Department of Physics and Astronomy
Heidelberg University

Bachelor Thesis in Physics
submitted by

Nathanael Ediger

born in Mayen (Germany)

2023

LHC event generation with an Autoregressive Transformer

From variable orderings to joint training

This Bachelor Thesis has been carried out by Nathanael Ediger
at the
Institute for Theoretical Physics in Heidelberg
under the supervision of
Prof. Dr. Tilman Plehn

Abstract

Machine learning is increasingly important in physics with Neural Networks emerging as a promising tool to accelerate LHC event generation, making it possible to generate large amounts of data in a much faster way. We will look at processes where a Z-Boson and a variable number of jets are produced through proton collision. We have engaged in the task of optimizing the autoregressive transformer (AT), which proves to be a powerful tool in the context of event generation. As the transformer is influenced by the order of the input sequence, we have been working on finding good orderings. We trained models on multiple randomly generated orderings, to distinguish good from bad ones, by comparing the quality of samplings for different orders. However, it turns out that the so called "permutrain" models learn to sample every component independently and thus give almost identical results for any sampling order. In addition, we devoted ourselves to joint training of events with different jet counts. Due to the structure of the AT we can use events with lower jet numbers to support and enhance the generation of events with larger jet quantities.

Zusammenfassung

Maschinelles Lernen spielt eine zunehmend zentrale Rolle in der Physik, wobei neuronale Netzwerke als vielversprechendes Instrument hervortreten, um die Ereignisgenerierung am LHC zu beschleunigen und somit die Möglichkeit bieten, große Datenmengen auf wesentlich schnellere Weise zu generieren. Wir untersuchen Prozesse, bei denen durch Protonenkollision ein Z-Boson und eine variable Anzahl von Jets erzeugt werden. Unsere Aufgabe besteht darin, den autoregressiven Transformer (AT) zu optimieren, der sich als leistungsstarkes Werkzeug im Kontext der Ereignisgenerierung erwiesen hat. Da der Transformer durch die Reihenfolge der Eingabesequenz beeinflusst wird, arbeiten wir daran, optimale Reihenfolgen zu finden. Modelle wurden auf mehreren zufällig generierten Reihenfolgen trainiert, um gute von schlechten zu unterscheiden, indem wir die Qualität von Stichproben für verschiedene Reihen verglichen. Es stellt sich jedoch heraus, dass die sogenannten "permutrain"-Modelle lernen, jede Komponente unabhängig zu sampeln, und somit für jede Sampelreihenfolge nahezu identische Ergebnisse liefern. Zusätzlich haben wir uns dem gemeinsamen Training von Ereignissen mit unterschiedlichen Jet-Zahlen gewidmet. Aufgrund der Struktur des AT können Ereignisse mit geringerer Jet-Anzahl genutzt werden, um die Generierung von Ereignissen mit größeren Jet-Mengen zu unterstützen und zu verbessern.

Contents

1 Introduction **1**

2 Physical Background **3**
2.1 Data Set 4
2.2 Distributions that will be examined 5

3 Machine Learning Background **8**
3.1 Prelude 8
3.1.1 Structure 8
3.1.2 Linear layers 9
3.1.3 Activation functions 9
3.1.4 Backpropagation 9
3.2 Transformers in machine learning 9
3.3 The architecture of an autoregressive transformer 10
3.3.1 The Autoregressive Transformer 10
3.3.2 Linear Embedding 11
3.3.3 Self Attention 12
3.3.4 Feed-Forward 13
3.4 Training algorithm 14
3.5 Sampling algorithm 15

4 Sampling from all channel orderings (with permutrain) **16**
4.1 Training on fix orderings 17
4.2 Sampling after varying the order 18
4.3 Training for several orderings at once (Permutrain) 19
4.3.1 Transition from fix to permutrain 19
4.3.2 Event-wise and batch-wise permuting 20
4.3.3 Improvement of the distributions with higher number of epochs . 21
4.3.4 Comparing different samples 21

5 Jet Multiplicity and Joint Training **23**
5.1 3-Jet-Events 23
5.1.1 Fair Training 24
5.2 5-jet-events 25
5.2.1 Investigating the support of specific jet numbers 25
5.2.2 Most optimal setting 27

6 Conclusion **29**

1

Introduction

With the Large Hadron Collider (LHC) it became possible to enforce high energy collisions between particles, to discover and examine standard particles and their behaviour. The most popular result is the discovery of the Higgs Boson [1, 2], which completes the Standard Model (SM) [3]. With its use it became possible to investigate particle scattering and decays, QCD effects, hadron showers and more. Our goal is to advance our understanding of fundamental particles and the laws of physics, by searching for new particles or studying the fundamental forces. We do that by performing high energy collisions and observing the resulting particles and their behaviour. The main subject of the research at LHC are tests of leading theories like the Standard Model and investigations of e.g. Dark Matter [4].
The usual way to combine theory and experiment is the following. In theoretical departments experimental results will be simulated by using tools like MadGraph, Pythia, Delphes, etc. [5–9]. Each of these tools covers a step of the simulation chain. First computing the amplitudes and probabilities of our actual events (MadGraph), then simulating the parton showering, hadronization, and decay of particles (Pythia) and finally computing the response of a particle detector to the produced particles, giving us the measured events.
The same thing will be done in experimental departments the other way around, so a theory will be postulated by doing experiments and several other tools to evaluate the data. One example are the experiments at LHC. By doing both we have the hope of meeting with similar results somewhere on the way. This event generation chain is illustrated in Fig. 1.1.

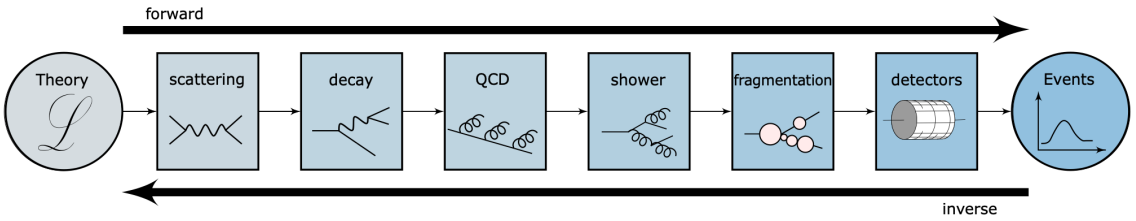

Figure 1.1: Event generation chain; taken from [10]

The theoretical predictions are done by the mentioned tools using Monte-Carlo-methods. These simulations process physical laws to generate events, but they are computationally costly and take a lot of time. So machine learning is used to enhance the process, by training a neural network on of a number of events, which are Monte-Carlo-simulated.

The model learns the phase space density and is able to reproduce physical events. Though the training of the model takes time, this process accelerates the event generation, because the sampling process is extremely fast and computationally cheap. In this process only the physical information of the original data set is processed so there is only a reasonable number events that can be considered statistically relevant. This number is dependent on the size of the original data set and an amplification factor which we call the GANplification factor[11].
In recent years many different kinds of networks have been probed on that task, including adversarial networks (GANs) [12, 13], invertible neural networks (INN) [14] or variational autoencoders (VAEs)[15].

In our case we want to examine the capability of an autoregressive transformer to tackle this task. The main structural feature of the AT is that it interprets the input components as a sequence. It learns the phase space density and generates events by predicting the following component, conditioned on the previous components.
We expect the AT to work well especially on high jet-multiplicity's (events with multiple of similar components), due to its structure. The architecture of the AT allows us to train the same model on events with different numbers of jets. We will use this to support the training of difficult tasks by adding a training of easier tasks with less jets.

2

Physical Background

As mentioned, our physical background is the investigation of particle collisions in the LHC. When high energy protons collide, a tremendous amount of energy is released. This can lead to the creation of new particles, including quarks and gluons.
One particular kind of event will be looked at, where in the collision process of two protons a Z-Boson is created by a quark and an anti-quark, which will decay in a muon and an anti-muon. The Feynman-diagram of this process is visualized in following Fig. 2.1:

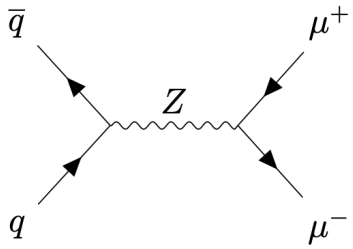

Figure 2.1: Feynman-diagram of the muon production from a Z-Boson; taken from [16]

The other hadrons that resolve from the collisions can be summarized as jets [17], using jet algorithms like anti-kt [18], that tell us which comments belong to which jet. Jets are particle sprays resulting from the collision process and the hadronization of quarks and gluons. They will be described by four components, by computing a mean angle and a summed up momentum and mass. From then on these jets will be treated as a single particle to enable the computer to isolate the main information of the events structure and to make the data more handy.

In the Feynman-diagram Fig. 2.2 we have another example of muon and anti-muon production including an earlier process, depending on the initial state particles. In this example the final state has two additional jets. The full collision process can be described as:

$$pp \rightarrow Z_{\mu\mu} + \text{jets} \tag{2.1}$$
$$Z_{\mu\mu} \rightarrow \mu^+ + \mu^- \tag{2.2}$$

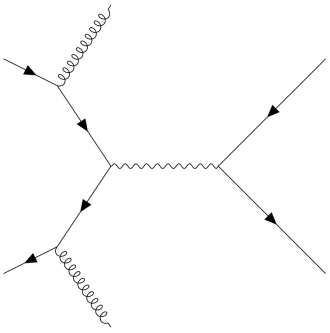

Figure 2.2: Feynman-diagram of jet production with two jets and the Z decay into muons; taken from [16]

2.1 Data Set

The resulting particles of e.g. two jets and two leptons will be measured and characterized in a vector of 16 components. Each jet and lepton have a mass, a momentum and two angular components, which are relative to the direction of the first lepton. From now on the first lepton will be described with only one angular component, due to the cylindrical symmetry. The masses of the leptons will be eliminated from the list of components as well, because they are fundamental physical quantities, which are fix at $\approx 105.658\,\text{MeV}$ so the model does not need to learn them. The dimension of this vector of components is defined by the number of jets. The number of components n equals $(2 + n_{\text{jets}}) \times 4 - 3$.
In this way, the events in the particle accelerator can be described by a vector of n components.

