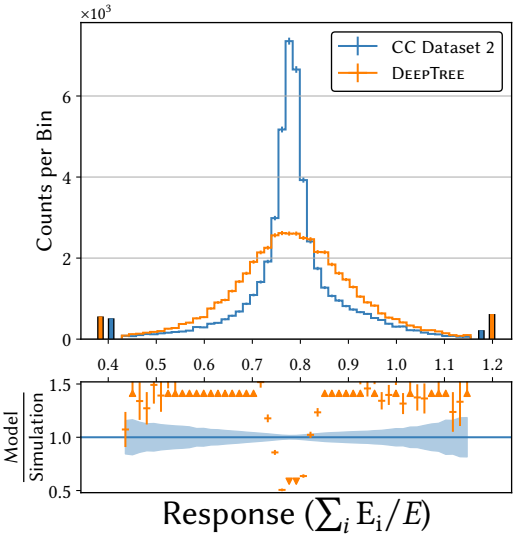


Figure 10.21. | Distributions of the Sum of Hit Energies over the Energy of the Shower Initiating Particle.|
See Section 10.4.3 for the discussion. The plot design is explained in Appendix A.1.

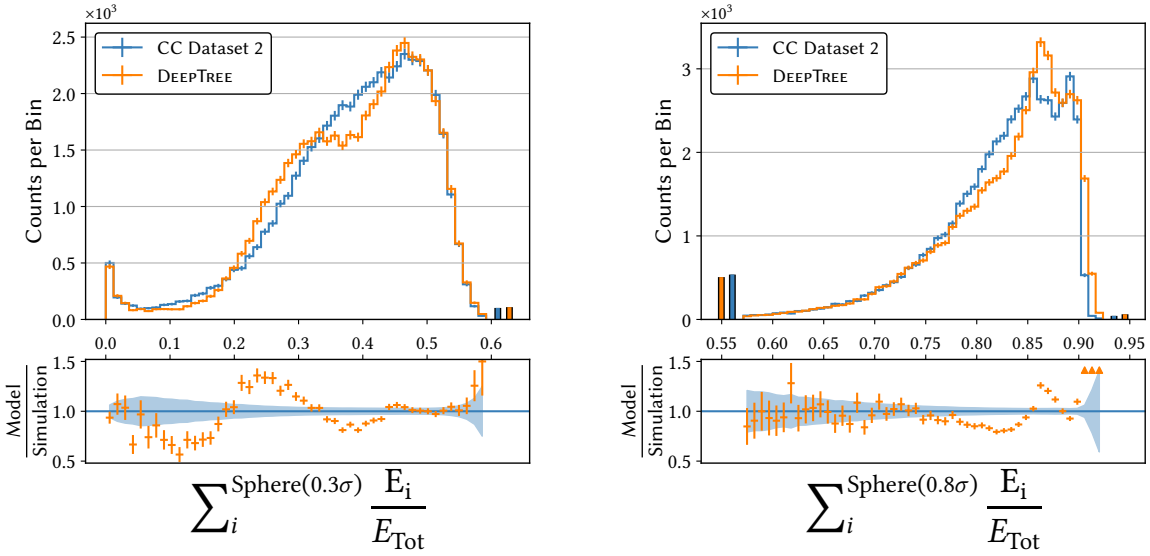


Figure 10.22. | Distribution of the Energy in inside a Sphere around the Center of the Shower.|
See Section 10.4.3 for the discussion. The plot design is explained in Appendix A.1.

10.4.4. Run Time Analysis

Applying hit-shifting enhances fidelity across all distributions, but it significantly increases the generation time. As shown in Table 10.2, hit-shifting accounts for approximately 90% of the total generation time. As detailed in Section 10.3.4, the algorithm's run time heavily depends on occupancy and is expected to decrease significantly in the targeted low-sparsity setting. Nevertheless, DeepTree remains one of the fastest models in the CaloChallenge competition, as detailed in the following section.

Table 10.2. | Run Time for the Different Steps in the Generation for 1000 batches of size of 100 using an Nvidia V100.|

Step	Time per shower [ms]
Generator	0.53
Hit-Shifting	4.49
Map PC to grid	0.06
Total time	5.08

10.5. Comparison to Other Models in the CaloChallenge Competition

In the CaloChallenge competition, the performance of DeepTree is compared to 17 other models on dataset 2. While the final performance evaluation is still pending [141], preliminary results are available [147].

Fidelity A range of classifier-based metrics is used to evaluate the fidelity of the models. For the first set of metrics, a classifier is trained to differentiate the model output from the dataset distribution. The harder it is for the classifier to separate the two classes, the better the generative model. In this comparison, DeepTree places 9th, 13th, and 15th depending on the classifier used (high-level, low-level, ResNet) [141, Tab. 20]. Another set of metrics is obtained by training a classifier to distinguish between the datasets of the different submitted models. The best generative model is the one to which the classifier assigns the highest score when classifying the dataset. In this evaluation, the DeepTree model places 14th and 15th [141, Tab. 14/Tab. 15]. In summary, the DeepTree model performs poorly in the classifier-based metrics.

However, a classifier may rely on a single discriminative feature to distinguish between two samples. In such cases, a classifier-based metric might down-rank a model based on one specific aspect, even if the model is otherwise more generalizable and versatile, and adaptable to various use cases, as outlined below.

Figure 52 in Ref. [141] shows a high separation power for the number of hits per layer for the DeepTree model. Additionally, the DeepTree model samples the cardinality it generates from the dataset. During postprocessing, hits assigned to the same cells may be combined (Section 10.3). As a result, the model produces fewer hits than expected (Fig. 10.9). Therefore, the number of hits, combined with the shower energy, can be used to distinguish samples generated by DeepTree. Moreover, the mismodeling in the response (Fig. 10.21) may also serve as a distinguishing feature for samples generated by DeepTree. Any of these easily constructed and discriminative features could explain the reduced performance on classifier-based metrics. This clearly underestimates the potential of the model.

In the KPD and FPD metrics, which are based on EFPs (Section 6.3), DeepTree ranks 8th and 5th, respectively [141, Tab. 19].

Run Time With postprocessing, DeepTree ranks 4th and 7th for a batch size of 100 on CPU and GPU, respectively [141, Tab. 24/Tab. 25]. Table 10.2 shows that hit-shifting accounts for the majority of the run time, while the model itself only accounts for approximately 10% of the run time. If this fraction is used to estimate the run time without postprocessing, DeepTree would likely rank first on both CPU and GPU. As detailed in Sections 10.3.4 and 10.4.4, the run time would be drastically lower in the high-sparsity setting, that DeepTree was designed.

Number of Parameters It is worth noting that DeepTree is the model with the second-lowest parameter count required for generation [141, Tab. 13].

10.6. Conclusion of the CaloChallenge Study

Fidelity

Overall, the model reproduces the investigated variables quite well. Notable issues include high-energy outlier hits, challenges in modeling the response, and undersampling in the $r = 0$ / $z \in \{1 \dots 18\}$ bins (Figs. 10.10, 10.16 and 10.21). However, these issues could likely be addressed with further improvements to the model.

Ultimately, it has become clear that several issues[4] arise from the PC-based nature of the model combined with the low sparsity of this dataset, for which the model was not originally designed. While hit-shifting (Section 10.3.4) significantly improves performance, it cannot fully recover the model's fidelity.

Further Improvements

Due to time constraints, the training of this model was completed before the ablation study (Chapter 9) and, therefore, could not benefit from its conclusions. This includes the observation that gating the generator likely deteriorates performance (`46-gen.cgated`). Further improvements could also be achieved by refining the pre- and postprocessing steps. The model might find Euclidean coordinates easier to handle than the provided cylindrical coordinates. While the transformation chain for the discrete cylindrical coordinates produces distributions that are quite close to a standard normal distribution (Figs. 10.4 to 10.2), the "CDF quantization" introduced in Ref. [100] could further improve fidelity without changing the model architecture. The transformation of the hit energy distribution (Section 10.3.2) also shows some challenging features that could affect the modeling in Fig. 10.10. In the hit-shifting (Section 10.3.4), prioritizing an axis with less variation from bin to bin, such as α over z over r, could help reduce the smearing effect.

Conclusion

As outlined in Section 10.1, the designated use case for the DeepTree model is orthogonal to the CaloChallenge datasets. Moreover, this dataset poses the test of whether this model can scale to such high cardinalities. The model passed this scaling test and was able to produce distributions with reasonable fidelity at this high cardinality. The properties of the dataset recommended a pixel- or voxel-based model (Section 10.1). Many of the models in the CaloChallenge competition follow this approach and yield better fidelity than the DeepTree model. However, considering that DeepTree is a novel model designed for an orthogonal use case and can be applied to any irregular calorimeter, this is an impressive result nevertheless. Moreover, the discussed improvements will likely further enhance the fidelity.

A final evaluation will take place in Ref. [141].

[4] E.g., in the z vs. r (Fig. 10.16) distribution.

Chapter Review In this chapter, the DeepTree model was applied to a particle shower dataset containing up to approximately 5600 hits, with each hit represented by a single point. A dedicated pre- and postprocessing strategy was developed to transform the discrete calorimeter cells into a continuous space (Milestone 6). Additionally, the developed hit-shifting algorithm significantly mitigated issues arising from multiple points being assigned to the same cells (Milestone 7). The achieved fidelity with this high number of points demonstrates the model's scaling capabilities and its applicability to particle showers (Milestone 8). The model's performance in the context of the CaloChallenge has been thoroughly discussed. In the following chapter, the model will be positioned within the current state of the art in generative modeling.

CHAPTER 11

Conclusion

With the ever-increasing number of events, LHC experiments are intensively searching for generative models to accelerate their calorimeter simulations [5]. A particularly challenging task is the generation of particle showers for the next generation of high-granularity calorimeters.

Current Landscape of Generative Models for Jets and Particle Showers

Since the start of this PhD project, the number of models for jet and calorimeter simulation has rapidly increased [15, 89, 94–96, 99–103, 106, 107, 110, 121, 137, 138, 149–168]. For jets, the PC-based approach (Chapter 1) has become predominant [99, 101–103, 121, 137, 166–168]. In contrast, for calorimeter simulation, PC-based models remain in the minority, as demonstrated by the CaloChallenge [147] competition. In Table 11.1, the models submitted to the competition are listed. Only 5 of the 22 submitted models are PC-based. When considering contributions for the high-granularity datasets 2 and 3, only 4 of the 16 models are PC-based. However, it should be noted that at the start of this PhD project, no PC-based model was available. Most models operating on datasets 2 and 3 use convolutions, which are easier to design because the simulated calorimeters in these datasets are completely regular. In contrast, PC-based models require more complex pre- and post-processing. Additionally, the CaloChallenge datasets are not sparse, meaning a large fraction of the cells contain hits. This scenario is the opposite of one where a PC-based model would excel. However, the prospect of moving to a realistic, irregular calorimeter with more than an order of magnitude more cells favors a PC-based approach.

Thesis Review

The main contribution of this thesis is the DeepTree model, which generates PCs representing the particle showers in a tree-based manner. The aim is to model the response of calorimeters. After the introduction (Chapter 1), the necessary background in both physics and machine learning (Section 2.2 and Chapter 3) was provided, followed by a detailed explanation of the DeepTree model in Chapter 5. As discussed in Chapter 1, jets can serve as a suitable proxy task for developing generative models for particle showers. The JetNet dataset (Chapter 6), which contains jets and their constituents, allows to first investigate the model's fidelity with a fast turnaround time, while the scalability is demonstrated later. The fidelity of the DeepTree model was demonstrated on this dataset in Chapters 7 and 8. In Chapter 9, an approach was developed to systematically review the design choices of the model. The model's scaling potential to higher cardinalities, necessary for generating particle showers, was demonstrated on the CaloChallenge dataset (Chapter 10).

Table 11.1. | Data Handling Strategy of Models Submitted to the CaloChallenge Competition.|
See Chapter 11 for the discussion. See Chapter 4 for the discussion of the different approaches. The name of the models is the one given in CaloChallenge [147].

Approach	Model	Data Handling	Dataset 1	2	3
GAN	CaloShowerGAN [149]	Dense	✓		
	MDMA [102]	PC		✓	✓
	BoloGAN [89]	Dense	✓		
	DeepTree [106, 107]	PC		✓	
NF	L2LFlows [150]	Pixel Sequence		✓	✓
	CaloFlow [151, 152]	Pixel Sequence	✓	✓	✓
	CaloINN [153]	Dense	✓	✓	
	SuperCalo [154]	Dense		✓	
	CaloPointFlow [155]	PC		✓	✓
Diffusion	CaloDiffusion [96]	GLaM[a]+ Voxels	✓	✓	✓
	CaloClouds [156, 157]	PC			✓
	CaloScore [138, 158]	Pixels/Voxels (Ds. 1/2+3)	✓	✓	✓
	CaloGraph [95]	PC	✓		
	CaloDiT [92]	Voxels+Tokens		✓	
VAE	Calo-VQ [94]	Tokens	✓	✓	✓
	CaloMan [159]	Dense	✓		
	DNNCaloSim[b] [93]	Dense	✓		
	Geant4-Transformer[c]	Voxels/Tokens			✓
	CaloINN+VAE [153]	Dense/Voxels (Ds. 1/2+3)	✓	✓	✓
	CaloLatent [160]	Voxels		✓	
CFM	CaloDREAM [161]	Voxels/Tokens		✓	✓
	CaloForest[d]	Dense	✓		

a See Section 4.3
b Also known as "CERN-Geneva VAE".
c To be published by Dalila Salamani et al.
d To be published by Jesse C. Cresswell and Taewoo Kim

Thesis Contribution

In summary, the main contributions of this thesis are as follows:

- DeepTree, a PC-based GAN.
  The concept of a tree-based generative model for PCs was advanced from a model that failed to produce a Gaussian distribution to a model capable of generating particle showers with thousands of hits. To implement a tree-based GAN, novel methods for the differentiable up- and downscaling of PCs were developed. PC upscaling and downscaling is a rarely focused area of machine learning, especially for large PCs with varying cardinalities. This is the first time that a tree-based generative model is used for the generation of a tree-like physics process. Additionally, it is the first graph-based model that changes the cardinality of the PC. The developed Bipartite Pool provides a fast, permutation-invariant, and generally applicable embedding of PCs (Section 5.2.1). During the ablation study (Chapter 9), several potential pathways for further improving DeepTree emerged, suggesting that its full capabilities of this approach have yet to be realized. The DeepTree model was published in Refs. [107, 146] and is part of the CaloChallenge competition [141], which is expected to be published soon. This makes DeepTree one of the first PC-based models to be developed, optimized, and benchmarked in a open calorimeter competition.

- caloutils [148], a fully PC-based library containing tools for handling calorimeter data and metrics for their evaluation.

- A CMS-internal tool [169] was implemented to extract the positions and neighborhood relations between cells. This neighborhood construction works across subdetectors and from one layer to the next.

- A contribution[1] was submitted to the popular GNN library PyG [77], enabling the manipulation of batches of PCs or graphs in PyG's dense representation.

These contributions have been recognized by the CMS collaboration as service work. For a full list of the public contributions, see Appendix G.

CaloChallenge Review

Employing PC-based models like DeepTree is advantageous when the calorimeter is irregular and highly granular, and the showers deposit their energy in a small fraction of cells (Section 10.1). While these assumptions do not hold for the CaloChallenge datasets, the competition establishes metrics and baselines to evaluate fidelity and run time. A model that demonstrates good fidelity and run time on the CaloChallenge dataset and can handle the HGCAL geometry is likely to perform well on HGCAL data. The preliminary results of the competition are discussed in Section 10.5. The DeepTree model demonstrates reasonable fidelity and run time. *This is an impressive result for a novel model designed for a use case orthogonal to the CaloChallenge datasets, highlighting its remarkable flexibility and making it a strong starting point for a wide range of application.* While some models show a better fidelity, most use approaches that *cannot be applied* to calorimeters with irregular geometries or are *unlikely to scale* to highly granular calorimeters (Chapter 4). For

[1] Currently under review: github.com/pyg-team/pytorch_geometric/pull/8414

DeepTree, on the other hand, the path forward is much clearer, as detailed in the following. Moreover, due to time constraints, the ablation study took place after the model had been submitted to the CaloChallenge competition. Consequently, numerous significant improvements could not be included.

Perspectives for the upcoming CMS High Granularity Calorimeter

This PhD project was conducted within the CMS collaboration. It aimed to explore the application of generative modeling for the future CMS High Granularity Calorimeter (HGCAL). The assumptions for the DeepTree model developed in this thesis are motivated by the specific requirements and characteristics of the HGCAL. The final step of such an endeavor would be the integration of the generative model into the software framework of the CMS experiment and the HGCAL, leading to the routine application of the presented methods and tools. Integrating a model is only worthwhile if its fidelity is high enough, its runtime behavior is acceptable, it can handle the geometry of the HGCAL, and it has been thoroughly tested by the collaboration. At present, no existing model fully satisfies these criteria.

While several challenges need to be addressed, rapid progress has been made in overcoming them:

1 High-Fidelity Models
As stated, the number of available models has greatly increased since the start of this thesis. Among these is the DeepTree model, which is designed to scale well and handle arbitrary geometries due to its PC-based nature.

2 Sensitive Metrics
In Ref. [121], a set of sensitive metrics is developed and studied in detail, implemented in the JetNet library [18]. These metrics were originally developed for jets but could be adapted for calorimeters and are used, among other metrics, in the CaloChallenge. While these metrics have an acceptable run time for evaluation, they are too slow for validation during training. My caloutils library [148] implements a set of fast, physics-inspired metrics for PCs. These metrics are based on the shower-level variables used in Section 10.4.3.

3 A Benchmark Dataset with the Scale and Complexity of the HGCAL
Currently, no publicly available dataset exists for the HGCAL. This restricts the publication of results produced with HGCAL simulated data and means that no competitive benchmarks are available. This limitation is one reason why the DeepTree model was scaled to the CaloChallenge dataset instead of an internal CMS HGCAL dataset. The CMS collaboration has recognized the need for a public dataset and plans to publish one shortly. I was involved in defining this dataset, and the internal datasets created in this context will likely serve as the template for the public version. Once the dataset is published, the community will likely provide valuable benchmarks.

4 PC-based Tool Chain for HGCAL Showers
The extremely high cell count and irregular structure of HGCAL strongly suggest

the use of a PC-based model (Section 4.3). Training and evaluating such a model requires an extensive pre- and post-processing chain. The data structure of a shower is a set of tuples containing the cell IDs and the energies for each hit. It can easily be converted to a PC by replacing the cell IDs with the positions of the cells. As described in Section 4.3, dequantizing the cell positions during training is advantageous, as implemented for the CaloChallenge processing in Section 10.3. During a hackathon [170], I developed an algorithm [171] that allows dequantization of cells from arbitrary geometries based on their neighbors. After the PC is generated, it must be quantized, meaning the points need to be assigned to calorimeter cells. An algorithm to quantize PCs to the HGCAL cells still needs to be implemented. An efficient option for finding the nearest cell to each point would be a k-d tree. With such a 'nearest cell' search, multiple points may be assigned to the same cell. caloutils includes an algorithm that mitigates this problem by moving energy depositions to a neighboring empty cell. This PC-based algorithm is implemented for the CaloChallenge datasets but could be adapted to the HGCAL with moderate effort.

5 Generating Events instead of Showers

The currently developed models focus on generating showers for a single particle rather than an entire event. To be usable in an analysis, either algorithms to combine these showers must be developed, or generative models need to produce complete events with their full complexity.

Outlook

While the DeepTree model generates large PCs quickly, its major drawbacks are the run time and instability of the training typical for GANs. In the CaloChallenge [147] competition, a wide range of architectures are compared. Although the results are not yet final, preliminary results of the CaloChallenge competition [147] indicate that score-based and "conditional flow matching" [113] (CFM) approaches most often yield the highest fidelity. CFM models, in particular, have recently gained popularity in the community for their stability and high fidelity, though this comes at the cost of slower generation speed. For JetNet, the dominant approach has already shifted from GANs to CFM. Both MDMA and EPiC-GAN, which provide baselines in this thesis, have CFM versions ([112] and [137]) that achieve higher fidelity but also slower generation speeds compared to their GAN predecessors. Moreover, the current state-of-the-art model on JetNet-150 is a CFM model [110].

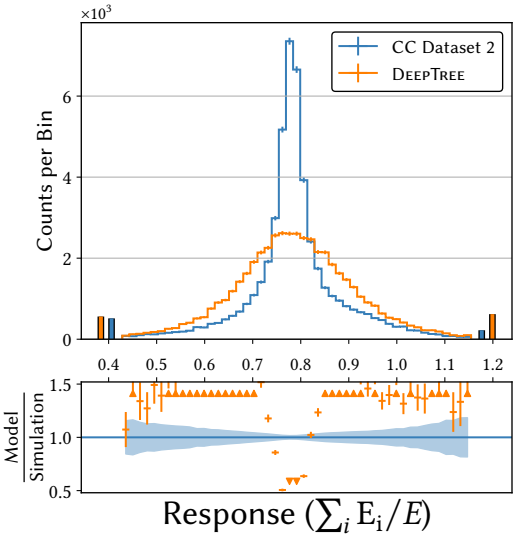
To use this objective, a model would need to map from a PC to a PC for a given condition. In image generation, models frequently use a UNet-like [172] architecture that iteratively downscales and then iteratively upscales the image. At each level, skip connections are added between the downscaled and upscaled images. Using the PC up- and downscaling methods developed in this PhD project, a CFM model could be constructed that extends the UNet structure to PCs.

This demonstrates that the developed concepts are ready to be combined with more recent generative approaches, maximizing the benefits of both methods.

Appendix A.

Plot Descriptions

A.1. 1D Histogram



The plot is divided into two subplots: a histogram on the top and a ratio plot on the bottom. The framed vertical bars in the color of the model (orange) or the dataset (blue) indicate the number of samples outside the range of the bins, i.e., over- and underflow. Their position on the x-axis is arbitrary. These bars are omitted if all data fall within the range of the bins. If the histogram is weighted, the weights are scaled by the average weights in the dataset. Error bars indicate the statistical uncertainty in both plots. The ratio plot shows the ratio of counts in the model distribution to those in the dataset distribution, represented by orange crosses. Triangle markers at the top or bottom of the subplot indicate when the ratio exceeds 1.52 or falls below 0.48. To represent the uncertainty in the dataset distribution itself, the error of the ratio between the dataset and itself is calculated and drawn as a blue band around 1. This band is displayed only if the dataset distribution count in the bin is at least one.

The error σ is computed for each bin from the counts n:

$$\sigma = \sqrt{n}.$$

For a weighted histogram, with event weights w_i, the error is calculated as:

$$\sigma = \sqrt{\sum_{i=1}^{n} w_i^2}.$$

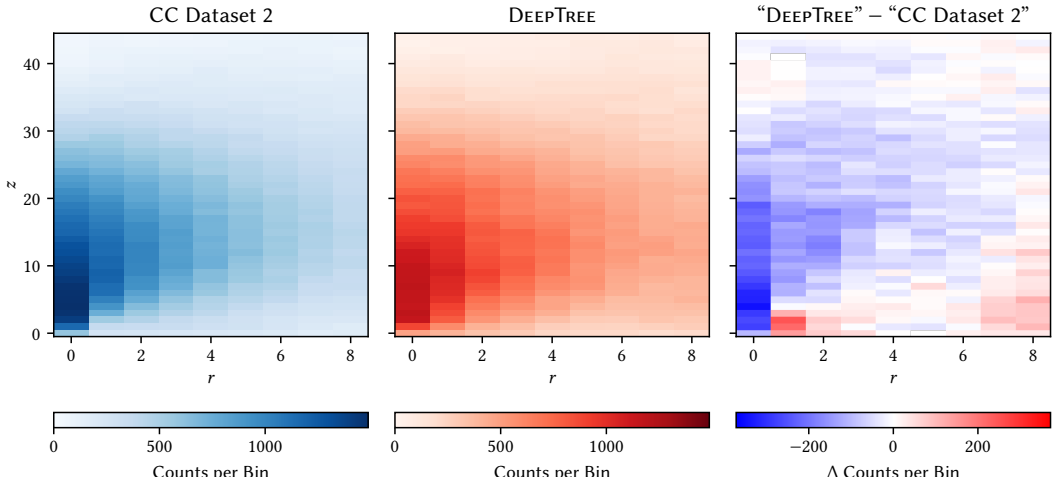
The error on the ratio is then computed from the error on the bin contents:

$$\sigma_{\text{ratio}} = \frac{n_{\text{gen}}}{n_{\text{sim}}} \cdot \sqrt{\left(\frac{\sigma_{\text{sim}}}{n_{\text{sim}}}\right)^2 + \left(\frac{\sigma_{\text{gen}}}{n_{\text{gen}}}\right)^2}.$$

If the number of data points becomes large, the error bars may not be visible. In the ratio plot, the error on the x-axis corresponds to the width of the bin.

In the histograms of continuous variables, the number of bins is set to 50. For discrete variables, such as the layer index of the cardinality, each unique value is placed in a separate bin.

A.2. 2D Histograms



This plot consists of three 2D heatmaps with the same axes and binning. The first two heatmaps display the 2D distribution of hits from 100 showers, either from the dataset or the model. The third heatmap highlights areas where the model has more (red) or fewer (blue) data than the dataset. If the histogram is weighted, the weights are normalized by the sum of the weights in the dataset. The maximum of the color scale for the difference histogram is set to half of the maximum count in the dataset histogram. The contours in the histogram are produced using Seaborn's [173] `kdeplot` with default hyperparameters, and they enclose 30%, 60%, and 90% of the points.

Appendix B.

Quantile Transformer

Sklearn's `QuantileTransformer` [133] implements the "Inverse transformation method" [174] to transform between a source distribution X and a target distribution Y. This method uses an empirical approximation of the CDF of X and the known quantile function of Y. The quantile function F^{-1} is the inverse of the CDF F[1] for a random variable X:

$$F(x) = P(X \leq x), \tag{B.1}$$

$$F^{-1}(p) = \inf\{x : F(x) \geq p\}, \tag{B.2}$$

Thus,

$$\begin{aligned} F^{-1}(F(\tilde{x})) &= \inf\{x : F(x) \geq F(\tilde{x})\} \\ &= \inf\{x : x \geq \tilde{x}\} \\ &= \tilde{x}. \end{aligned}$$

In other words, the quantile $F^{-1}(p)$ is the lowest value such that the probability of drawing a smaller (or equal) value is at least p. While the CDF maps the random variable to the probability $p \in [0, 1]$, the quantile function maps the probability back to the random variable. Applying the CDF F to its random variable X yields a standard uniform ($\mathcal{U}([0, 1])$) distribution [174]:

$$P(F(X) \leq x) = P(X \leq F^{-1}(x)) = F(F^{-1}(x)) = x,$$

and thus,

$$F(X) \sim \mathcal{U}([0, 1]).$$

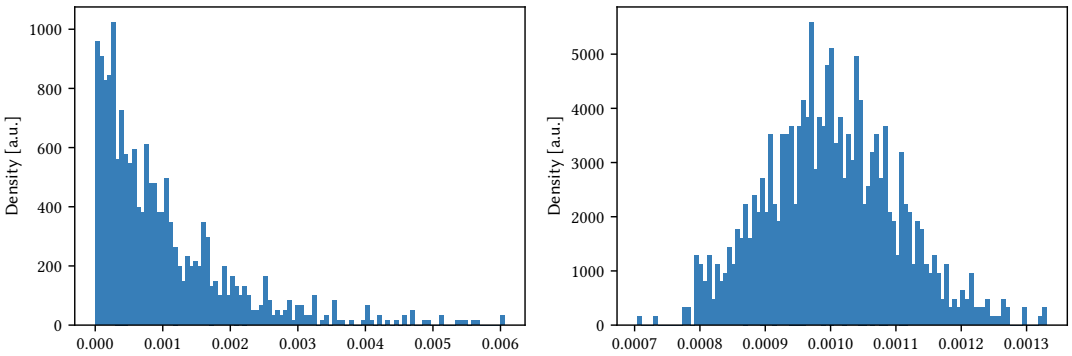
By chaining the CDF F_X of a source distribution X with the quantile function F_Y^{-1} of a target distribution Y, we can transform X into Y:

$$P(Y \leq x) = P(F_Y(Y) \leq F_Y(x)) = P(U \leq F_Y(x)) = P(F_X(X) \leq F_Y(x)) = P(F_Y^{-1}(F_X(X)) \leq x),$$

with $U \sim \mathcal{U}([0, 1])$.