The data sets that will be used are created by Monte-Carlo-simulation using the tools MadGraph and Pythia. In the process of our data set creation we generate 0 to 5 jet events, but there are more events created for smaller jet numbers. The reason given is that events with more jets are more computationally costly. The final shape of our data set is shown in Tab. 2.1.

Number of jets	Shape(Number of events, Sequence length)
0	(4776763, 8)
1	(4776763, 12)
2	(4776763, 16)
3	(4776763, 20)
4	(1064147, 24)
5	(235823, 28)

Table 2.1: Shape of our data set, before eliminating the unnecessary components

The data sets for 0 to 5 jet events will be used to compare the work of the transformer on

different jet multiplicities and to take a look at how the transformer responses to learning several jet multiplicities simultaneously. Of course it is expected that the transformer and any Neural Network learn lower jet events easier, as the tasks are less complex.

2.2 Distributions that will be examined

For every component there is a distribution and a plot, which is basically a histogram of the value of one component in different events. These will be called 1d-plots and are shown in Fig. 2.3.

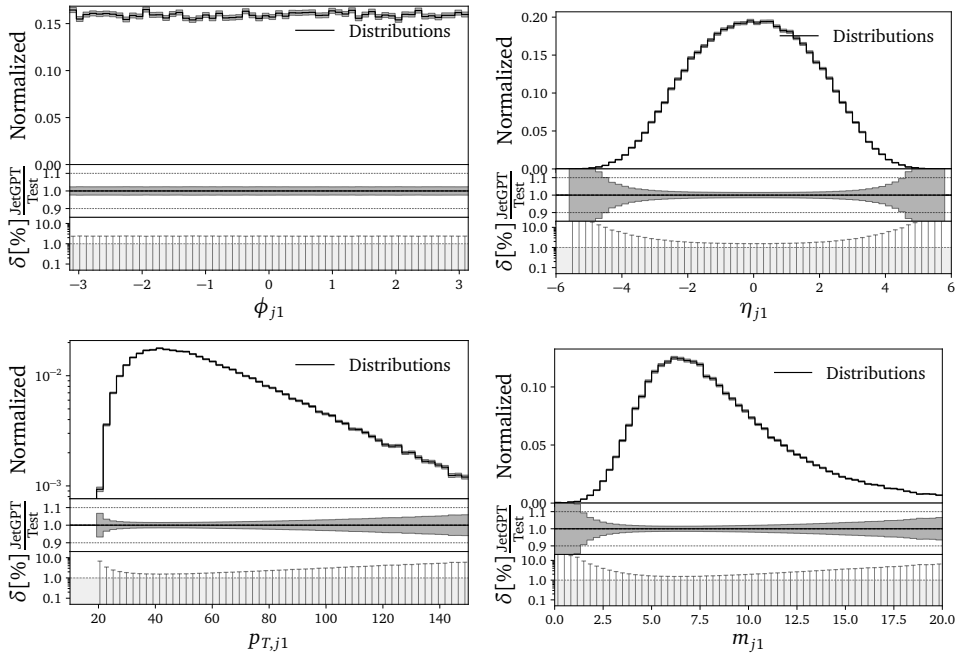

Figure 2.3: ϕ(angle), η(angle/pseudo-rapidity), p_T(momentum), m(mass) of the first jet

To be more exact the η component is not actually an angle, but the pseudo rapidity of the particle which by definition contains the information of the elevation angle $\eta = -\ln\left[\tan\left(\frac{\theta}{2}\right)\right]$. ϕ is the azimuth angle orthogonal to the particle stream.

Several other distributions that will be looked at are the ΔR-distribution and the distribution of the Z-mass. Those are especially important, because they carry important and difficult to learn features.

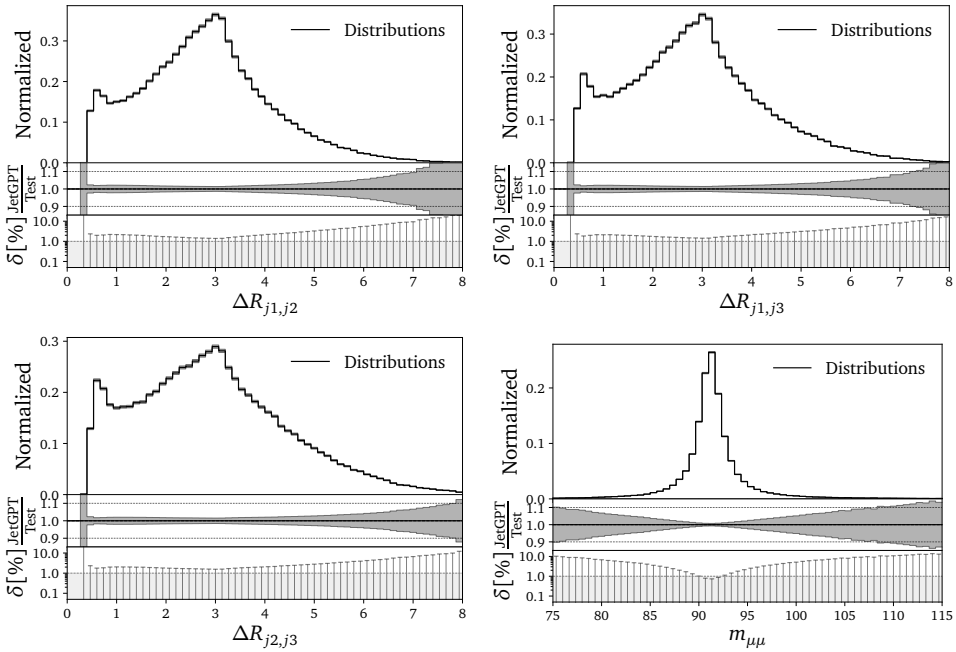

Figure 2.4: The three ΔR-plots and the Z-mass

There are three ΔR distributions in the case of 3-jet-events, which are defined by the angles of two jets in the following way,

$$\Delta R_{j1,j2} = \sqrt{(\Delta\phi)^2 + (\Delta\eta)^2} \tag{2.3}$$

$$\text{with } \Delta\phi = \phi_{j1} - \phi_{j2} \text{ and } \Delta\eta = \eta_{j1} - \eta_{j2}. \tag{2.4}$$

Though η is not actually an angle, its values are within the range of 0 to 5 so it can be approximately treated as an angle over 2π.
The Z-mass is one of the more difficult components to learn. Especially because the model needs to compute it only from correlations. So later on we changed the input components by boosting the lepton particles into the rest-frame of the Z-Boson. Instead of giving the model the initial components

$$\phi_{l1}, \eta_{l1}, p_{T,l1}, \phi_{l2}, \eta_{l2}, p_{T,l2},$$

we gave the model

$$\phi_Z, \eta_Z, p_{T,Z}, m_Z, \phi_{l2'}, \eta_{l2'}.$$

The momentum therefore is the sum of the lepton momenta $p_{T,Z} = p_{T,l1} + p_{T,l2}$. With the information of the Z-mass given initially, the peaks will get almost perfect. During this work this will not yet be used, so the Z-mass-peak can only be learned from the correlations of the components. It is one of the most difficult distributions for the model to learn and a good plot to evaluate the quality of our training.
Another interesting feature are the smaller peaks on the left, which are an artefact of the ISR enhancement [19]. When separating our jets further by splitting them in smaller

jets we get jets that are closer together and if we proceed this we get a divergence close to 0 as every particle is considered a jet. To get rid of that we position a cut, still leaving the peak as a secondary feature to the distribution, which is quite hard to learn. Because they are difficult to learn as well they are likewise a good measure of the quality of our models. The physical explanation of the cut is therefore that two jets can not be arbitrarily close together.

3

Machine Learning Background

3.1 Prelude

Neural Networks (NN) enable us to enhance the speed of event generation. Monte-Carlo-simulations are slow and very computationally intensive tasks, in contrast to sampling from a NN. When reproducing events with a NN that learned the phase-space-density, physical events can be generated in a much faster way. There are several different kinds of NNs and all bring different advantages and ideas with them. With the goal of finding the optimal way to do these event generations, we investigated the autoregressive transformer.
Before coming to that we will clear up some important terminology and basic machine learning concepts.

3.1.1 Structure

The structure of a basic NN consists of multiple layers of neurons (Illustrated in Fig. 3.1).

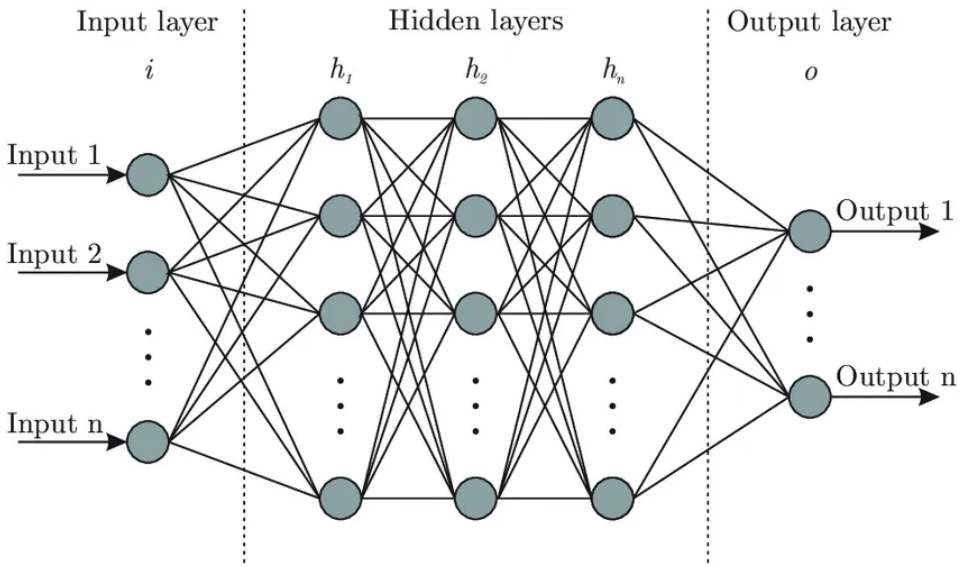

Figure 3.1: Structure of a fully connected neural network; taken from [20]

A basic neural network consists of an input layer, where data is fed into the network,

one or more hidden layers, which process the input data through weighted connections and an output layer, which produces the final prediction. Neurons within each layer are interconnected, and the network learns by adjusting the weights of these connections during training to optimize its performance on a given task.