Most often, Y is the standard normal distributed, and its quantile function F_Y^{-1} can be numerically computed. The CDF of the source distribution F_X must be estimated from the sample. In a regression task, this transformation would be applied in this direction. In a generation task, features should be produced in the domain of Y and then transformed back to the (unknown) source distribution X. To achieve this, the transformation needs to be inverted, using $F_X^{-1}(F_Y(Y))$.

[1] F is assumed to be invertible.



(a) Computed on n_s=10k Samples. **(b)** Computed on n_s=100k Samples.

Figure B.1. | Histogram of the Distances between one Quantile to the Following.| See Section 6.4 for the discussion.

`QuantileTransformer`

The working principle of the `QuantileTransformer` can be summarized as follows: To fit the transformation to the source distribution X, a set of n_r reference points is computed for quantiles distributed along [0,1] on a subset of the training sample with size n_s. An empirical estimate of the CDF (eCDF) is then constructed by linearly interpolating between the quantiles over the reference points.

In the forward transformation, the eCDF is first evaluated to map the values to [0,1]. These values are then transformed to a standard normal distribution using the quantile function ("probit").

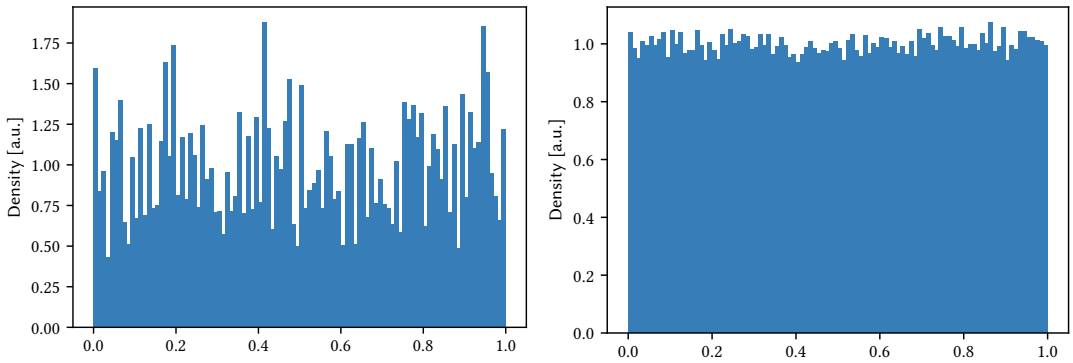
In a generative setting, the inverse transformation is also needed to map the generated features back to their original space. First, the CDF of the standard normal distribution is applied to the generated data to map them to [0,1]. Then, an empirical estimate of the quantile function is used to map the values back to the original distribution of X. This quantile function is approximated by interpolating the recorded reference points over the quantiles, in exactly the opposite way to the eCDF.

`QuantileTransformer` Failure Mode

If the subsample size is not sufficiently large, the distances between the quantiles, on which the transformation is based, become susceptible to statistical fluctuations. In the transformation, the CDF of the source distribution is approximated by using a linear interpolation that maps reference points to quantiles. This means that an interval with a low (high) quantile distance will contain fewer (more) data than expected. This effect is reversed in the inverted transformation.

In this example, the `QuantileTransformer` is used to transform from and to a standard uniform distribution. The 1k quantiles, which are the SKLEARN default, are estimated on n_s = 10k (subsample `a`, SKLEARN default) or 100k (subsample `b`) samples.

Figure B.1 shows the distribution of the differences between consecutive quantiles. These differences correspond to a probability interval that is mapped to a value between the two bordering reference points. To perfectly reproduce the uniform distribution, these differences would need to be equal to 1 divided by the number of quantiles (=0.001). The distribution for `a` features a sharp peak at 0 and then falls off steeply. In contrast, the distribution for `b` exhibits a Gaussian-like shape centered at 0.001.



(a) Computed on n_s=10k Samples.

(b) Computed on n_s=100k Samples.

Figure B.2. | Histogram of a Transformed Uniform Sample.|
The sample is independent of the sample that was used to fit the quantiles. See Section 6.4 for the discussion.

The result of the forward transformation with an independent sample is shown in Fig. B.2. While the distribution for `b` closely resembles a uniform distribution, the bin counts in the histogram for `a` vary significantly. The same effect occurs when the inverse transformation is applied to an independent sample.

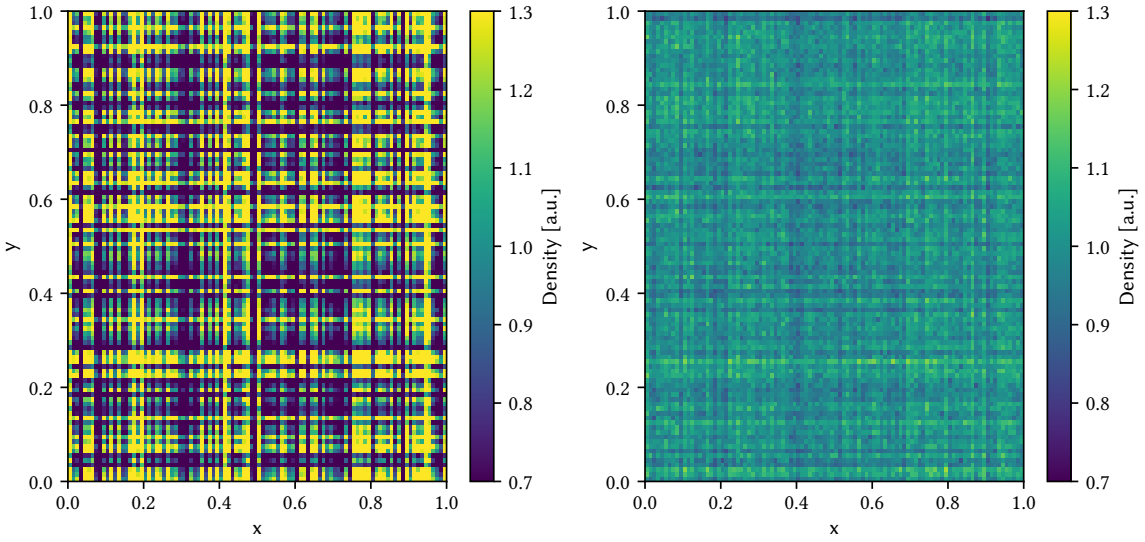
Checkerboard Artifacts for a 2D Distribution

The same fitting, scaling, and transformation process is now repeated for another independent variable following the same uniform distribution. In Fig. B.3, the combination of the variables is shown for both subsamples. For subsample `a`, this results in a distinct checkerboard pattern with alternating stripes of lower density (dark) and higher density (bright). In contrast, for subsample `b`, this effect is greatly reduced, and the density is almost constant, as desired.

Conclusion

The `QuantileTransformer` can be very useful, as it transforms any distribution to a standard normal distribution. However, it may also produce unintended artifacts, particularly if the dataset used to estimate the quantiles is too small. The default sample size of 10k is insufficient for the default value of 1000 quantiles. This issue becomes especially problematic when the `QuantileTransformer` is applied to multiple variables simultaneously. These artifacts can occur in both the forward and inverse transformations.

Typically, the sample used to fit the `QuantileTransformer` is the same sample that is transformed, in which case the described issues do not appear. However, in a generative model, the inverse transformation is applied to the output variables. Using a `QuantileTransformer` to transform multiple output variables without fitting the quantiles on a sufficiently large sample will likely cause the described issues. Thus, the `QuantileTransformer` is applied only to the conditioning variables in Section 10.3.



(a) Computed on n_s=10k Samples. **(b)** Computed on n_s=100k Samples.

Figure B.3. | 2D Histogram of two Standard Uniform Samples yields a Checkerboard pattern after the Transformation.|
See Section 6.4 for the discussion.

Appendix C.

Superseded Design Alternatives

C.1. Equivariant Branching Mechanism

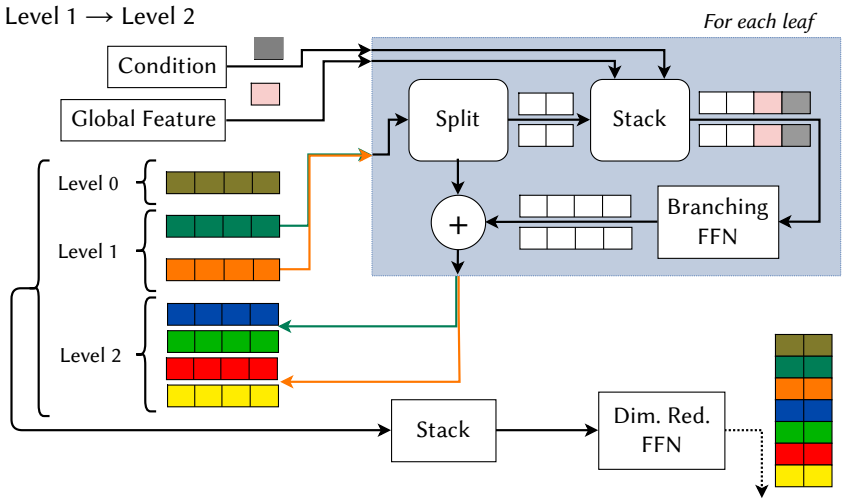


Figure C.1. | Equivariant Branching Mechanism.|
See Appendix C.1 for the discussion. This branching mechanism was used for the configuration `33-gen.br.eqv`, described in Section 9.4.2.

As PCs have no inherent ordering, NN architectures for PCs are often designed to be permutation invariant (if they map to a vector or scalar) or equivariant (if they preserve cardinality). This design choice offers the advantage that equivariance or invariance does not need to be learned from the dataset. The DeepTree critic is permutation invariant by design (see Section 5.2). However, the generator, particularly the branching layers, cannot achieve this, as it increases the cardinality.

In Fig. C.1, a branching layer is shown that implements this invariance by treating each node in the tree as a concatenation of its children, thereby preserving the cardinality.

In the branching layer, the parent is split into its b children, with the condition and global vector appended to each child. The branching FFN (FFN^{BR}) then maps each child to the dimension of its parent, before the dimensionality is reduced by another FFN[1].

Compared to the default branching (Section 5.1.1), the parent is split into b children z_i

[1] As the Dimensionality Reduction FFN has no effect on the following calculation, it will be omitted.

before the branching FFN is applied, rather than after. This approach makes the siblings, i.e., the children of the same parent, independent of each other. Thus, the branching layer B becomes equivariant under permutation P:

$$B(P\{z_1, z_2\}) = B(\{z_2, z_1\}) = P(\{\mathrm{FFN}^{\mathrm{BR}}(z_1), \mathrm{FFN}^{\mathrm{BR}}(z_2)\}) = P(B(\{z_1, z_2\}))$$

For multiple sequential branching layers ($\bigcirc_l B_l$), this extends to any permutation $S_{i\leftrightarrow j}$:

$$\begin{aligned}
&S_{i\leftrightarrow j}\left(\bigcirc_l B_l \vec{z}\right)\\
=&S_{i\leftrightarrow j}\left\{\bigcirc_l \mathrm{FFN}_l^{\mathrm{BR}} z_1, \ldots, \bigcirc_l \mathrm{FFN}_l^{\mathrm{BR}} z_i, \ldots \bigcirc_l \mathrm{FFN}_l^{\mathrm{BR}} z_j, \ldots \bigcirc_l \mathrm{FFN}_l^{\mathrm{BR}} z_n\right\}\\
=&\left\{\bigcirc_l \mathrm{FFN}_l^{\mathrm{BR}} z_1, \ldots, \bigcirc_l \mathrm{FFN}_l^{\mathrm{BR}} z_j, \ldots \bigcirc_l \mathrm{FFN}_l^{\mathrm{BR}} z_i, \ldots \bigcirc_l \mathrm{FFN}_l^{\mathrm{BR}} z_n\right\}\\
=&\bigcirc_l B_l\left(S_{i\leftrightarrow j}\vec{z}\right).
\end{aligned}$$

In practice, the global feature and the condition are stacked onto each part and passed through the branching FFN. The global feature is constructed in a permutation-invariant way from the leaves (Section 5.1.3), and the condition is independent of the entire tree. Therefore, adding them does not break the permutation equivariance.

If the ancestor MPL is set aside, the entire generator becomes permutation equivariant. However, this also means that information between points is exchanged only through the global feature constructed from the parents.

Break of the Equivariance through the Ancestor MPL

However, with the ancestor MPL, every node is updated with messages from all its ancestors. This means that the generator is only invariant to permutations that exchange two children within the same branch of the tree. An example of permitted and forbidden permutations is shown in Fig. C.2. The tree has two levels, a branching factor of 2, and produces the points $\{a, b, c, d\}$:

- a and b, or c and d, can be exchanged, as they share the same parent.
- a and b can be exchanged with c and d because the parents of these pairs share the same parent, the root node.
- a or b cannot be exchanged with c or d.

As the number of levels increases, the fraction of allowed permutations also increases.

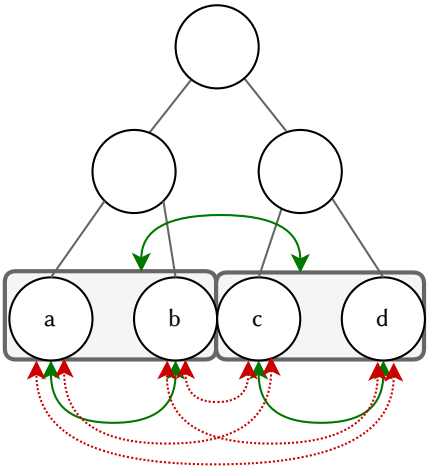


Figure C.2. | Example of the Allowed (in green/solid) and Forbidden (in red/dotted) Permutations for the Ancestor Invariant Branching.|
See Section 9.4 for the discussion. While the branching and global feature layer would allow all permutations, the ancestor MPL breaks the red/dotted permutations.

C.2. Noise Branching Mechanism

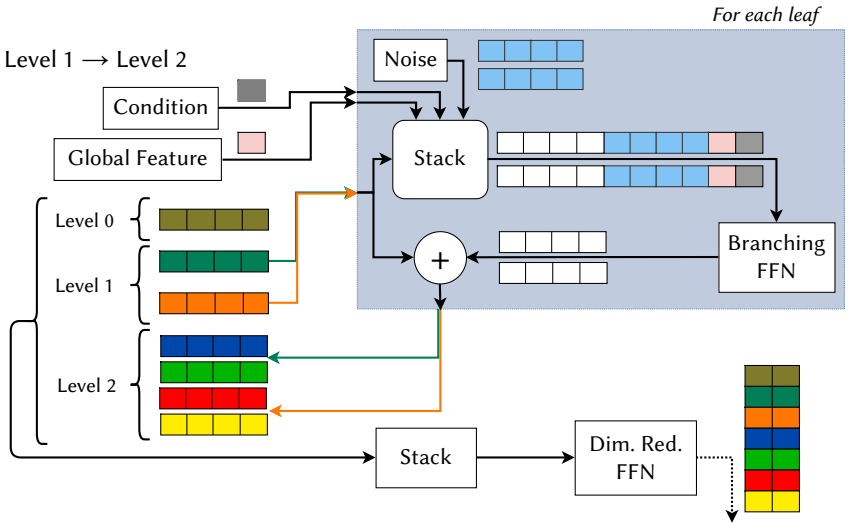


Figure C.3. | Noise Branching Mechanism.|
See Appendix C.2 for the discussion. This branching mechanism was used for configuration `34-gen.br.noise`, described in Section 9.4.2.

An alternative approach to the branching mechanism is shown in Fig. C.3. For each child, the parent (size p) is concatenated with:

- a new noise vector of size p, sampled from $\mathcal{U}[0, 1)$,
- the global feature (size g),
- the condition (size c).

To avoid extremely small or large vectors, the dimension of the noise vector is bounded within [5,15]. Let b be the branching factor. For the branching FFN, the input and output dimensions change from $\mathbb{R}^{p+g+c}$ and $\mathbb{R}^{b \cdot p}$ to $\mathbb{R}^{p+p+g+c}$ and $\mathbb{R}^{p}$, respectively, compared to the default branching mechanism (Fig. 5.2). This means the input dimension is increased, while the output dimension is drastically reduced. In the default branching mechanism, the children of the same parent originate from a single output vector of the branching FFN, meaning they depend on each other. In this alternative mechanism, as with the equivariant branching mechanism, the children of the same parent are independent of each other.

C.3. Ordering of Linear Layer, Normalization, and Activation in FFNs

Figure C.4 shows the distributions of activations for two FFNs. In the FFN shown in the right column, Batch Normalization is applied after the LeakyReLU activation, while in the left column, the order is reversed. The first layer of both networks is a linear layer. A batch size of 10k is used, and the number of hidden nodes is set to 10. The other hyperparameters are described in Section 5.3.4. As input, 10 standard normal distributed values were given to the newly initialized FFNs. The matrices from the linear layers contain values drawn from a uniform distribution, symmetric around 0 (Section 3.1.1). When standard normal distributed data is passed through these layers, a Gaussian-like distribution with a standard deviation of less than 1 (histograms "(0)Linear") is produced.

In the right column, the linear layer is followed by LeakyReLU, then Batch Normalization. LeakyReLU shifts negative values closer to 0, leaving positive values unaffected. This reduces the standard deviation and because the distribution is symmetric around 0, the mean becomes positive while the median remains 0 ("(1)LeakyReLU"). Batch Normalization then shifts the distribution to a mean of 0 and scales the standard deviation to 1. Due to the changes introduced by LeakyReLU, the following Batch Normalization ("(2)BatchNorm1d") requires a larger factor to scale the standard deviation up to 1 and needs to shift the mean back to zero. Together, this results in a distribution with a heavy tail to the right and a large peak around ≈ -0.5.

In the left column, where Batch Normalization is applied before LeakyReLU, swapping their order sharpens the peak, shifts it to 0, and reduces the tail to the right ("(2)LeakyReLU").

Therefore, the distribution entering all subsequent linear layers differs between the networks, potentially leading to different performance outcomes.

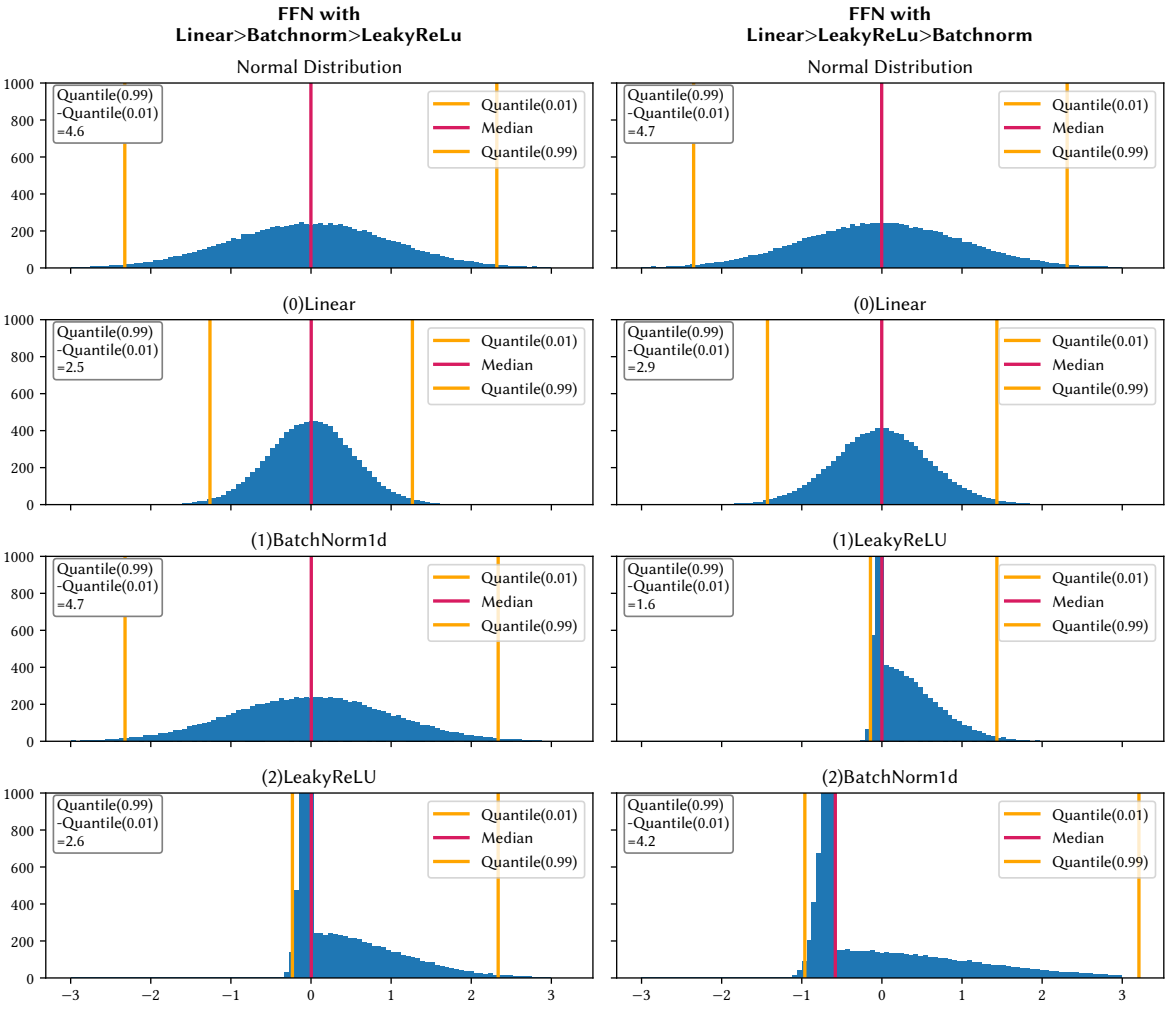


Figure C.4. | Comparison of the Order of the Activation and Normalization Layers inside an FFN.|
See Appendix C.3 for the discussion.

Appendix D.

Listings

D.1. Jet Momentum Rescaling

```
# pTi is the tensor holding the p_T^rel values
pTi = pTi.double() * σ + μ # increase percision from float to double
## Backwards transform (1/2) of the StandardScaler
# Limit tranformed values of pTi to 30, to avoid exceeding the addressable space
idx_to_large = pTi > 30
offset = pTi[idx_to_large].detach() - 30 # remove gradient from problematic pTis values with `detach`
pTi[idx_to_large] = pTi[idx_to_large] - offset # shift pTis to max 30
## Backwards transform (2/2) of the BoxCox transformation
if λ == 0: pTi = torch.exp(pTi)
else: pTi = torch.pow(pTi * λ + 1, 1 / λ)
# Calculate the sum over pTi for each jet with scatter_add from torch_scatter
# batchidx encodes, to which jet each constituent pTi belongs
ptsum_per_batch = scatter_add(pTi, batchidx, dim=-1)
## Normalize pTi values
pTi = pTi / ptsum_per_batch[batchidx]
## Forward transform (1/2) of the BoxCox transformation
if λ == 0: pTi = torch.log(pTi)
else: pTi = (torch.pow(pTi, λ) - 1) / λ
# Revert introduced shift
pTi[idx_to_large] = pTi[idx_to_large] + offset
## Forward transform (2/2) of the StandardScaler
pTi = (pTi - μ) / σ
```

The listing above shows the algorithm for jet momentum rescaling (Section 6.5). Since the model is trained to reproduce the transformed distribution, the transformation must first be inverted before normalization can occur. After normalization, the PC is passed to the critic, so the transformation needs to be reapplied to the normalized data. To train the generator, the gradient from the critic must pass through these operations. Thus, the algorithm must be differentiable. First, the $p_{\mathrm{T}}^{\mathrm{rel}}$ of the component is unscaled using the inverse Box-Cox transformation with standardizing (line 2). Then, the `scatter_add` function from PyTorch Scatter [175] is used to sum the p_{T} of the constituents for each jet

in the batch. To avoid exceeding the numerical range covered by the floating-point type in the power function, the precision is switched from `float` to `double`, and the values are capped at 30. This cap is only needed during the initial steps of model training.

D.2. Nearest Neighbor Distance Loss

```python
import torch
from caloutils.distances import energy_distance
from torch_geometric.nn import knn_graph
from torch_geometric.data import Batch

def scale_b_to_a(a: torch.Tensor, b: torch.Tensor):
  # Scale tensor b to the same mean and standard deviation as tensor a.
  mean, std = a.mean(), a.std()
  return (a - mean)/std, (b - mean)/std
def nndist(batch: Batch, space):
  # knn_graph produces tuples with the indices of points next to each other
  ei = knn_graph(batch.x[:, space], k=1, batch=batch.batch)
  # convert the tuples to distances
  return (x[ei[0]] - x[ei[1]]).abs().mean(1)

loss = torch.tensor(0.0)
# Compute the distance in both the energy and the position space
for space in [[energy_index], position_indices]:
  dists_sim = nndist(sim_batch, space)
  dists_gen = nndist(gen_batch, space)
  nnd_sim_scaled, nnd_gen_scaled = scale_b_to_a(dists_sim, dists_gen)
  # Compute the distance between the distributions
  loss += energy_distance(nnd_sim_scaled, nnd_gen_scaled)
```

Above, the PyTorch code for the `nndist` loss term is shown. In `nndist`, the nearest neighbor in the selected space is computed using PyG's `k-NN_graph` function. This selected space (`space`) is either the energy or the position space. In this space, the L_1 distance between the nearest neighbors is calculated (`nndist`). The mean μ and standard deviation σ of `delta` are computed for the distances of the training dataset and used to scale both delta distributions with $x \to \frac{x-\mu}{\sigma}$ (`scale_b_to_a`). After scaling, the distance between the two distributions is measured using the energy (or Wasserstein-2) distance (`energy_distance`). The implementation of the energy distance used in CALOUTILS [148] is based on SciPy but was reimplemented in PyTorch to allow for automatic differentiation.

D.3. Gated Conditioning Unit

```python
from torch.nn import Linear, Module, LeakyReLU
class GatedCondition(Module):
    def __init__(
        self, x_dim: int, cond_dim: int, out_dim: int
    ) -> None:
        super().__init__()
        self.out_dim = out_dim
        self.x_tf = Linear(x_dim, 2 * out_dim)
        self.cond_tf = Linear(cond_dim, 2 * out_dim)
        self.act = LeakyReLU(0.1)
    def forward(self, x, cond):
        common = self.x_tf(x) + self.cond_tf(cond)
        a, b = common.split(self.out_dim, -1)
        return a * self.act(b)
```

Inspired by the Gated Linear Unit [176], a Gated Conditioning Unit is defined in the code above. This unit combines the two input vectors (`x` and `cond`) and maps them to the dimension of `x`. First, both inputs are mapped to twice the output dimension using two linear layers. The results are then summed (`common`) and split into `a` and `b`. LeakyReLU activation is applied to `b`, and the element-wise product of `a` and `b` is returned. Unlike the Gated Linear Unit, LeakyReLU is used as the activation function.