3.1.2 Linear layers

A linear layer is a transformation, where every input neuron is connected to each output neuron with a weight and a bias ($output = input \times weights + bias$). Linear layers are responsible for learning relationships in the data and are often followed by non-linear activation functions.

3.1.3 Activation functions

Activation Functions introduce non-linearities to a NN. They allow the network to model more complex data. The most common activation functions are the sigmoid, hyperbolic tangent and rectified linear unit (ReLU).

3.1.4 Backpropagation

Backpropagation is the actual training algorithm that minimizes the error between a predicted output and the actual outputs of the neural networks. Backpropagation will be done after feeding the input data through the network, creating a prediction and an error for the prediction. The gradient of the error with respect of the models parameters is calculated and used to update the models parameters. This process is done repeatedly, iteratively updating the model and improving the models parameters.

3.2 Transformers in machine learning

Transformers are a type of machine learning model architecture that was introduced in the paper "Attention is All You Need" [21]. Transformers have since become a fundamental architecture for various natural language processing (NLP) tasks and have been extended to other domains as well. They are known for their ability to capture long-range dependencies in data and parallelize training effectively.
The key innovation of transformers is the self-attention mechanism, which allows the model to weigh the importance of different parts of the input sequence when making predictions for a particular element in the sequence. This mechanism enables transformers to process input data in parallel, making them more efficient than other neural network architectures for certain tasks.

3.3 The architecture of an autoregressive transformer

This research focuses on working with the Autoregressive Transformer for LHC event generation, which was presented by our group in that context [22]. The main difference to most generative models is that the AT would interpret the phase space vector $x = (x_1, ...x_n)$ as a sequence of elements x_i. Like this we can factorize the probability $p_{model}(x|\theta)$ as a product of n one-dimensional conditional probabilities:

$$p_{model}(x|\theta) = \prod_{i-1}^{n} p(x_i|x_1, ...x_{i-1}) \approx p_{data} \tag{3.1}$$

All n probabilities are only dependent on the earlier components. That leads to the property, that a distribution $p(x_i|x_1, ...x_{i-1})$ is easier to learn than a distribution which is conditional on the full phase space vector x. Another advantage of this property is, that we can choose freely in which order we will feed the components to the model. We can put the more challenging phase space directions early in the sequence, as $p(x_i|x_1, ...x_{i-1})$ gets easier to compute for small i, because it has less conditional components. The conditional dependence on the $x_1, ...x_{i-1}$ components will be encoded in a representation $\omega^{(i-1)}$. By using the AT we map the x_i components on a representation $\omega^{(i)}$. The architecture of the transformer is illustrated in Fig. 3.3.
When sampling the resulting probability density function (PDF) Eq. 3.2 will be used to make a prediction for the following x_i component.

$$p(x_i|x_1, ...x_{i-1}) = p(x_i|\omega^{i-1}) \tag{3.2}$$

The construction of this PDF follows a Gaussian Mixture Model (GMM), which represents the probability function as a sum of weighted Gaussian components:

$$p(x_i|\omega^{(i-1)}) = \sum_{\text{Gaussian } j} w_j^{(i-1)} \mathcal{N}(x_i; \mu_j^{(i-1)}, \sigma_j^{(i-1)}) \tag{3.3}$$

All representations and the attention matrix A_{ij}, which encodes the relation between two components x_i and x_j, will be computed in parallel as shown in Fig. 3.2

3.3.1 The Autoregressive Transformer

Let us now take a look at the architecture of our transformer.
The network will start by receiving a sequence of components in a specific order and adding a zero in front of the sequence. This will be an auxiliary x_0 component, which will be given the network to have a starting point to predict the first actual component from. Then every component x_i will be embedded in a d-dimensional latent space through a linear projection, denoted as $x_{i\alpha}$. This vector will run through several Transformer-Decoder

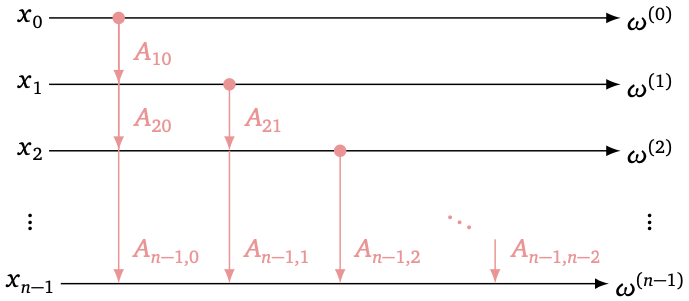

Figure 3.2: Autoregressive computation of the representations, with the attention matrix A_{ji} and an added $x_0 = 0$ component; taken from [22].

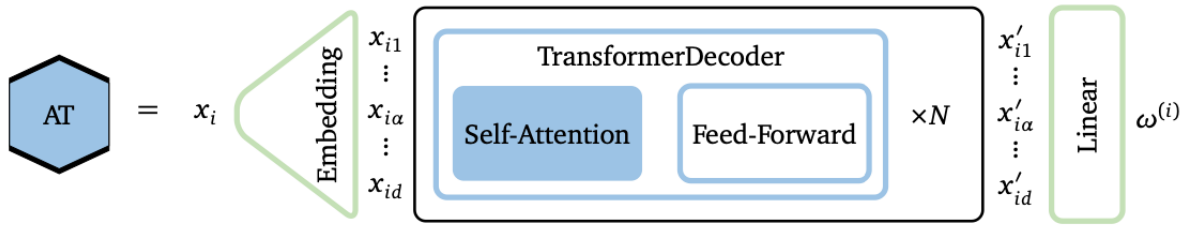

Figure 3.3: Architecture of the autoregressive transformer; taken from [22]

blocks transforming it into a latent space representation $x'_{i\alpha}$. The resulting vector will again be processed through a linear layer giving us the latent space representation $\omega^{(i)}$.

3.3.2 Linear Embedding

When representing the components into the latent space the vector $x_{i\alpha}$ is created by three vectors. Each vector carries essential information about the component that adds up to its individual point in latent space.

- The jet embedding tells how many jets the whole event has, so it is the same for all events with the same number of jets.
- The channel embedding carries the information of which particle and what type components it is. e.g. the mass of the first jet m_{j1}.
- The token embedding is different for every event, individualising them in the latent space, by embedding the value of the momenta.

The first two are called the positional encoding (aka. positional embedding).
This changes a little when implementing the permutated training. The idea of permu-training is to switch the ordering during training. More on that will be discussed later. In terms of the embedding it means that it is necessary to add another encoding vector.

3.3.3 Self Attention

The self-attention mechanism is the most defining part of the transformer. It allows the model when processing each position to capture dependencies and relationships between different elements in the sequence.
When we compute our latent space projection $x'_{i\alpha}$ each input component x_i will be associated with three linear projections giving us three vectors $q_{i\alpha}$, $k_{i\alpha}$ and $v_{i\alpha}$ per component i. These three are called query, key and value vector. The complete computation of our resulting vector $x'_{i\alpha}$ is Eq. 3.4.

$$x'_{i\alpha} = A_{ij} v_{j\beta} = \mathrm{Softmax}_j \left(\frac{q_{i\delta} k_{j\delta}}{\sqrt{d}} \right) v_{j\beta} \tag{3.4}$$

$$= \mathrm{Softmax}_j \left(\frac{W^Q_{\delta\gamma} x_{i\gamma} W^K_{\delta\sigma} x_{j\sigma}}{\sqrt{d}} \right) W^V_{\alpha\beta} x_{j\beta} \tag{3.5}$$

Let's break it apart and look at the different parts.

First we want to define an attention matrix A_{ij} holding at every value the attention score between x_i and x_j, which is commonly computed by the scalar product. To make the matrix trainable we encode our phase components with learnable transformations $W^{Q,K}$, which will allow the transformer to choose a useful basis in the latent space, giving us our query and key vector:

$$q_{i\alpha} = W^Q_{\alpha\beta} x_{i\beta} \tag{3.6}$$

$$k_{j\alpha} = W^K_{\alpha\beta} x_{j\beta} \tag{3.7}$$

Using the scalar product on these two vectors will give us the attention score, which is the similarity/compatibility between the transformed representations of elements i and j.

$$A_{ij} \sim q_{i\alpha} k_{j\alpha} \tag{3.8}$$

The resulting scores need to be scaled, so they are not dependent on the size of the latent space, so we will divide by $\sqrt{d}$ to get rid of the dependence. Also we want to normalize the components, so that the sum of all j entries referring to a certain i are 1.

$$\sum_j A_{ij} = 1 \tag{3.9}$$

We can achieve that by using the SoftMax-function which is defined in the following way.

$$\mathrm{Softmax}_j(x_j) = \frac{e^{x_j}}{\sum_k e^{x_k}} \tag{3.10}$$

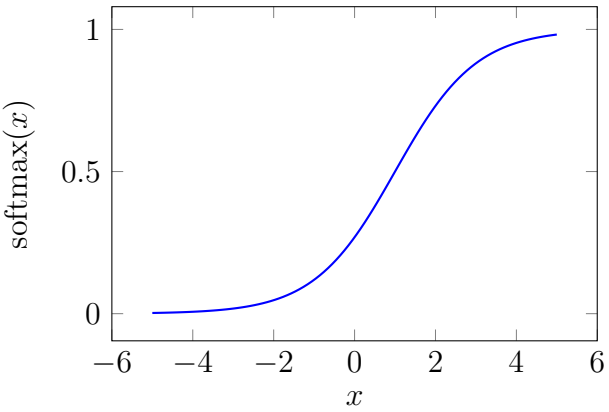

Figure 3.4: SoftMax Function

The SoftMax-function which is an extension of the Sigmoid function is commonly used for ML tasks and maps a vector of real numbers to a probability distribution.
Our attention matrix can now be expressed as

$$A_{ij} = \text{Softmax}_j\left(\frac{q_{i\alpha}k_{j\alpha}}{\sqrt{d}}\right) = \text{Softmax}_j\left(\frac{W^Q_{\delta\gamma}x_{i\gamma}W^K_{\delta\sigma}x_{j\sigma}}{\sqrt{d}}\right). \tag{3.11}$$

Finally we will apply the attention matrix to the input data, which as well will be transformed into latent space with a learnable matrix W^V, giving us $v_{j\alpha}$. In conclusion we get to Eq. 3.4 as a way to compute a latent space representation for our data.
These attention layers can be applied multiple times on our input data. This is called multi-headed-attention. So for each input component we get $3 \times n_{\text{head}}$ key, query and value vectors. Also the whole Transformer-Decoder-block can be connected several times in series, which is implied by the $\times N$ in Fig. 3.3.