Appendix E.

Geometry Mapping

E.1. Geometry Latent Mapping

In Ref. [96], a learnable mapping between an arbitrary geometry and a fixed cylindrical grid, called "Geometry Latent Mapping" (GLaM), is introduced. For the irregular calorimeter (with *real* cells), a regular calorimeter is constructed in the latent space (with *latent* cells) on which the generative model operates.

Embedding Matrix and Initialization

The mapping from the real cells to latent cells and vice versa is provided by a trainable *embedding* matrix and an *inverse embedding* matrix. The two matrices are optimized independently. Each element of the embedding matrix, which connects a real cell to a latent cell, is initialized based on the overlapping area between the real cell and the latent cell. The inverse matrix is initialized as the pseudoinverse of the embedding matrix.

Neighborhood Relations One desired property of such a mapping is that the neighborhood relations are preserved, i.e., latent cells are neighboring if, and only if, they represent two neighboring real cells or parts of the same real cell, and vice versa for real cells. While the matrices can, in principle, connect any real cell with any latent cell, this initialization strongly biases the mappings toward those that preserve neighborhood relations, at least approximately.

Training

The provided diffusion model is trained by first applying noise to the irregular calorimeter image, embedding the noised result in the regular latent space, evaluating the model, and then mapping the result back to the irregular calorimeter space. The loss is calculated for this irregular calorimeter image. This process allows the embedding/inverse matrices and the model to be optimized simultaneously. Both the convolution operations and this initialization provide a useful inductive bias to the model. Because the mapping preserves the neighborhood relations of the cells, the convolutions can exploit the underlying translation invariance of the data.

Application to CaloChallenge Dataset 1

In the CaloChallenge dataset 1 (Chapter 10), the calorimeter is cylindrical, and the cells are divided based on cylindrical coordinates. The calorimeter would be completely regular

if not for some radial and angular bins being merged in certain layers.

The embedding is performed independently for each calorimeter layer, reducing the size of the embedding matrices. The regular latent space is defined by combining the maximum resolutions of each coordinate. In this way, GLaM acts as a learnable version of the "Superfine Grid" approach.

Through this mapping, the diffusion model produces state-of-the-art results [147, 177].

Challenges & and Applicability to Highly Granular Calorimeters

In this use case, the calorimeter geometry can be regularized simply by splitting cells, which likely benefits the GLaM approach. However, more extensive modifications are typically required, and the applicability of this method to an irregular, highly granular calorimeter has yet to be demonstrated. Such a calorimeter may have an increased number of cells, gaps, and cells of varying shapes and sizes. Additionally, the cell geometry may differ between parts of the calorimeter, as in the CMS HGCAL (Section 2.2.3), where hexagonal silicon tiles are combined with grid-like scintillator tiles. For layers with many cells, the matrix could become large and difficult to train. The authors suggest that layers could be split into multiple blocks of cells, each mapped to the regular latent space independently, thus replacing one large matrix with many smaller matrices. While blocks of cells can be mapped individually, they still need to be combined into a regular, dense shape in the latent space.

Furthermore, it is likely advantageous if the latent space shares the same geometric structure as the real calorimeter. For calorimeters with cells having different numbers of neighbors, such as the CMS HGCAL with its hexagonal silicon tiles (6 neighbors) and grid-like scintillator tiles (4 neighbors), no regular latent space can fully replicate this structure.

E.2. Vector Quantization

The vector-quantized VAE (VQVAE) [97] introduces a method to discretize the latent space of a VAE. In addition to the encoder and decoder, it features a codebook, which is a list of trainable vectors. First, the input is passed through the encoder, producing a set of vectors in a continuous latent space. Each of these encoded vectors is then replaced by the closest vector from the codebook, a process known as quantization. The closeness is determined by the norm of the distance between the vectors. The quantized vectors are subsequently passed through the decoder to produce the output. This quantization would ordinarily break the gradient flow to the encoder. To avoid this, the gradient is redirected from the quantized vectors to the encoded vectors during backpropagation. In this way, the reconstruction loss influences both the encoder and the codebook vectors. The codebook vectors are optimized to match the encoded vectors, and vice versa. To generate samples, vectors are sampled from the codebook and passed to the decoder.

Calo-VQ

In contrast, in the Calo-VQ model [94], after training the VQVAE, the codebook vectors are interpreted as tokens and serve as the input sequence for a transformer (Section 3.5). The transformer is trained to predict the next token in the sequence and can then be used to iteratively produce a sequence of tokens that is subsequently passed to the decoder.

This approach allows a particle shower to be treated as a sequence of tokens, enabling the use of state-of-the-art models from natural language processing. However, the encoder and decoder rely on dense/convolutional layers and therefore face the same issues and restrictions regarding their applicability to irregular calorimeters.

Appendix F.

Calculation of the Gradients of an FFN

An FFN is a sequence of n linear layers (or "dense/hidden" layers) and a nonlinear, piecewise differentiable activation function $\sigma : \mathbb{R} \to \mathbb{R}$. A layer l maps the activation of the previous layer $\mathbf{a}^{l-1} \in \mathbb{R}^{m_{l-1}}$ using the matrix $\mathbf{W} \in \mathbb{R}^{m_l \times m_{l-1}}$ and adds the bias $\mathbf{b} \in \mathbb{R}^{m_l}$:

$$f_i^l = \sum_j W_{i,j}^l a_j^{l-1} + b_i^l \qquad \mathbf{f}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l.$$

For the first layer, $\mathbf{a}^0$ is the input vector $\mathbf{x}$. After each linear mapping, the activation function is applied elementwise to the resulting vector to produce the activation $\mathbf{a}^l$:

$$a_i^l = \sigma\left(f_i^l\right) \qquad \mathbf{a}^l = \sigma\left(\mathbf{f}^l\right).$$

The derivative of layer $\mathbf{f}^l$ with respect to $\mathbf{x}$, $\mathbf{b}$, and $\mathbf{W}$ can be calculated as

$$\frac{\partial f_i^l}{\partial a_j^{l-1}} = W_{i,j} \qquad \frac{\partial \mathbf{f}^l}{\partial \mathbf{a}^{l-1}} = \mathbf{W}^l$$
$$\frac{\partial f_i^l}{\partial b_k^l} = \delta_{i,k} \qquad \frac{\partial \mathbf{f}^l}{\partial \mathbf{b}^l} = \mathbf{1}_n$$
$$\frac{\partial f_i^l}{\partial W_{k,j}^l} = \delta_{i,k} a_j, \qquad \frac{\partial \mathbf{f}^l}{\partial \mathbf{W}^l} = \mathbf{1}_n \otimes \mathbf{a}^{l-1},$$

where $i, k \in \{1, ..., m_l\}$, $j \in \{1, ..., m_{l-1}\}$. $\mathbf{1}_n$ is the identity matrix with dimensions $n \times n$, and δ is the Kronecker delta:

$$(\mathbf{1})_{i,j} = \delta_{i,j} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}.$$

Furthermore, the derivate of the activation $\mathbf{a}^l$ with respect to the linear mapping $\mathbf{f}^l$ is

$$\frac{\partial a_i^l}{\partial f_k^l} = \delta_{i,k}\sigma'(f_k^l)$$

$$\frac{\partial \mathbf{a}^l}{\partial \mathbf{f}^l} = \operatorname{diag}\left(\sigma'(\mathbf{f}^l)\right), \qquad \text{where } \operatorname{diag}(\mathbf{v}) \text{ is the square matrix with the elements of vector } \mathbf{v} \text{ on the diagonal: } \operatorname{diag}(\mathbf{v}) = \mathbf{1}_{m_l}\mathbf{v}.$$

An FFN is constructed by chaining linear layers and activations: $\mathbf{a}^n \circ \mathbf{f}^n \circ \mathbf{a}^{n-1} \circ \mathbf{f}^{n-1} \circ \mathbf{a}^{n-2} \circ \ldots$ To propagate the gradient from one layer to the previous layer, the activation of a layer with respect to the activation of the previous layer must be computed:

$$\begin{aligned} \frac{\partial a_i^l}{\partial a_j^{l-1}} &= \sum_r \frac{\partial a_i^l}{\partial f_r^l} && \frac{\partial f_r^l}{\partial a_j^{l-1}} \\ &= \sum_r \delta_{r,i}\sigma'(f_r^{l-1}) && W_{r,j}^l \\ &= \sigma'(f_i^{l-1}) && W_{i,j}^l. \end{aligned}$$

$$\begin{aligned} \frac{\partial \mathbf{a}^l}{\partial \mathbf{a}^{l-1}} &= \frac{\partial \mathbf{a}^l}{\partial \mathbf{f}^l} && \frac{\partial \mathbf{f}^l}{\partial \mathbf{a}^{l-1}} \\ &= \operatorname{diag}\left(\sigma'(\mathbf{f}^{l-1})\right) && \mathbf{W}^l. \end{aligned}$$

By applying this rule iteratively from one layer to the next, the derivative can be calculated between any layers $\tilde{l} > l$:

$$\frac{\partial \mathbf{a}^{\tilde{l}}}{\partial \mathbf{a}^l} = \frac{\partial \mathbf{a}^{\tilde{l}}}{\partial \mathbf{a}^{\tilde{l}-1}} \cdot \ldots \cdot \frac{\partial \mathbf{a}^{l+1}}{\partial \mathbf{a}^l} = \prod_{u=0}^{\tilde{l}-l+1} \frac{\partial \mathbf{a}^{\tilde{l}-u}}{\partial \mathbf{a}^{\tilde{l}-u-1}} = \prod_{u=0}^{\tilde{l}-l+1} \operatorname{diag}\left(\sigma'(\mathbf{f}^{\tilde{l}-u-1})\right)\mathbf{W}^{\tilde{l}-u}$$

Thus, the gradients of the loss L with respect to the weights in layer l for an FFN with n layers can be computed as:

$$\begin{aligned} \nabla_{\mathbf{W}^l} L = \frac{\partial L}{\partial \mathbf{W}^l} &= \frac{\partial L}{\partial \mathbf{a}^l}\frac{\partial \mathbf{a}^{\tilde{n}}}{\partial \mathbf{a}^{\tilde{n}-1}} \ldots \frac{\partial \mathbf{a}^{l+1}}{\partial \mathbf{a}^l}\frac{\partial \mathbf{a}^l}{\partial \mathbf{W}^l} = \frac{\partial L}{\partial \mathbf{a}^n}\left(\prod_{u=0}^{n-l+1} \frac{\partial \mathbf{a}^{n-u}}{\partial \mathbf{a}^{n-u-1}}\right)\frac{\partial \mathbf{a}^l}{\partial \mathbf{f}^l}\frac{\partial \mathbf{f}^l}{\partial \mathbf{W}^l} \\ &= \frac{\partial L}{\partial \mathbf{a}^n}\left(\prod_{u=0}^{n-l+1} \operatorname{diag}\left(\sigma'(\mathbf{f}^{n-u-1})\right)\mathbf{W}^{n-u}\right)\operatorname{diag}\left(\sigma'(\mathbf{f}^l)\right) \otimes \mathbf{x}, \end{aligned}$$

and for the bias as

$$\nabla_{\mathbf{b}^l} L = \frac{\partial L}{\partial \mathbf{b}^l} = \frac{\partial L}{\partial \mathbf{a}^n}\left(\prod_{u=0}^{n-l+1} \operatorname{diag}\left(\sigma'(\mathbf{f}^{n-u-1})\right)\mathbf{W}^{n-u}\right)\operatorname{diag}\left(\sigma'(\mathbf{f}^l)\right).$$

The $\mathbf{f}^l$ values are saved during the forward pass. In the backward pass, the $\operatorname{diag}\left(\sigma'(\mathbf{f}^{n-u-1})\right)$ and $\mathbf{W}^{n-u}$ matrices are iteratively multiplied to calculate the derivatives from the last layer to the first. This backpropagation with matrix operations provides an efficient method to optimize large networks with many parameters.

Appendix G.