3.3.4 Feed-Forward

The second part of our Transformer-Decoder-block is the Feed-Forward function. The Feed-Forward is another type of transformation with learnable weights that helps our NN to capture complex patterns and relationships. It consists of two linear Transformations using learned weights and a non-linear activation function. In our case a GELU function [23] is used, which can be approximated like this

$$GELU(x) \approx 0.5x(1 + \tanh[\sqrt{2/\pi}(x + 0.044715x^3)]). \tag{3.12}$$

Because of its non-linearity it enables the model to learn more complex representations.

3.4 Training algorithm

As mentioned in Fig. 3.2, the training will evaluate all components in parallel.

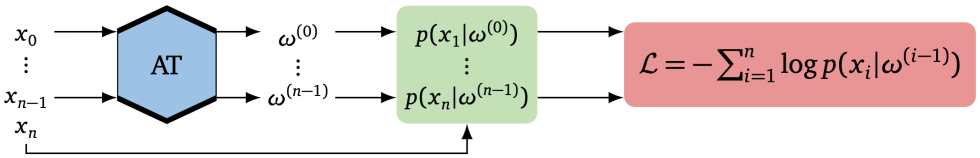

Figure 3.5: The structure of the training algorithm of the AT; taken from [22]

After computing the linear representations the network will compute the probability distributions $p(x_i|\omega^{i-1})$ and with them the loss function illustrated in Fig. 3.5. The used loss function is a typical log-likelihood function

$$\mathcal{L} = \sum_{i=1}^{n} \langle -\log p(x_i|\omega^{(i-1)}) \rangle_{x \sim p_{data}}. \tag{3.13}$$

With the loss function the model receives a score about the quality of the predictions. The gradient of the loss is afterwards computed in respect to the different weights ($W^{Q,K,V}$). These gradients are propagated backwards through the decoder layers. For that reason this process is called back-propagation. All the weights will be updated by the gradient of the loss function. Importantly, since the transformer is an autoregressive model, where each step depends on the previous one, the gradients are propagated backwards step by step considering the dependencies.
This process will be repeated in several epochs until the model reproduces the events on an acceptable level. Our model will go through the whole training data once per epoch updating the model, always after a number of a fraction of the events, which is called a batch.

3.5 Sampling algorithm

The sampling works a little different. Compared to the likelihood evaluation this process is less efficient, because it can not be parallelized. The structure is illustrated in Fig. 3.6.

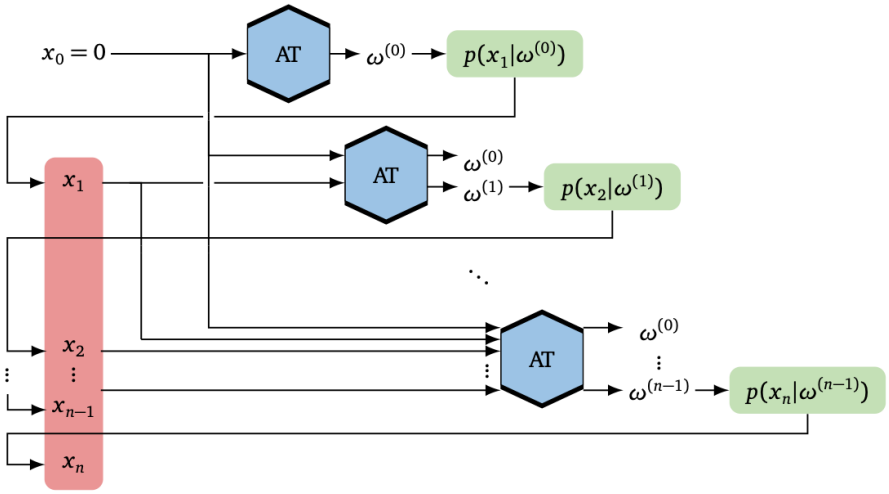

Figure 3.6: The structure of the sampling algorithm of the AT; taken from [22]

The process can not be parallelized, because the input values x_i are predicted with the probability density function $p(x_i|\omega^{(i-1)})$. And the representation $\omega^{(i-1)}$ is computed by the transformer block applied on x_{i-1}. Like earlier, an auxiliary initial component $x_0 = 0$ will be inserted. The representations $\omega^{(0)}, ...\omega^{(i-2)}$ have to be regenerated in every step. This makes the task more computationally expensive. When looking at the process it still it is very fast compared to the training time.

4

Sampling from all channel orderings (with permutrain)

As mentioned the current task is to improve our autoregressive transformer model. There are several ideas to look into, some of which will be presented in the following part.

One of our first and main ideas is rooted in a result of the paper, that introduces the autoregressive transformer in the context of event generation [22]. It says, that there are orderings that improve the precision of our network compared to networks trained on other orderings. We proceeded with the question of whether it is possible to find an optimal ordering and possibly an algorithm to find it for different kinds of events as well. Since our 3-jet-events have 17 values of information the number of possible permutations is $17! \approx 10^{14}$. Obviously it is not possible to train a model for every single one of them and compare them, as each training takes several hours.

The idea at hand is to train one model on multiple orderings, which will later be called "permutrain" and to sample from that training. The reasoning behind this is fairly simple, namely the sampling time is only a fraction of the time needed for the training. So it is possible to train one model and compare different samplings, assuming their results are better for good orderings and worse for bad ones. That would be a faster possibility of looking for a good or even optimal ordering.
As mentioned in the theory part, our positional encoding changes when using permutrain. For the original training it is sufficient to tell the model which component should be predicted next. With permutraining the events get another degree of freedom, because the channel order changes for every event. For that reason, it is necessary to give the model additional information so it can distinguish between different orderings. This is done with another positional encoding vector that contains the information of which was the previous component in the sequence. Like that the model can iteratively reconstruct the whole event from every component.
I want to offer an intuition. Our model tries to learn the structure of the events. If the model does not change it is arbitrary which component was last, when embedding component x_i, because the structure stays the same. If it does change our model needs to "take a look back" to know on what bases the prediction for the next step was made.

4.1 Training on fix orderings

First we will look into training and sampling with only one fix ordering to get an impression of the differences resulting from different orderings. The three most frequent orderings, here for three jet events, that we used are

- the "assumed to be" optimal order
  fix0 $[\phi_{j1},\eta_{j1},\phi_{j2},\eta_{j2},\phi_{j3},\eta_{j3},p_{T,l1},\eta_{l1},p_{T,l2},\phi_{l2},\eta_{l2},p_{T,j1},m_{j1},\ p_{T,j2},m_{j2},p_{T,j3},m_{j3}]$,
- the inverted opt. order
  fix1 $[m_{j3},p_{T,j3},m_{j2},p_{T,j2},m_{j1},p_{T,j1},\eta_{l2},\phi_{l2},p_{T,l2},\eta_{l1},p_{T,l1},\eta_{j3},\phi_{j3},\eta_{j2},\phi_{j2},\eta_{j1},\phi_{j1}]$
- and the intuitive order
  fix4 $[p_{T,l1},\eta_{l1},p_{T,l2},\phi_{l2},\eta_{l2},p_{T,j1},\phi_{j1},\eta_{j1},m_{j1},p_{T,j2},\phi_{j2},\eta_{j2},m_{j2},p_{T,j3},\phi_{j3},\eta_{j3},m_{j3}]$.

Fix4 is the order in which the events are generated, so it would be intuitive to use this order as well. The "optimal ordering" fix 0 is the ordering that was initially found by trial and error in the paper putting the jet angles first to improve the ΔR cuts. Fix1 is the inverse ordering to the optimal ordering which we choose to have a good comparison to fix0 with as much of a difference as possible.

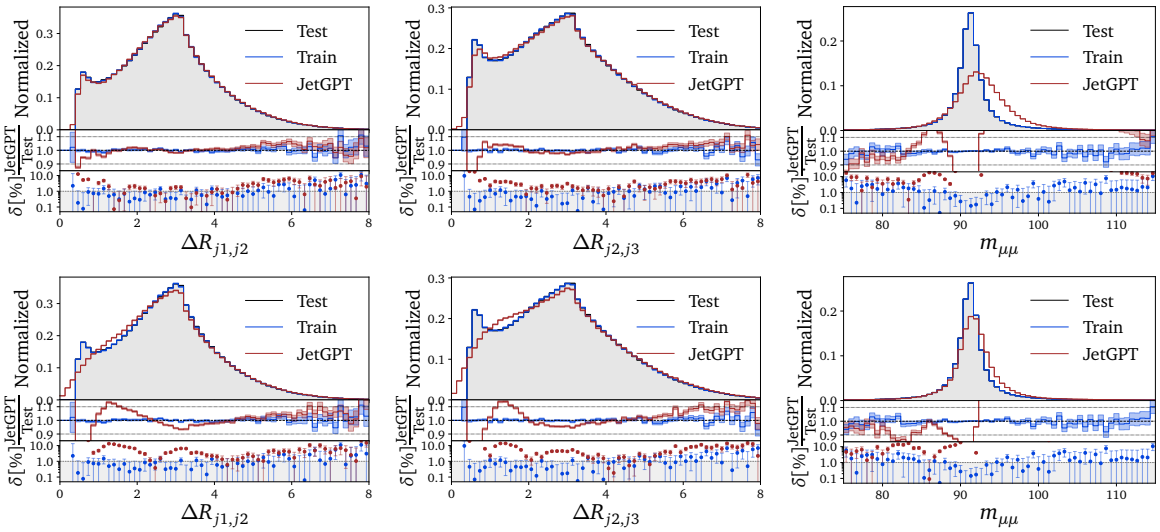

Figure 4.1: Comparing the models of fix0 (upper) and fix1 (lower)
We have three different distributions, because we split the data set in two parts. **Training (blue)** is the part of the data set on which our model is trained. **Test (black)** is the rest of the data set. We split it to get an impression of the statistical fluctuations. **jetGPT (red)** is the distribution for the sampled data, which should be close to the blue training data, if the model is well trained.

In Fig. 4.1 we have the exact equal training of two different orderings (fix0 and fix1).

It is clearly visible that as expected the ΔR cuts are learned by the fix0 model and not by the inverse one. Instead the mass distribution is better for fix1. This should give an idea of how much the results can differ for different orders.

4.2 Sampling after varying the order

Since the transformer learns to predict components based on the previous components, it will fail when we try to sample for an ordering, that is different from the channel order during training. Especially when we change the order completely, e.g. training for the order fix0 and sampling for the inverse fix1 our results look as expected horrible.

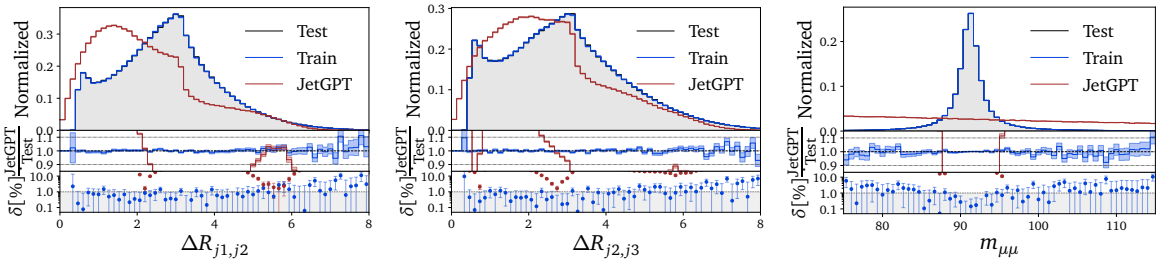

Figure 4.2: The ΔR and Z-mass distribution, after training for the order fix0 and sampling for the order fix1.

As Fig. 4.2 shows, the main feature for the ΔR distribution, which is the central peak, is barely recognisable and the Z-mass is not learned at all. The Z-Mass is completely calculated through correlations and this information is lost by inverting the ordering. The more the ordering changes the worse the training gets. When inverting the ordering completely, even the 1d-distributions are absolute chaos, as we would expect from the theory section.

If we change e.g. two components in the middle of the sequence, then all 1d-components, that come before the change, will be trained normal. This is because they receive all the information they received during training as well. The components later on are influenced by the change. This is exactly what we observe. Fig. 4.3 shows, that up to the 6th component both models are similar, but a statistic deviation. From then on differences are clearly visible, when looking at the ratio panel ($\frac{\text{jetGPT}}{\text{Test}}$).

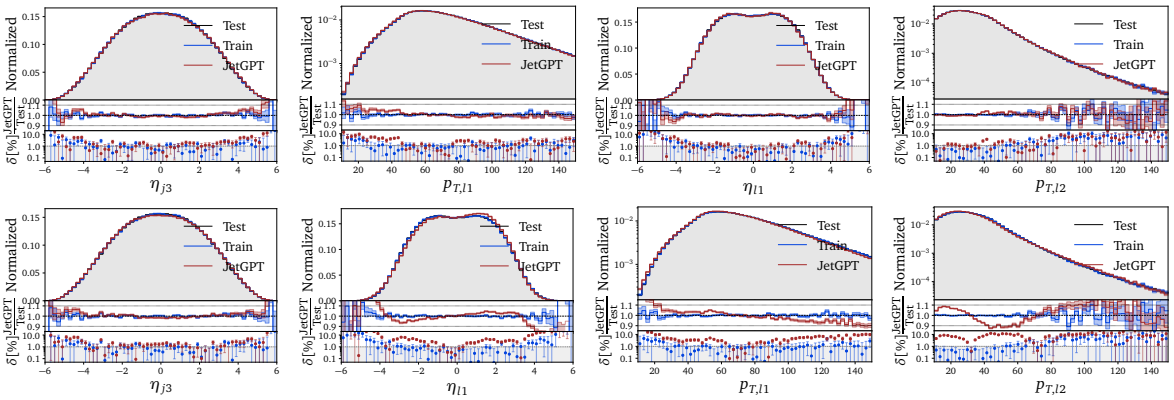

Figure 4.3: Comparing the 1d-distributions of two trained models where the 7th and 8th component of the ordering are switched in sampling of the second one. Showing the 6th, 7th, 8th and 9th component.

4.3 Training for several orderings at once (Permutrain)

4.3.1 Transition from fix to permutrain

Defining for a permutrain model is of course the number of permutations that it will be exposed to. In Fig. 4.4 different models with different numbers of permutations are being compared. The models were trained for randomly chosen permutations, excluding the sampling order fix0. We exclude this order to sample from it and to see if the model is able to reproduce the events when seeing multiple orderings that can be similar or rather different. As a reminder, when sampling from a model trained for an order, that was different than the sampling order our result were very bad (see Fig. 4.2).

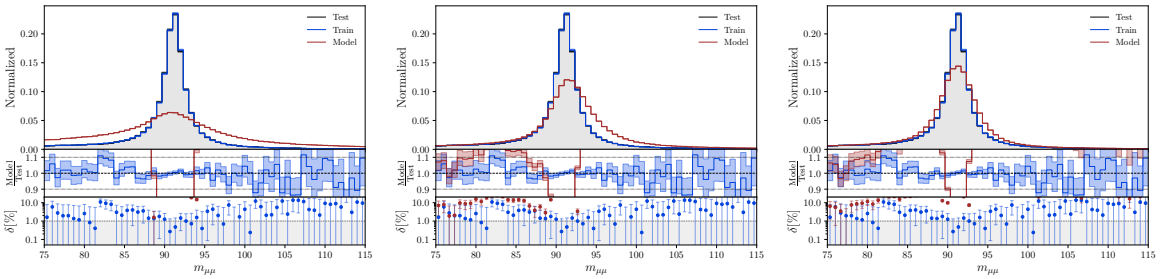

Figure 4.4: The Z-mass-distributions of six different models, seeing different numbers of permutations (4!/5!/6!). All samplings were done for the order fix0.

Though in none of the runs in Fig. 4.4 the sampling order fix0 is learned, the results get better for more differently chosen permutations. When choosing more different permutations more orderings which are similar to fix0 are learned and therefore the model still learns to sample for these events. One has to take into account, that the

Z-mass is in our given setting probably the distribution that is hardest to learn.

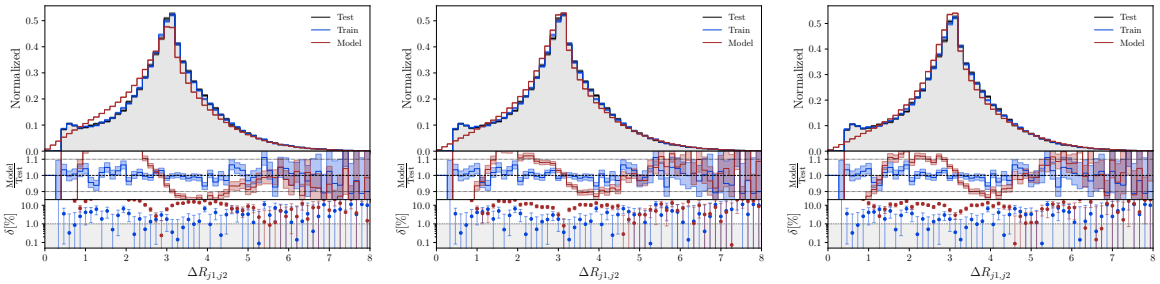

Figure 4.5: The $\Delta R_{j1,j2}$ of the same models Fig. 4.4

The same thing can be done for one of the ΔR distributions in Fig. 4.5. With very few permutations the main features of the distributions are learned, but even with many permutations the more difficult features like the ΔR cuts are not learned.

4.3.2 Event-wise and batch-wise permuting

There are several ways to implement permuted training, depending on how many different orders should be included. First, there is the possibility to change the permutation for every batch and train the whole batch on the same ordering. Secondly, it is possible to permute the ordering for every event.
The event-wise permutraining sees as many permutations as events in the training data set. Depending on the batch-size the batch-wise permutraining sees a fraction of that. In our case we choose a batch-size of 1024 so it only sees 0.1% of the permutations.

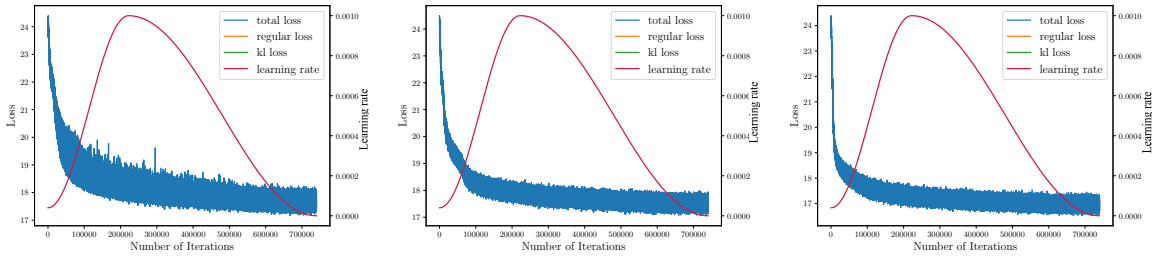

Figure 4.6: Loss functions of batch-wise (left) and event-wise (middle) permutrain and the fixtrain (right)

As presented in Fig. 4.6, the loss for the event-wise permutraining is more stable, so proceeding this will be used. Compared to the fixtrain the value to which the loss function converges is higher and therefore worse. The simple reason is the complexity of the task. We can not expect to receive better results for a permutrained model.

Which is not a problem, as that is not what we are aiming for, but to compare the orderings by sampling from a permutrained model.

4.3.3 Improvement of the distributions with higher number of epochs

The results are getting better the longer the models are trained. Still there is a reasonable number of epochs as the model reaches a plateau during training. In Fig. 4.7 the fixed order training of fix0 is compared with the permutrain model.