Public Contributions

Papers

- B. Kaech et al. "JetFlow: Generating Jets with Conditioned and Mass Constrained Normalising Flows", and ACAT 2022, 23-28 Oct. 2022, Bari, accepted, to be published in IOPscience in the Journal Of Physics: Conference Series (poster and paper)
  arXiv: 2211.13630

- S. Schnake et al. "CaloPointFlow - Generating Calorimeter Showers as Point Clouds", ACAT 2022, 23-28 Oct. Bari, accepted, to be published in IOPscience in the Journal Of Physics: Conference Series (poster and paper)

- M. Scham et al. "DeepTreeGAN: Fast Generation of High Dimensional Point Clouds", 26th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2023), conference in Norfolk, VA, 8-12 May 2023, (poster and paper), published in EPJ Web Conf. 295 (2024) 09010
  DOI: 10.1051/epjconf/202429509010

- M. Scham et al. "DeepTreeGANv2: Iterative Pooling of Point Clouds", at "Machine Learning and the Physical Sciences workshop" at the 37th Conference on Neural Information Processing Systems (NeurIPS), 15 Dec. 2023 (poster and paper)
  arXiv: 2312.00042

- C. Krause et. al., "CaloChallenge 2022: A Community Challenge for Fast Calorimeter Simulation", to be published

Software Packages

- M. Scham et al. "caloutils - Metrics and tools for evaluation of generative models for calorimeter showers based on pytorch_geometric", 28 July 2023
  URL: pypi.org/project/caloutils/

- M. Scham et al. "Extracting the Geometry of Cells in the HGCAL from CMSSW", 12 Oct. 2021, CMS internal only
  URL: github.com/DeGeSim/cms_geo_extractor/

Talks

- M. Scham et al. "Fast simulation of the CMS HGCal with Generative Models", Annual meeting, (BMBF Forschungsschwerpunkt) FSP CMS Workshop, 22-24 Sep. 2021
- M. Scham et al. "Fast simulation of the CMS HGCal with Generative Models", 14th Annual Meeting of the Helmholtz Alliance "Physics at the Terascale", 23-24 Nov. 2021
- M. Scham on behalf of the CMS collaboration, "Preprocessing for GNNs on HGCAL calorimeter showers", LPCC (LHC Physics Centre at CERN) Fast Detector Simulation Workshop, 22-23 Nov. 2021 (invited talk)
- M. Scham et al. "Generative Modeling with Graph Neural Networks for the CMS HGCal", FastSim Days 2022, Workshop at the LPC, virtual, 11-12 Oct. 2022
- M. Scham et al. "Generative modeling with Graph Neural Networks for the CMS HGCal", Annual meeting, (BMBF Forschungsschwerpunkt) FSP CMS Workshop, 28-30 Sep. 2022
- M. Scham et al. "DeepTreeGAN: Fast Generation of High Dimensional Point Clouds", Helmholtz AI conference, Hamburg, 12-14 June 2023
- M. Scham et al. "DeepTreeGAN: Fast Generation of High Dimensional Point Clouds", ML4Jets Workshop, Hamburg, 6–10 Nov. 2023
- M. Scham et al. "caloutils - Utilities and Metrics for Generative Models of Calorimeter Showers", ML4Jets Workshop, Hamburg, 6–10 Nov. 2023
- B. Käch et al. "Attention to Mean Fields for Particle Cloud Generation", ML4Jets Workshop, Hamburg, 6–10 Nov. 2023
- S. Schnake et al. "CaloPointFlow - Generating Calorimeter Showers as Point Clouds", ML4Jets Workshop, Hamburg, 6–10 Nov. 2023
- M. Scham et al. "DeepTreeGANv2: Iterative Pooling of Point Clouds", 6th Inter-experiment Machine Learning Workshop (IML), 02. Feb. 2024 (talk and poster)

Posters

- M. Scham et al. Poster presenting the DeGeSim Projects, Helmholtz AI virtual conference, 14-15 April 2021
- M. Scham, "Generative modeling with Graph Neural Networks for the CMS HGCal", CDCS Eröffnungssymposium, Hamburg, 26-28 April 2022
- M. Scham et al. Poster presenting the DeGeSim Projects, at Helmholtz AI conference, Dresden, 2-3 June 2022
- M. Scham et al. "DeepTreeGAN: Fast Generation of High Dimensional Point Clouds", Hammers & Nails 2023-Swiss Edition, Ascona, 29 Oct. to 3 Nov. 2023 (poster)
- B. Käch et al. "Pay Attention to Mean Fields for Point Cloud Generation", Hammers & Nails 2023-Swiss Edition, Ascona, 29 Oct. to 3 Nov. 2023 (poster)
- S. Schnake et al. "The Calorimeter Pyramid: Rethinking the design of generative calorimeter shower models", Hammers& Nails 2023-Swiss Edition, Ascona, 29 Oct. to 3 Nov. 2023 (poster)

My work has been accepted by the CMS collaboration as relevant for the experiment, I have become a CMS author and appear on all publications of the CMS collaboration since 2021 (201 publications).

Bibliography

[1] GEANT4 Collaboration. "GEANT4–a simulation toolkit". *Nucl. Instrum. Meth. A* 506 (2003), p. 250.
DOI: 10.1016/S0168-9002(03)01368-8.

[2] HEP Software Foundation Collaboration. "A Roadmap for HEP Software and Computing R&D for the 2020s". *Computing and Software for Big Science* 3.1 (Mar. 2019), p. 7.
DOI: 10.1007/s41781-018-0018-8.

[3] Roland Jansky. "The ATLAS Fast Monte Carlo Production Chain Project". *J. Phys. Conf. Ser.* 664.7 (2015), p. 072024.
DOI: 10.1088/1742-6596/664/7/072024.

[4] G. Apollinari et al. "High Luminosity Large Hadron Collider HL-LHC". *CERN Yellow Report* (2015), pp. 1–19.
DOI: 10.5170/CERN-2015-005.1.

[5] CMS Collaboration. "CMS Phase-2 Computing Model: Update Document". Tech. rep. Geneva: CERN, 2022.
URL: cds.cern.ch/record/2815292. Visited 04/30/2024.

[6] "The CMS HGCAL detector for HL-LHC upgrade". *5th Large Hadron Collider Physics Conference*. Aug. 2017.
arXiv: physics.ins-det/1708.08234.

[7] Ian Goodfellow et al. "Generative Adversarial Networks". Tech. rep. 11. New York, NY, USA, June 2014, p. 139.
DOI: 10.1145/3422622.

[8] Diederik P. Kingma and Max Welling. "Auto-Encoding Variational Bayes" (2013).
arXiv: stat.ML/1312.6114.

[9] Luke de Oliveira, Michela Paganini, and Benjamin Nachman. *Comput. Software Big Sci.* 4.1 (Sept. 2017).
DOI: 10.1007/s41781-017-0004-6.

[10] Michela Paganini, Luke de Oliveira, and Benjamin Nachman. "Accelerating Science with Generative Adversarial Networks: An Application to 3D Particle Showers in Multilayer Calorimeters". *Phys. Rev. Lett.* 120.4 (2018), p. 042003.
DOI: 10.1103/PhysRevLett.120.042003.

[11] Michela Paganini, Luke de Oliveira, and Benjamin Nachman. "CaloGAN: Simulating 3D high energy particle showers in multilayer electromagnetic calorimeters with generative adversarial networks". *Phys. Rev. D* 97.1 (2018), p. 014021.
DOI: 10.1103/PhysRevD.97.014021.

[12] Martin Erdmann et al. "Generating and refining particle detector simulations using the Wasserstein distance in adversarial networks". *Comput. Softw. Big Sci.* 2.1 (2018), p. 4.
DOI: 10.1007/s41781-018-0008-x.

[13] Martin Erdmann, Jonas Glombitza, and Thorben Quast. "Precise simulation of electromagnetic calorimeter showers using a Wasserstein Generative Adversarial Network". *Comput. Softw. Big Sci.* 3.1 (2019), p. 4.
DOI: 10.1007/s41781-018-0019-7.

[14] F. Carminati et al. "Three dimensional Generative Adversarial Networks for fast simulation". *J. Phys. Conf. Ser.* 1085.3 (2018), p. 032016.
DOI: 10.1088/1742-6596/1085/3/032016.

[15] Atlas Collaboration. "AtlFast3: The Next Generation of Fast Simulation in ATLAS". *Computing and Software for Big Science* 6.1 (Mar. 2022), p. 7. ISSN: 2510-2044.
DOI: 10.1007/s41781-021-00079-7.

[16] Raghav Kansal et al. *JetNet*. Version 2. Datasets containing jets with 30 constituents. Aug. 2022.
DOI: 10.5281/zenodo.6975118.

[17] Raghav and others Kansal. *JetNet150*. Version 2.0.0. Datasets containing jets with 150 constituents. Aug. 2022.
DOI: 10.5281/zenodo.6975117.

[18] Raghav Kansal. *JetNet library for machine learning in high energy physics*. Sept. 2022.
DOI: 10.5281/zenodo.7140150.

[19] C. W. Fabjan and F. Gianotti. "Calorimetry for particle physics". *Rev. Mod. Phys.* 75 (Oct. 2003), p. 1243.
DOI: 10.1103/RevModPhys.75.1243.

[20] Olin L. Pinto. "Shower Shapes in a Highly Granular SiPM-on-Tile Analog Hadron Calorimeter". PhD thesis. Staats-und Universitätsbibliothek Hamburg Carl von Ossietzky, 2022.

[21] "Review of Particle Physics". *Phys. Rev. D* 98 (3 Aug. 2018), p. 030001.
DOI: 10.1103/PhysRevD.98.030001.

[22] Buhmann, Erik et al. "Fast and Accurate Electromagnetic and Hadronic Showers from Generative Models". *EPJ Web Conf.* 251 (2021), p. 03049.
DOI: 10.1051/epjconf/202125103049.

[23] "The CMS Electromagnetic Calorimeter: overview, lessons learned during Run 1 and future projections". *Journal of Physics: Conference Series* 587 (Feb. 2015), p. 012001.
DOI: 10.1088/1742-6596/587/1/012001.

[24] M. Krammer. "Silicon detectors in High Energy Physics experiments". *Scholarpedia* 10.10 (2015). revision #197033, p. 32486.
DOI: 10.4249/scholarpedia.32486.

[25] CMS Collaboration. “The Phase-2 Upgrade of the CMS Endcap Calorimeter”. Tech. rep. Geneva: CERN, 2017.
DOI: 10.17181/CERN.IV8M.1JY2.

[26] CMS Collaboration. *High-Luminosity Large Hadron Collider (HL-LHC) Preliminary Design Report*. en. 2015.
DOI: 10.5170/CERN-2015-005.

[27] Simone Marzani, Gregory Soyez, and Michael Spannowsky. “Looking Inside Jets: An Introduction to Jet Substructure and Boosted-object Phenomenology”. Springer International Publishing, 2019. ISBN: 9783030157098.
DOI: 10.1007/978-3-030-15709-8.

[28] Matteo Cacciari, Gavin P. Salam, and Gregory Soyez. “The anti-k_t jet clustering algorithm”. *JHEP* 04 (2008), p. 063.
DOI: 10.1088/1126-6708/2008/04/063.

[29] *GPT-4*.
URL: openai.com/index/gpt-4-research. Visited 05/02/2024.

[30] *DALL·E 3*.
URL: openai.com/index/dall-e-3. Visited 05/02/2024.

[31] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080.
DOI: https://doi.org/10.1016/0893-6080(89)90020-8.

[32] Tom M. Mitchell. “The Need for Biases in Learning Generalizations”. Rutgers CS tech report CBM-TR-117. 1980.
URL: www.cs.cmu.edu/~tom/pubs/NeedForBias_1980.pdf. Visited 07/24/2024.

[33] Peter W. Battaglia et al. *Relational inductive biases, deep learning, and graph networks*. 2018.
arXiv: 1806.01261.

[34] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. “Deep Learning”. MIT Press, 2016. ISBN: 9780262035613.
URL: www.deeplearningbook.org. Visited 04/30/2024.

[35] John S. Bridle. “Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition”. *Neurocomputing*. Springer Berlin Heidelberg, 1990, pp. 227–236. ISBN: 9783642761539.
DOI: 10.1007/978-3-642-76153-9_28.

[36] Shiv R. Dubey, Satish K. Singh, and Bidyut B. Chaudhuri. *Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark*. 2022.
arXiv: cs.LG/2109.14545.

[37] Papers with Code. *An Overview of Activation Functions*. en. June 27, 2024.
URL: paperswithcode.com/methods/category/activation-functions. Visited 06/27/2024.

[38] Vinod Nair and Geoffrey E. Hinton. "Rectified linear units improve restricted boltzmann machines". *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10. Haifa, Israel: Omnipress, 2010, pp. 807–814. ISBN: 9781605589077.

[39] Dan Hendrycks and Kevin Gimpel. "Gaussian Error Linear Units (GELUs)". Tech. rep. June 2023.
arXiv: cs/1606.08415.

[40] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 1476-4687.
DOI: 10.1038/323533a0.

[41] Andrew L. Maas et al. "Rectifier nonlinearities improve neural network acoustic models". *Proc. icml*. Vol. 30. 1. Citeseer. 2013, p. 3.
URL: robotics.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf. Visited 04/30/2024.

[42] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee W. Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, May 2010, pp. 249–256.
URL: proceedings.mlr.press/v9/glorot10a.html. Visited 04/30/2024.

[43] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015.
arXiv: cs.CV/1512.03385.

[44] Adam and others Paszke. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Ed. by H. Wallach et al. Version 2.0.1. 2019.
arXiv: 1912.01703.

[45] Christopher M. Bishop. "Pattern Recognition and Machine Learning". Corrected at 8th printing. Information science and statistics. OCLC: 1270244228. New York: Springer, 2009. ISBN: 9780387310732.
URL: link.springer.com/book/9780387310732. Visited 11/29/2023.

[46] Sergios Theodoridis. "Chapter 8 - Parameter Learning: a Convex Analytic Path". *Machine Learning (Second Edition)*. Ed. by Sergios Theodoridis. Second Edition. Academic Press, 2020, pp. 351–425. ISBN: 978-0-12-818803-3.
DOI: https://doi.org/10.1016/B978-0-12-818803-3.00017-9.

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

[48] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". *Journal of Machine Learning Research* 15.56 (2014), p. 1929.
URL: jmlr.org/papers/v15/srivastava14a.html. Visited 04/30/2024.

[49] Ilya Loshchilov and Frank Hutter. *Decoupled Weight Decay Regularization*. 2019.
arXiv: cs.LG/1711.05101.

[50] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. Ed. by Francis Bach and David Blei. Lille, France, July 2015.
arXiv: cs.LG/1502.03167.

[51] Jimmy L. Ba, Jamie R. Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016.
arXiv: stat.ML/1607.06450.

[52] Tianle Cai et al. "GraphNorm: A Principled Approach to Accelerating Graph Neural Network Training". Tech. rep. arXiv:2009.03294 [cs, math, stat] type: article. arXiv, June 2021.
arXiv: 2009.03294.

[53] Tim Salimans et al. *Improved Techniques for Training GANs*. Ed. by D. Lee et al. 2016.
arXiv: cs.LG/1606.03498.

[54] Yuichi Yoshida and Takeru Miyato. "Spectral Norm Regularization for Improving the Generalizability of Deep Learning". Tech. rep. May 2017.
arXiv: cs,stat/1705.10941.

[55] Sitao Xiang and Hao Li. *On the Effects of Batch and Weight Normalization in Generative Adversarial Networks*. 2017.
arXiv: stat.ML/1704.03971.

[56] Takeru Miyato et al. "Spectral Normalization for Generative Adversarial Networks". *6th International Conference on Learning Representations*. 2018.
arXiv: cs.LG/1802.05957.