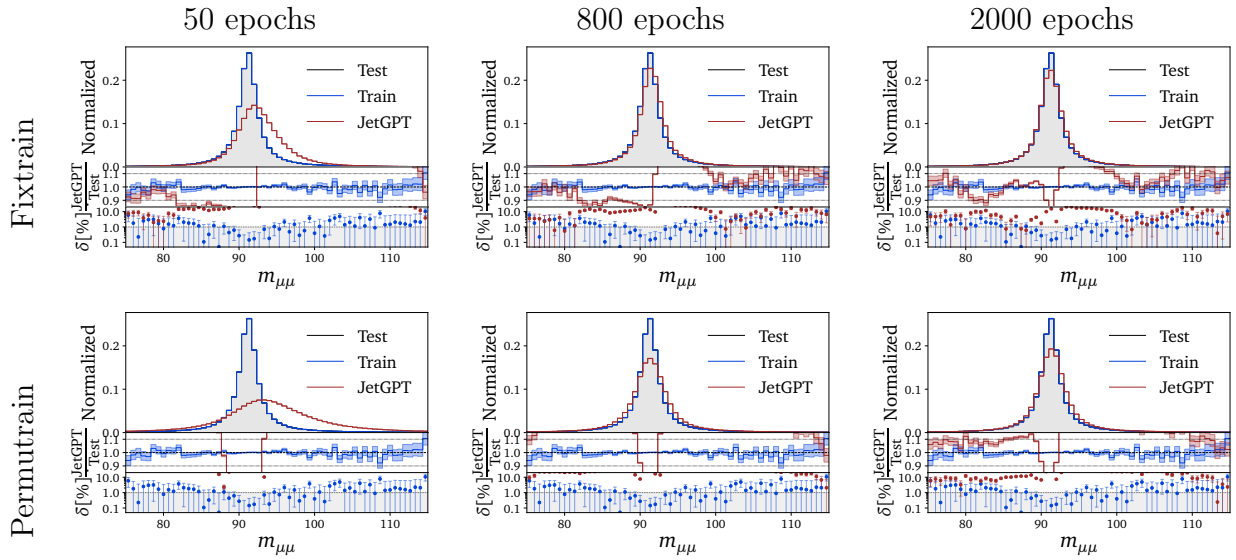

Figure 4.7: Training the Z-mass twice for different numbers of epochs (50/800/2000). Once trained with a fix ordering fix0 and then permutrained.

The mass-distributions get, as expected, better with longer training, for the fixtrain it seems, that it reaches a plateau already for 800 epochs and when training for more epochs it only show statistical deviations. The permutrain network reaches its plateau around 2000 epochs. The 2000 epoch runs for both trainings still show clearly that the permutraining is worse, as mentioned due to the complexity of the task. The runs for 50 epochs have the biggest difference, which shows that especially the permutraining profits from a longer training.

4.3.4 Comparing different samples

After all the crosschecking and examination of the properties of the transformer we take a look at the differences between different orderings used for sampling. The idea is to find an optimal ordering by sampling from the permutrained model and comparing the

samples. If a good order can be found like this, it can be used in fixtrain models, as they still deliver better results. Again we will especially look at the distribution for the Z-mass, because it is a very hard to learn distribution and therefore the differences are most visible. The model that is sampled from is trained for 2000 epochs to be sure that the plateau is reached.

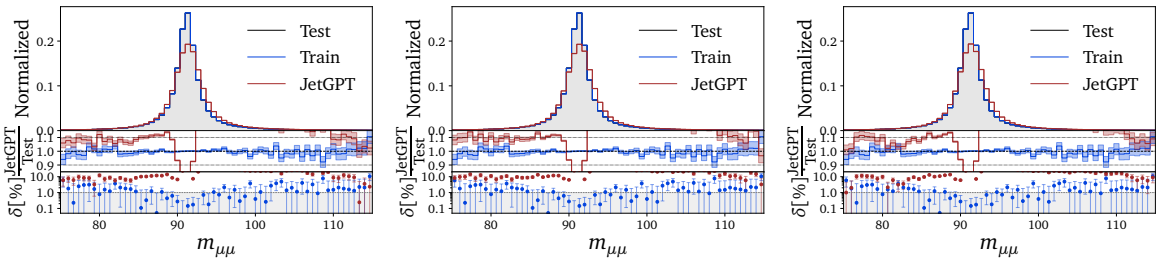

Figure 4.8: Big permutrained model trained for 2000 epochs sampling from it in three different ways: fix0, fix1, fix4

The results are surprisingly similar and differences are only of statistical nature, not allowing an assessment of quality, let alone ranking of different orderings. Contrary to our assumption that we could determine the usability of different channel sequences, the model simply does not care about the order of components, but learns to sample the events independently from another. Even the completely contrary orders fix0 and fix1 look almost identical (first and second distribution in Fig. 4.8).
As it is not possible to tell whether one of the orderings is better then the other, the assumed purpose of permutraining is invalid.

5

Jet Multiplicity and Joint Training

Another way to improve the training of an autoregressive transformer is to train the model on different numbers of jets. This is due the structure of the transformer, which allows to vary the length of the input sequence. That makes it possible to train a model e.g. on 1-, 2- and 3-jet-events simultaneously. As established the training for the one jet events is faster and easier, due to the much simpler task. So when training a model on different numbers of jets, it learns to predict the lepton components for any jet event and especially faster and easier for lower jet events. When as well training for higher jet events, the model learns to use the learned information from the smaller events. So the bigger events with higher jet numbers are supported by the training of smaller events.

5.1 3-Jet-Events

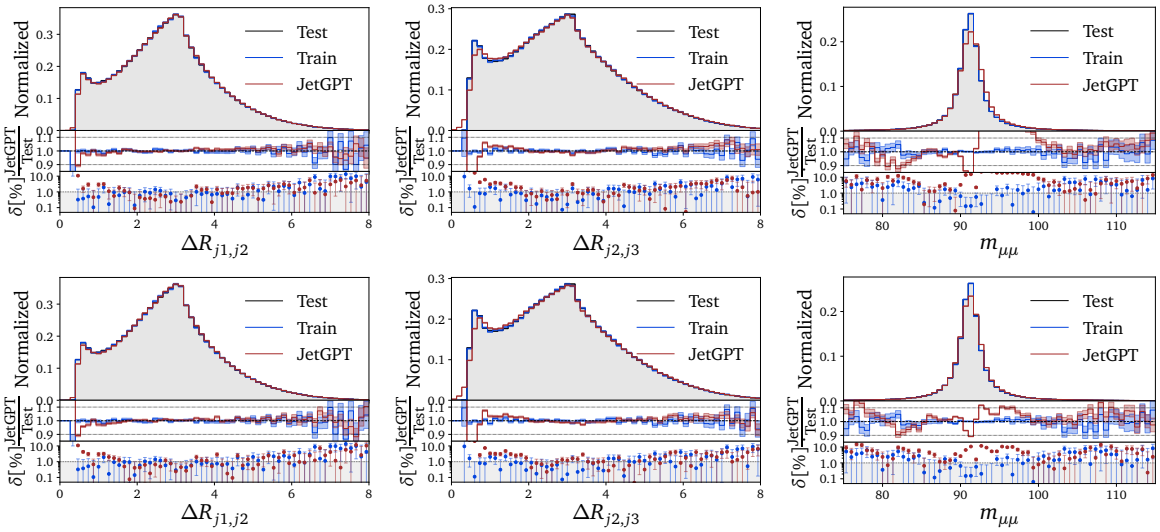

Figure 5.1: top: training 3-jet-events for 2200 epochs;
bottom: joint training of 1-, 2- and 3-jet-events for 1000 epochs.

In Fig. 5.1 the improvement of the Z-mass is clearly visible. The ΔR distributions are also better, but not as clearly. Obviously the events with lower jet counts do not carry much information about jet correlations, so it is rather surprising that the model is still able to produce the ΔR cuts, especially the ones with 3-jet in this quality, despite training for less then half of the epochs. In comparison the Z-mass is a component that

will be learned through the correlations of the lepton components, therefore it improves more with the joint training.
Fig. 5.2 shows that for shorter trainings the joint training is a bit worse the ΔR cuts are not as sharp and especially the mass-peak is not as high. So there is a point from which joint training is more efficient and from where the additional events compensate the fewer training epochs.

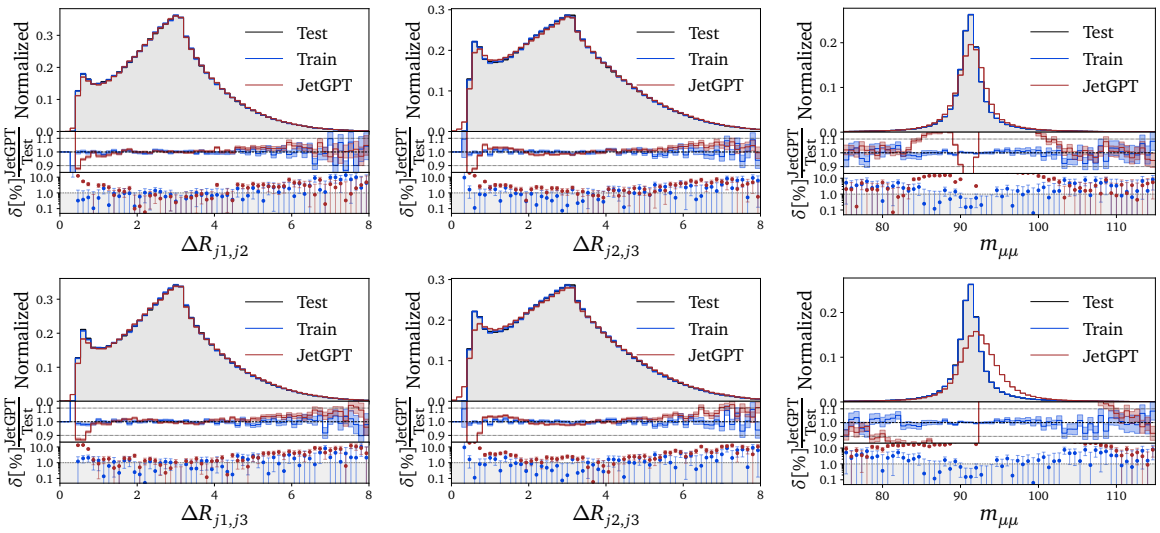

Figure 5.2: top: training 3-jet-events for 500 epochs;
bottom: joint training of 1-, 2- and 3-jet-events for 200 epochs.