[57] Danilo Rezende and Shakir Mohamed. "Variational Inference with Normalizing Flows". *Proceedings of the 32nd International Conference on Machine Learning*. Vol. 37. Proceedings of Machine Learning Research. PMLR, 2015, pp. 1530–1538.
arXiv: stat.ML/1505.05770.

[58] Martin Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein generative adversarial networks". *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia: JMLR.org, 2017, pp. 214–223.

[59] Cédric Villani. "The Wasserstein distances". *Optimal Transport: Old and New*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, p. 93. ISBN: 9783540710509.
DOI: 10.1007/978-3-540-71050-9_6.

[60] Ishaan Gulrajani et al. "Improved Training of Wasserstein GANs". *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, p. 5769. ISBN: 9781510860964.
arXiv: 1704.00028.

[61] Xudong Mao et al. "Least Squares Generative Adversarial Networks". Tech. rep. 2016.
arXiv: cs/1611.04076.

[62] Jae H. Lim and Jong C. Ye. *Geometric GAN*. 2017.
arXiv: stat.ML/1705.02894.

[63] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks". Tech. rep. Jan. 2016.
arXiv: cs/1511.06434.

[64] Tero Karras, Samuli Laine, and Timo Aila. "A Style-Based Generator Architecture for Generative Adversarial Networks". Tech. rep. Mar. 2019.
arXiv: cs,stat/1812.04948.

[65] Jun-Yan Zhu et al. "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks". Tech. rep. Aug. 2020.
arXiv: cs/1703.10593.

[66] Yann LeCun and Corinna Cortes. *MNIST handwritten digit database*. 2010.
URL: yann.lecun.com/exdb/mnist/. Visited 04/30/2024.

[67] Martin Arjovsky and Léon Bottou. "Towards Principled Methods for Training Generative Adversarial Networks". Tech. rep. Jan. 2017.
arXiv: cs,stat/1701.04862.

[68] Google. *Common Problems | Machine Learning*. July 18, 2022.
URL: developers . google . com / machine - learning / gan / problems. Visited 01/15/2024.

[69] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". *IEEE Transactions on Neural Networks* 5.2 (Mar. 1994), pp. 157–166. ISSN: 1941-0093.
DOI: 10.1109/72.279181.

[70] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training Recurrent Neural Networks". Tech. rep. Feb. 2013.
arXiv: cs/1211.5063.

[71] Ashish Vaswani et al. "Attention Is All You Need". *CoRR* abs/1706.03762 (2017).
arXiv: 1706.03762.

[72] William L. Hamilton. "Graph Representation Learning". Vol. 14. 3. Morgan and Claypool, Sept. 16, 2022, pp. 1–159.

[73] Evelyn Fix and J. L. Hodges. "Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties". *International Statistical Review / Revue Internationale de Statistique* 57.3 (Dec. 1989), p. 238. ISSN: 0306-7734.
DOI: 10.2307/1403797.

[74] Masashi Tsubaki, Kentaro Tomii, and Jun Sese. "Compound-protein interaction prediction with end-to-end learning of neural networks for graphs and sequences". *Bioinformatics* 35.2 (July 2018), pp. 309–318. ISSN: 1367-4803.
DOI: 10.1093/bioinformatics/bty535.

[75] Wayne W. Zachary. "An Information Flow Model for Conflict and Fission in Small Groups". *Journal of Anthropological Research* 33.4 (Dec. 1977), pp. 452–473. ISSN: 2153-3806.
DOI: 10.1086/jar.33.4.3629752.

[76] Yann Lecun et al. “Gradient-based learning applied to document recognition”. *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.
DOI: 10.1109/5.726791.

[77] Matthias Fey and Jan E. Lenssen. “Fast Graph Representation Learning with PyTorch Geometric”. 2019.
arXiv: cs.LG/1903.02428.

[78] Justin Gilmer et al. “Neural Message Passing for Quantum Chemistry”. *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee W. Teh. Vol. 70. PMLR, 2017, p. 1263.
arXiv: cs.LG/1704.01212.

[79] Y. LeCun et al. “Backpropagation Applied to Handwritten Zip Code Recognition”. *Neural Computation* 1.4 (Dec. 1989), pp. 541–551. ISSN: 0899-7667.
DOI: 10.1162/neco.1989.1.4.541.

[80] Valerio Biscione and Jeffrey S. Bowers. *Convolutional Neural Networks Are Not Invariant to Translation, but They Can Learn to Be*. 2021.
arXiv: cs.CV/2110.05861.

[81] Joan Bruna et al. *Spectral Networks and Locally Connected Networks on Graphs*. 2014.
arXiv: cs.LG/1312.6203.

[82] Thomas N. Kipf and Max Welling. “Semi-supervised classification with graph convolutional networks”. *IEEE Transactions on Neural Networks*. 5.1 (2016), pp. 61–80.
arXiv: 1609.02907.

[83] Keyulu Xu et al. “How Powerful are Graph Neural Networks?” *International Conference on Learning Representations*. 2019.
arXiv: cs.LG/1810.00826.

[84] Petar Veličković et al. *Graph Attention Networks*. 2018.
arXiv: stat.ML/1710.10903.

[85] Shaked Brody, Uri Alon, and Eran Yahav. “How Attentive are Graph Attention Networks?” *International Conference on Learning Representations*. 2022.
arXiv: 2105.14491.

[86] Sascha Diefenbacher et al. “DCTRGAN: Improving the Precision of Generative Models with Reweighting”. *JINST* 15.11 (2020), P11004.
DOI: 10.1088/1748-0221/15/11/P11004.

[87] Sascha Diefenbacher et al. “L2LFlows: Generating High-Fidelity 3D Calorimeter Images” (2023).
DOI: 10.1088/1748-0221/18/10/p10017.

[88] Sascha Diefenbacher et al. “New Angles on Fast Calorimeter Shower Simulation” (Mar. 2023).
DOI: 10.1088/2632-2153/acefa9.

[89] ATLAS Collaboration. *Fast simulation of the ATLAS calorimeter system with Generative Adversarial Networks*. ATL-SOFT-PUB-2020-006. 2020.
URL: cds.cern.ch/record/2746032.

[90] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021.
arXiv: 2010.11929.

[91] Salman Khan et al. "Transformers in Vision: A Survey". *ACM Computing Surveys* 54.10s (Jan. 2022), pp. 1–41. ISSN: 1557-7341.
DOI: 10.1145/3505244.

[92] Renato Cardoso et al. *CaloDiT: Diffusion with transformers for fast shower simulation*. Stony New York Brook, Mar. 9, 2024.
URL: indico.cern.ch/event/1330797/contributions/5796591/.

[93] Dalila Salamani. *Fast Calorimeter Simulation in ATLASwith DNNs*. ML4Jets. New York, Jan. 15, 2020.
URL: cds.cern.ch/record/2706209/files/ATL-SOFT-SLIDE-2020-009.pdf. Visited 06/13/2024.

[94] Qibin Liu et al. "Calo-VQ: Vector-Quantized Two-Stage Generative Model in Calorimeter Simulation" (May 2024).
arXiv: physics.ins-det/2405.06605.

[95] Dmitrii Kobylianskii et al. *CaloGraph: Graph-based diffusion model for fast shower generation in calorimeters with irregular geometry*. 2024.
arXiv: hep-ex/2402.11575.

[96] Oz Amram and Kevin Pedro. "Denoising diffusion models with geometry adaptation for high fidelity calorimeter simulation". *Phys. Rev. D* 108.7 (2023), p. 072014.
DOI: 10.1103/PhysRevD.108.072014.

[97] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. *Neural Discrete Representation Learning*. 2018.
arXiv: 1711.00937.

[98] Angel X. Chang et al. "ShapeNet: An Information-Rich 3D Model Repository". Tech. rep. arXiv: [cs.GR]. Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015.
arXiv: 1512.03012.

[99] Raghav Kansal et al. "Particle Cloud Generation with Message Passing Generative Adversarial Networks" (2022). Supplemental available at https://proceedings.neurips.cc/paper/2021/hash/c8512d142a2d849725f31a9a7a361ab9-Abstract.html.
arXiv: cs.LG/2106.11535.

[100] Simon Schnake, Dirk Krücker, and Kerstin Borras. *Generating Calorimeter Showers as Point Clouds*. Machine Learning and the Physical Sciences, Workshop at the 36th conference on Neural Information Processing Systems (NeurIPS). Dec. 2022.
URL: ml4physicalsciences.github.io/2022/files/NeurIPS_ML4PS_2022_77.pdf.

[101] Benno Käch, Dirk Krücker, and Isabell Melzer-Pellmann. *Point Cloud Generation using Transformer Encoders and Normalising Flows*. 2022.
arXiv: hep-ex/2211.13623.

[102] Benno Käch and Isabell Melzer-Pellmann. *Attention to Mean-Fields for Particle Cloud Generation*. May 2023.
arXiv: hep-ex/2305.15254.

[103] Erik Buhmann, Gregor Kasieczka, and Jesse Thaler. "EPiC-GAN: Equivariant Point Cloud Generation for Particle Jets" (Jan. 2023).
DOI: 10.21468/scipostphys.15.4.130.

[104] Chulin Xie et al. *Style-based Point Generator with Adversarial Rendering for Point Cloud Completion*. 2021.
arXiv: cs.CV/2103.02535.

[105] Dong W. Shu, Sung W. Park, and Junseok Kwon. *3D Point Cloud Generative Adversarial Network Based on Tree Structured Graph Convolutions*. 2019.
arXiv: cs.CV/1905.06292.

[106] Moritz A. W. Scham et al. "DeepTreeGAN: Fast Generation of High Dimensional Point Clouds". 2023.
arXiv: hep-ex/2311.12616.

[107] Moritz A. W. Scham, Dirk Krücker, and Kerstin Borras. "DeepTreeGANv2: Iterative Pooling of Point Clouds" (Dec. 2023).
arXiv: hep-ex,physics:physics/2312.00042.

[108] Panos Achlioptas et al. "Learning Representations and Generative Models for 3D Point Clouds". *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. PMLR, June 2018, p. 40.
arXiv: cs/1707.02392.

[109] Diego Valsesia, Giulia Fracastoro, and Enrico Magli. "Learning Localized Generative Models for 3D Point Clouds via Graph Convolution". en. Sept. 2018.
URL: openreview.net/forum?id=SJeXSo09FQ. Visited 04/25/2024.

[110] Vinicius Mikuni, Benjamin Nachman, and Mariel Pettee. "Fast Point Cloud Generation with Diffusion Models in High Energy Physics" (Apr. 2023).
arXiv: hep-ph/2304.01266.

[111] Matthew Leigh et al. *PC-JeDi: Diffusion for Particle Cloud Generation in High Energy Physics*. 2023.
arXiv: hep-ph/2303.05376.

[112] Benno Käch et al. "Attention to Mean Fields for Particle Cloud Generation". ML4Jets. Oct. 10, 2023.
URL: indico.cern.ch/event/1253794/contributions/5588648/. Visited 06/18/2024.

[113] Shitong Luo and Wei Hu. *Diffusion Probabilistic Models for 3D Point Cloud Generation*. 2021.
arXiv: cs.CV/2103.01458.

[114] Edward Wagstaff et al. *Universal Approximation of Functions on Sets*. 2021.
arXiv: cs.LG/2107.01959.

[115] Hongyang Gao and Shuiwang Ji. "Graph U-Nets". Tech. rep. ICML19. arXiv, May 2019.
arXiv: 1905.05178.

[116] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. "Self-Attention Graph Pooling". Tech. rep. arXiv:1904.08082 [cs, stat] type: article. arXiv, June 2019.
arXiv: 1904.08082.

[117] Juho Lee et al. "Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks". Tech. rep. May 2019.
arXiv: cs,stat/1810.00825.

[118] J. Alwall et al. "The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations". *Journal of High Energy Physics* 2014.7 (July 2014), p. 79. ISSN: 1029-8479.
DOI: 10.1007/JHEP07(2014)079.

[119] Torbjorn Sjostrand, Stephen Mrenna, and Peter Z. Skands. "A Brief Introduction to PYTHIA 8.1". *Comput. Phys. Commun.* 178 (2008), p. 852.
DOI: 10.1016/j.cpc.2008.01.036.

[120] Matteo Cacciari, Gavin P. Salam, and Gregory Soyez. "FastJet User Manual". *Eur. Phys. J. C* 72 (2012), p. 1896.
DOI: 10.1140/epjc/s10052-012-1896-2.

[121] Raghav Kansal et al. "Evaluating generative models in high energy physics". *Physical Review D* 107.7 (Apr. 2023).
DOI: 10.1103/physrevd.107.076017.

[122] Alex Williams. *A short introduction to optimal transport and Wasserstein distance*. Oct. 2020.
URL: alexhwilliams . info / itsneuronalblog / 2020 / 10 / 09 / optimal - transport/. Visited 04/30/2024.

[123] Aaditya Ramdas, Nicolás G. Trillos, and Marco Cuturi. "On Wasserstein Two-Sample Testing and Related Families of Nonparametric Tests". *Entropy* 19.2 (2017). ISSN: 1099-4300.
DOI: 10.3390/e19020047.

[124] Jason Altschuler, Jonathan Weed, and Philippe Rigollet. "Near-linear time approximation algorithms for optimal transport via Sinkhorn iteration". Tech. rep. Feb. 2018.
arXiv: 1705.09634.

[125] Sergey G. Bobkov and Michel Ledoux. "One-dimensional empirical measures, order statistics, and Kantorovich transport distances". *Memoirs of the American Mathematical Society* (2019).
DOI: 10.1090/memo/1259.

[126] Patrick T. Komiske, Eric M. Metodiev, and Jesse Thaler. "Energy flow polynomials: A complete linear basis for jet substructure". *Journal of High Energy Physics* 04.4 (Apr. 2018), p. 013. ISSN: 1029-8479.
DOI: 10.1007/JHEP04(2018)013.

[127] Gavin P. Salam. "Towards jetography". *The European Physical Journal C* 67.3–4 (May 2010), pp. 637–686. ISSN: 1434-6052.
DOI: 10.1140/epjc/s10052-010-1314-6.

[128] Martin Heusel et al. “GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium”. Advances in Neural Information Processing Systems. Curran Associates, Inc., 2017.
arXiv: 1706.08500.

[129] Christian Szegedy et al. “Rethinking the Inception Architecture for Computer Vision”. Tech. rep. arXiv, Dec. 2015.
arXiv: 1512.00567.

[130] Huilin Qu and Loukas Gouskos. “ParticleNet: Jet Tagging via Particle Clouds”. *Physical Review D* 101.5 (Mar. 2020), p. 056019.
DOI: 10.1103/PhysRevD.101.056019.

[131] Yue Wang et al. “Dynamic Graph CNN for Learning on Point Clouds”. Tech. rep. June 2019.
arXiv: cs/1801.07829.

[132] Min J. Chong and David Forsyth. “Effectively Unbiased FID and Inception Score and where to find them”. Tech. rep. June 2020.
arXiv: cs/1911.07023.

[133] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[134] G. E. P. Box and D. R. Cox. “An Analysis of Transformations”. *Journal of the Royal Statistical Society. Series B (Methodological)* 26.2 (1964), pp. 211–252. ISSN: 00359246.
DOI: 10.1111/j.2517-6161.1964.tb00553.x.

[135] Charles R. Qi et al. “PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation”. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. July 2017.

[136] Leslie N. Smith. “Cyclical Learning Rates for Training Neural Networks”. Tech. rep. Apr. 2017.
arXiv: cs/1506.01186.

[137] Erik Buhmann et al. “EPiC-ly Fast Particle Cloud Generation with Flow-Matching and Diffusion”. Tech. rep. Sept. 2023.
arXiv: 2310.00049.

[138] Vinicius Mikuni and Benjamin Nachman. “Score-based generative models for calorimeter shower simulation”. *Physical Review D* 106.9 (Nov. 2022), p. 092009.
DOI: 10.1103/PhysRevD.106.092009.

[139] Michel Dekking. “A modern introduction to probability and statistics. Understandig why and how”. Ed. by Cornelis Kraaikamp, Hendrik P. Lopuhaä, and Ludolf E. Meester. Springer texts in statistics. London: Springer-Verlag London Limited, 2010. 487 pp. ISBN: 9781849969529.

[140] Günter Klambauer et al. “Self-Normalizing Neural Networks”. Tech. rep. Sept. 2017.
arXiv: cs,stat/1706.02515.

[141] Claudius Krause et al. *CaloChallenge 2022: A Community Challenge for Fast Calorimeter Simulation*. to be published. 2022.
arXiv: 2410.21611.

[142] Michele F. Giannelli et al. *Fast Calorimeter Simulation Challenge 2022 - Dataset 1*. Zenodo, June 2023.
DOI: 10.5281/zenodo.8099322.

[143] ATLAS collaboration. *FastCaloGAN Training Project*. Version 1.0. Oct. 2021.
DOI: 10.5281/zenodo.5589623.

[144] Faucci M. Giannelli et al. *Fast Calorimeter Simulation Challenge 2022 - Dataset 2*. Zenodo, Mar. 2022.
DOI: 10.5281/zenodo.6366271.

[145] Faucci M. Giannelli et al. *Fast Calorimeter Simulation Challenge 2022 - Dataset 3*. Zenodo, Mar. 2022.
DOI: 10.5281/zenodo.6366324.

[146] Moritz Scham et al. *DeepTreeGAN: Fast Generation of High Dimensional Point Clouds*. ML4Jets. Hamburg, Nov. 7, 2023.
URL: indico.cern.ch/event/1253794/contributions/5588597/. Visited 03/20/2024.

[147] Claudius Krause et al. *The Fast Calorimeter Simulation Challenge 2022*. ML4Jets. Hamburg, Nov. 9, 2023.
URL: indico.cern.ch/event/1253794/contributions/5588599/. Visited 03/20/2024.

[148] Moritz A. W. Scham and Benno Käch. *caloutils: Metrics and tools for evaluation of generative models for calorimeter showers based on pytorch_geometric*. v.0.0.18. July 28, 2023.
URL: github.com/DeGeSim/caloutils. Visited 04/30/2024.

[149] Faucci M. Giannelli and Rui Zhang. "CaloShowerGAN, a Generative Adversarial Networks model for fast calorimeter shower simulation" (Sept. 2023).
arXiv: physics.ins-det/2309.06515.

[150] Thorsten Buss et al. "Convolutional L2LFlows: Generating Accurate Showers in Highly Granular Calorimeters Using Convolutional Normalizing Flows" (May 2024).
arXiv: physics.ins-det/2405.20407.

[151] Claudius Krause, Ian Pang, and David Shih. "CaloFlow for CaloChallenge Dataset 1" (Oct. 2022).
arXiv: physics.ins-det/2210.14245.

[152] Matthew R. Buckley et al. "Inductive simulation of calorimeter showers with normalizing flows". *Phys. Rev. D* 109.3 (2024), p. 033006.
DOI: 10.1103/PhysRevD.109.033006.

[153] Florian Ernst et al. "Normalizing Flows for High-Dimensional Detector Simulations" (Dec. 2023).
arXiv: hep-ph/2312.09290.

[154] Ian Pang, John A. Raine, and David Shih. "SuperCalo: Calorimeter shower superresolution" (Aug. 2023).
arXiv: physics.ins-det/2308.11700.

[155] Simon Schnake, Dirk Krücker, and Kerstin Borras. *CaloPointFlow II Generating Calorimeter Showers as Point Clouds*. Mar. 2024.
arXiv: physics.ins-det/2403.15782.

[156] Erik Buhmann et al. "CaloClouds: fast geometry-independent highly-granular calorimeter simulation". *JINST* 18.11 (2023), P11025.
DOI: 10.1088/1748-0221/18/11/P11025.

[157] Erik Buhmann et al. "CaloClouds II: ultra-fast geometry-independent highly-granular calorimeter simulation". *JINST* 19.04 (2024), P04020.
DOI: 10.1088/1748-0221/19/04/P04020.

[158] Vinicius Mikuni and Benjamin Nachman. "CaloScore v2: single-shot calorimeter shower simulation with diffusion models". *JINST* 19.02 (2024), P02001.
DOI: 10.1088/1748-0221/19/02/P02001.

[159] Jesse C. Cresswell et al. "CaloMan: Fast generation of calorimeter showers with density estimation on learned manifolds". *NeurIPS Workshop on Machine Learning and the Physical Sciences*. 2022.
arXiv: hep-ph/2211.15380.

[160] Thandikire Madula and Vinicius M. Mikuni. "CaloLatent: Score-based Generative Modelling in the Latent Space for Calorimeter Shower Generation". *NeurIPS Workshop on Machine Learning and the Physical Sciences*. 2023.
URL: ml4physicalsciences.github.io/2023/files/NeurIPS_ML4PS_2023_19.pdf.

[161] Luigi Favaro et al. "CaloDREAM – Detector Response Emulation via Attentive flow Matching" (May 2024).
arXiv: hep-ph/2405.09629.

[162] Erik Buhmann et al. "Getting High: High Fidelity Simulation of High Granularity Calorimeters with High Speed". *Comput. Softw. Big Sci.* 5.1 (May 2020), p. 13.
DOI: 10.1007/s41781-021-00056-0.

[163] Erik Buhmann et al. "Decoding Photons: Physics in the Latent Space of a BIB-AE Generative Network". *EPJ Web Conf.* 251 (2021), p. 03003.
DOI: 10.1051/epjconf/202125103003.

[164] Claudius Krause and David Shih. "CaloFlow: Fast and Accurate Generation of Calorimeter Showers with Normalizing Flows" (June 2021).
arXiv: physics.ins-det/2106.05285.

[165] Claudius Krause and David Shih. "CaloFlow II: Even Faster and Still Accurate Generation of Calorimeter Showers with Normalizing Flows" (2021).
arXiv: physics.ins-det/2110.11377.

[166] Benno Käch et al. *JetFlow: Generating Jets with Conditioned and Mass Constrained Normalising Flows*. 2022.
arXiv: hep-ex/2211.13630.

[167] Joschka Birk et al. *Flow Matching Beyond Kinematics: Generating Jets with Particle-ID and Trajectory Displacement Information*. 2023.
arXiv: hep-ph/2312.00123.

[168] Joschka Birk, Anna Hallin, and Gregor Kasieczka. *OmniJet-α: The first cross-task foundation model for particle physics*. 2024.
arXiv: hep-ph/2403.05618.

[169] Moritz Scham and Soham Bhattacharya. *Extracting the Geometry of Cells in the HGCal from CMSSW*. Oct. 12, 2021.
URL: github.com/DeGeSim/cms_geo_extractor.

[170] Thea Aarrestad et al. *CMS ML Hackathon: FastSim*. CMS internal only. 2023.
URL: indico.cern.ch/event/1243650/.

[171] Moritz Scham and Benno Käch. *Dequantisation of Calorimeter Cells*. CMS ML Hackathon on FastSim. Feb. 10, 2023.
URL: indico.cern.ch/event/1243650/contributions/5226400.

[172] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". Tech. rep. May 2015.
arXiv: cs/1505.04597.

[173] Michael L. Waskom. "seaborn: statistical data visualization". *Journal of Open Source Software* 6.60 (2021), p. 3021.
DOI: 10.21105/joss.03021.

[174] Joram Soch et al. *The Book of Statistical Proofs*. Version 2023. Jan. 2024.
DOI: 10.5281/zenodo.10495684.

[175] Matthias Fey. *pytorch_scatter*. Version 2.1.2. Library with highly optimized sparse update operations for PyTorch. May 2024.
URL: github.com/rusty1s/pytorch_scatter. Visited 05/06/2024.

[176] Yann N. Dauphin et al. "Language Modeling with Gated Convolutional Networks". Tech. rep. Sept. 2017.
arXiv: 1612.08083.

[177] Oz Amram and Kevin Pedro. *CaloDiffusion with GLaM for HighFidelity Calorimeter Simulation*. ML4Jets. Nov. 7, 2023.
URL: indico . cern . ch / event / 1253794 / contributions / 5588571/. Visited 06/14/2024.

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to express my deep gratitude to my supervisor, **Dirk Krücker**. Thank you for your guidance, patience, and for giving me the freedom to occasionally run headfirst through a wall. You were right — it had to work *somehow*.

I am also grateful to **Kerstin Borras** for her supervision and guidance throughout my PhD. I appreciate **Gregor Kasieczka** for the valuable feedback he provided during our discussions. I would also like to thank **Michael Krämer** for stepping in as my second supervisor.

Benno, thank you for the insightful discussions and valuable ideas that undoubtedly contributed to the success of this thesis.

Soham, thank you for keeping my spirits up, for the much-needed mental breaks, for the support with CMSSW, and for your great company over the years. And, of course, for introducing me to Indian cuisine.

Ana, from the very beginning of our PhDs during COVID to the final stretch, you have been a constant. I'm grateful to have shared this journey with you.

Beatriz, you left Hamburg too soon — thank you for the time we shared.

Matthias, thank you for your insightful feedback and for patiently explaining physics whenever I needed it. More than that, I am grateful for your friendship. So long, and thanks for all the chilis.

Valle, thank you for being a true friend when I really needed one.

Grazie a **Nicola** per il ragù. Also for being a wonderful friend, but mostly for the ragù. Ricordi: la parte più difficile è spaccare la zucca.

Lucia & **Nayla**, thank you for making me a little more Latino with every hour we spent together — and for the chisme.

I am grateful to all the people at DESY for welcoming me and making this place feel like home. To the volleyball teams at DESY and Altona 93 — thank you for the endless fun, the great matches, and the stress relief.

Finally, to my **family** — thank you for your love and steadfast support through all the years of my education. I couldn't have done this without you.

Eidesstattliche Erklärung

Moritz Alfons Wilhelm Scham erklärt hiermit, dass diese Dissertation und die darin dargelegten Inhalte die eigenen sind und selbstständig, als Ergebnis der eigenen originären Forschung, generiert wurden.

Hiermit erkläre ich an Eides statt

1. Diese Arbeit wurde vollständig oder größtenteils in der Phase als Doktorand dieser Fakultät und Universität angefertigt;
2. Sofern irgendein Bestandteil dieser Dissertation zuvor für einen akademischen Abschluss oder eine andere Qualifikation an dieser oder einer anderen Institution verwendet wurde, wurde dies klar angezeigt;
3. Wenn immer andere eigene- oder Veröffentlichungen Dritter herangezogen wurden, wurden diese klar benannt;
4. Wenn aus anderen eigenen- oder Veröffentlichungen Dritter zitiert wurde, wurde stets die Quelle hierfür angegeben. Diese Dissertation ist vollständig meine eigene Arbeit, mit der Ausnahme solcher Zitate;
5. Alle wesentlichen Quellen von Unterstützung wurden benannt;
6. Wenn immer ein Teil dieser Dissertation auf der Zusammenarbeit mit anderen basiert, wurde von mir klar gekennzeichnet, was von anderen und was von mir selbst erarbeitet wurde;
7. Teile dieser Arbeit sind bereits in den folgenden Referenzen veröffentlicht worden:

[106] Moritz A. W. Scham et al. "DeepTreeGAN: Fast Generation of High Dimensional Point Clouds". 2023.
arXiv: hep-ex/2311.12616.

[107] Moritz A. W. Scham, Dirk Krücker, and Kerstin Borras. "DeepTreeGANv2: Iterative Pooling of Point Clouds" (Dec. 2023).
arXiv: hep-ex,physics:physics/2312.00042.

[141] Claudius Krause et al. *CaloChallenge 2022: A Community Challenge for Fast Calorimeter Simulation.* to be published. 2022.
arXiv: 2410.21611.

Hamburg, den 07.02.2025 Moritz Alfons Wilhelm Scham