5.1.1 Fair Training

When comparing this, it is necessary to take into account that we can not train a model with only 3-jet-events and a joint training for 1- ,2- and 3-jet-events for the same number of epochs. This would not be a fair comparison, as for every epoch the whole data set is evaluated Three times as many events will be seen for the joint training. Clearly the joint training will be better in that case, as it receives more information and has a much longer training time.
Considering this, it is possible to do a fair training by calculating the number of components each model sees. So with the shape of the training data (compare Tab. 2.1), if training model on one jet quantity the following applies $n_{seen-components} = n_{seq;length} \times n_{events}$.
It is also necessary to take into account, that when training on 4- or 5-jet-events joint with other events, the number of all other events will be reduced. So during training we have the same number of events for each jet quantity.
Giving an example of the simple fairness calculation for 5-jet-training and 4- and 5-jet-

training Eq. 5.1, the values are used in Fig. 5.4.

$$n(\text{components in 4-jet-sequence}) = 21 \tag{5.1}$$
$$n(\text{components in 5-jet-sequence}) = 25 \tag{5.2}$$
$$n(\text{epochs j-5-event is trained for}) = 2400 \tag{5.3}$$
$$n(\text{epochs j-4and5-event is trained for}) = 2400 * \frac{21+25}{25} \approx 1305 \tag{5.4}$$

5.2 5-jet-events

Now looking at the joint training of 5-jet-events in Fig. 5.3.

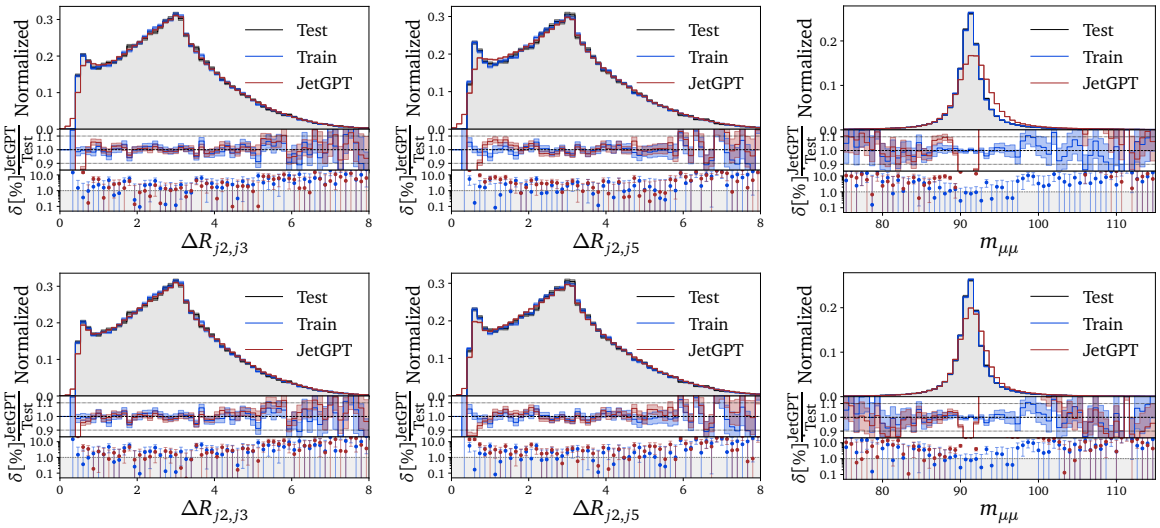

Figure 5.3: Comparing 5-jet-training for 3600 epochs (top) with joint training of 0- to 5-jet-training for 1000 epochs (bottom). Two selected ΔR and the Z-mass distribution.

The joint model was trained on all jet-events. As elaborated previously, the strongest difference is visible in the peak of the Z-mass. The ΔR distributions show consistent improvement on the cuts as well. For $\Delta R_{j2,j3}$ this was easily to expect, but for $\Delta R_{j2,j5}$ the improvement is more interesting, as this distribution is only computed for 5-jet-events. In contrast the Z-mass is computed six times. This means the more profound training of the $j2$ components supports even the 5-jet-ΔR-distributions.

5.2.1 Investigating the support of specific jet numbers

Fig. 5.4 shows a fair comparison of the 5-jet-event training to five joint trainings each with one jet-quantity. As the 4-jet-events hold more components, then any other event it is given a fraction of the training time. A reasonable expectation would be, that higher

number jet events improve the ΔR cuts more and lower number jet events improve the Z-mass, as the events with lower jet numbers see the lepton components more often.

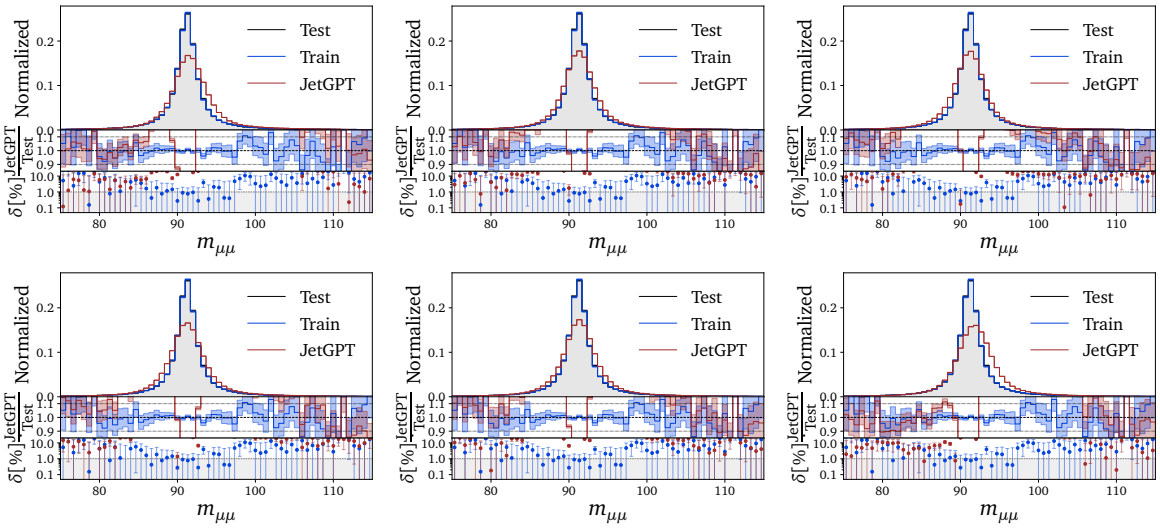

Figure 5.4: The mass distributions of 5-jet training for 2400 and the joint training including 0- (1-; 2-; 3-; 4-) jet events for 2000 (1765; 1580; 1430; 1305) epochs

In Fig. 5.4 the joint training of only one additional jet component is compared with the training of only 5-jet-events. All models see approximately the same number of components but different numbers of jet and lepton components. Still almost all the mass distributions look very similar. In all cases the joint training is slightly better then the non joint training. Only the 45-jet-event training seems to be slightly shifted. This could still be due to statistical deviations.

One can see more differences, comparing the numerous ΔR distributions of the earlier models, one of them illustrated in Fig. 5.6. The models that were supported by 0- or 1-jet-events did not learn any cuts at all. These events do not have multiple jets and they are therefore apparently no help in learning the cuts. The 5-jet-event model and the one supported with 2-jet-events learn almost all cuts and the models trained with 3- and 4-jet-events learn the same cuts, but in most cases a bit sharper then the others. The ΔR cuts that are a correlation of one or multiple events with high jet numbers are very difficult to learn, which physically and computationally makes sense as 5-jet-events are rare in comparison to 1- or 2-jet-events.

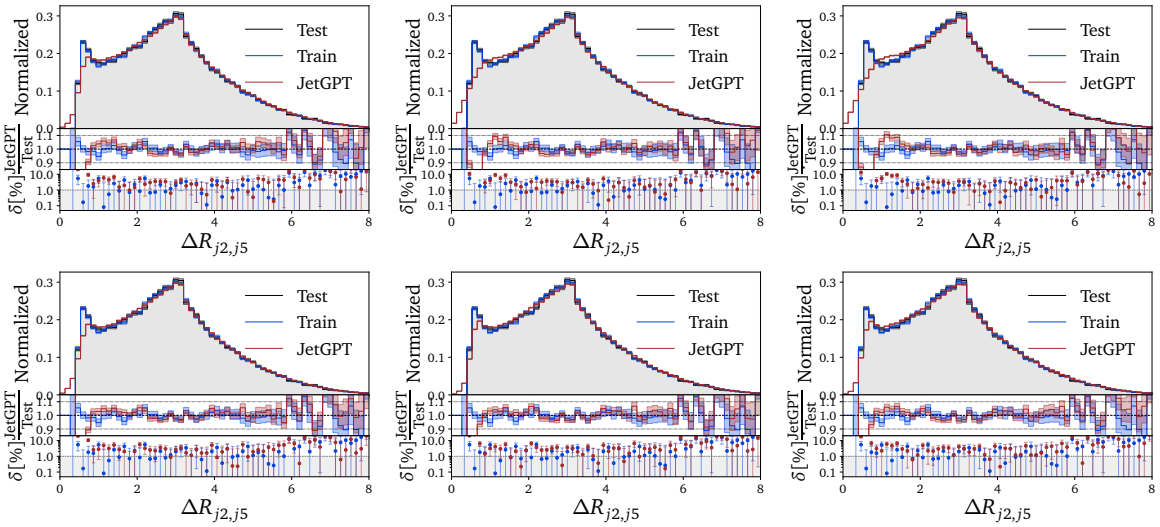

Figure 5.5: The ΔR distributions of 5-jet training for 2400 and the joint training including 0- (1-; 2-; 3-; 4-) jet events for 2000 (1765; 1580; 1430; 1305) epochs

5.2.2 Most optimal setting

These results now tells us, that it may be optimal to do a joint training of 345-jet-events, as these components supported the training the most. So now this will be compared to a fully joint training of 012345-jet-events.

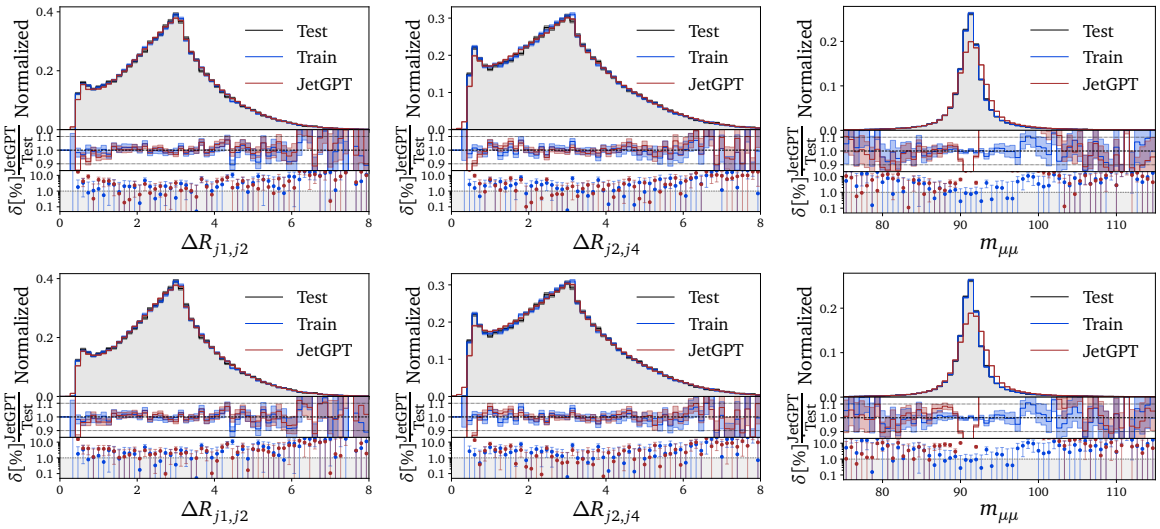

Figure 5.6: Comparison of 0-5 jet training for 1000 epochs (top) against 3-5 jet training for 1430 epochs (bottom)

Fig. 5.6 shows very little differences between the two models. The mass peak for the fully joint training is a bit higher and the second shown ΔR cut slightly sharper. So

even when using the components that support the training most, smaller events can still improve the results. Still the differences are not enormous. For any task one has to evaluate, if producing lower jet events and loading the data sets is worth the effort for the given task. But if one is doing joint training, the best way is to produce events with high jet counts, as they improve our training the most.

6

Conclusion

In this thesis we presented the characteristics and functions of the autoregressive transformer in the context of event generation. Working through the architecture and the individual properties of the AT, we isolated two ideas of improving the training on our Z-decay-events. We were able to achieve the improvement of our results and to build an intuition for further task.

Permutraining

First we engaged in the task of finding an optimal ordering using "permutrain". The training of a model on multiple orderings turns out to be possible and provides good results. The model learns to sample every component independently, so it is able to sample even for orders it has not been trained on.
Through this effect we can not, as initially planned, compare the quality of different orderings. We saw that sampling for different orderings achieved equal results and therefore we can not find an optimal ordering this way. Also the permutraining proves to be a computationally costly task compared to the fix training, so it does not serve as a replacement for the fixed training either.

Joint Training

Secondly we used the individual structure of the transformer to implement a joint training, where one model is trained on multiple jet orderings simultaneously. We showed that actual improvements are visible through this property when training for a sufficient amount of epochs. Even when training for an equal number of seen components, the more profound training of the components in events with few jets seems to enable the AT to focus on the additional jets later on.
We took a look at the degree of how much different support training data improve our results and confirmed, that e.g. 0-jet-events influence our training very differently then 4-jet-events. Events with higher numbers of jets are, even when training fair (the same amount of seen components), a better support to our event generation.
Last but not least we we trained another model "partially joint" for 3-, 4- and 5-jet-events, because 3- and 4-jet-events improved our data visibly more then the others. When comparing this model to the fully joint training we still saw small non statistical improvements in the results for the fully joint training.

Outlook

Obviously there are numerous things we can further look into to improve the autoregressive transformer. We already have another data set of events with $t\bar{t}$-events, which are more complex and therefore an interesting test for the model.
For example one of our possibilities is to implement a Bayesian network, by exchanging all the weights with a Gaussian the network is able to sample the weights, which gives us a network and sample from this network. When doing this multiple times we can compute a mean and a deviation for the results, showing the uncertainties from the training process. This has already proven to be successful.
Also we can improve our results by reweighting them. So after sampling the model compares the generated events, with the test data and adds weights to the events improving the distributions. As we still want to have unweighted events we need to include a unweighting process which eliminates events that were sampled poorly. This is very costly as a lot of events are eliminated and a training for the same quality takes much more time. So we are working on implementing a DiscFormer, which is the equivalent of a DiscFlow with a transformer instead of a Normalizing Flow. It gives us the possibility of reweighting the events during training.
Another thing our group is working on is a "two-fold-transformer" using one transformer to learn the correlations between the different particles and one to learn the component relations of each particle. Again we are making use of the transformers property to learn similar components easier. As the weights of the second transformer are shared for all the particles we have an improvement in scaling of the model.

To summarize, the Transformer is a great model to advance LHC event generation, with very special features due to its unique structure.

References

[1] G. Aad et al., *Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC*, Phys. Lett. B **716**, 1 (2012), arXiv:1207.7214 [hep-ex].

[2] S. Chatrchyan et al., *Observation of a New Boson at a Mass of 125 GeV with the CMS Experiment at the LHC*, Phys. Lett. B **716**, 30 (2012), arXiv:1207.7235 [hep-ex].

[3] S. Weinberg, *A Model of Leptons*, Phys. Rev. Lett. **19**, 1264 (1967).

[4] M. Felcini, *Searches for Dark Matter Particles at the LHC*, in 53rd Rencontres de Moriond on Cosmology (2018), pages 327–336, arXiv:1809.06341 [hep-ex].

[5] J. Alwall, M. Herquet, F. Maltoni, O. Mattelaer, and T. Stelzer, *MadGraph 5 : Going Beyond*, JHEP **06**, 128 (2011), arXiv:1106.0522 [hep-ph].

[6] 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*, JHEP **07**, 079 (2014), arXiv:1405.0301 [hep-ph].

[7] T. Sjostrand, S. Mrenna, and P. Z. Skands, *PYTHIA 6.4 Physics and Manual*, JHEP **05**, 026 (2006), arXiv:hep-ph/0603175.

[8] T. Sjöstrand, *The PYTHIA Event Generator: Past, Present and Future*, Comput. Phys. Commun. **246**, 106910 (2020), arXiv:1907.09874 [hep-ph].

[9] J. de Favereau et al., *DELPHES 3, A modular framework for fast simulation of a generic collider experiment*, JHEP **02**, 057 (2014), arXiv:1307.6346 [hep-ex].

[10] T. Plehn, A. Butter, B. Dillon, and C. Krause, *Modern Machine Learning for LHC Physicists*, (2022), arXiv:2211.01421 [hep-ph].

[11] A. Butter, S. Diefenbacher, G. Kasieczka, B. Nachman, and T. Plehn, *GANplifying event samples*, SciPost Phys. **10**, 139 (2021), arXiv:2008.06545 [hep-ph].

[12] M. Mirza and S. Osindero, *Conditional Generative Adversarial Nets*, (2014), arXiv:1411.1784 [cs.LG].

[13] I. J. Goodfellow et al., *Generative Adversarial Networks*, (2014), arXiv:1406.2661 [stat.ML].

[14] A. Butter et al., *Generative networks for precision enthusiasts*, SciPost Phys. **14**, 078 (2023), arXiv:2110.13632 [hep-ph].

[15] D. P. Kingma and M. Welling, *Auto-Encoding Variational Bayes*, (2013), arXiv:1312.6114 [stat.ML].

[16] S. Vent, *Expressive uncertainties for generated lhc events*, (2021), https://www.thphys.uni-heidelberg.de/~plehn/includes/theses/vent_b.pdf.

[17] S. Marzani, G. Soyez, and M. Spannowsky, *Looking inside jets: an introduction to jet substructure and boosted-object phenomenology*, Vol. 958 (Springer, 2019), arXiv:1901.10342 [hep-ph].

[18] G. Soyez, *The SISCone and anti-k(t) jet algorithms*, in 16th International Workshop on Deep Inelastic Scattering and Related Subjects (July 2008), page 178, arXiv:0807.0021 [hep-ph].

[19] K. Shiina, H. Mori, Y. Tomita, H. K. Lee, and Y. Okabe, *Inverse renormalization group based on image super-resolution using deep convolutional networks*, Sci. Rep. **11**, 9617 (2021), arXiv:2104.04482 [cond-mat.stat-mech].

[20] L. Shukla, *Designing Your Neural Networks*, (2019), https://towardsdatascience.com/designing-your-neural-networks-a5e4617027ed.

[21] A. Vaswani et al., *Attention Is All You Need*, in 31st International Conference on Neural Information Processing Systems (June 2017), arXiv:1706.03762 [cs.CL].

[22] A. Butter, N. Huetsch, S. P. Schweitzer, T. Plehn, P. Sorrenson, and J. Spinner, *Jet Diffusion versus JetGPT – Modern Networks for the LHC*, (2023), arXiv:2305.10475 [hep-ph].

[23] D. Hendrycks and K. Gimpel, *Gaussian Error Linear Units (GELU's)*, (2023), arXiv:1606.08415 [hep-ph], https://arxiv.org/abs/1606.08415.

Acknowledgments

Finally! Now I have the possibility to drop some words of gratitude.
Obviously special thanks to Tilman for letting me join this group, and being a great supervisor, always on point with directions, critical thoughts and questions. I have huge respect for your overview over all the groups projects, let alone my thesis that you "don't care about" :). Thank you for letting me join the up-to-date research of the group. Secondly I thank the whole work group for the relaxing and usually also interesting lunch walks. It was really great getting to know all of you and hearing from your research, from which I usually couldn't even grasp the full extent. Namely I want to mention Meave, it "was" great working with you and Jonas on understanding and improving the transformer. Thanks for helping me with all of my questions. Thanks also Björn Schäfer who kindly agreed to be the second examiner for this Thesis "The aggressive Transformer". Thank you I really had to laugh when I wrote things like 'we tackle this task'. My biggest thanks regarding this thesis go to Jonas. I am truly grateful that I had you as a supervisor. For your patience in all redundant questions, an open ear for ideas and inviting me in the project. For proofreading this thesis obviously, but also everything else and even if it's 'just' your "good morning" smile. Thank you.
Also all my friends who so spontaneously sat down and corrected my horrible english as good as possible, namely Juan whos corrections could have been a thesis for itself. But seriously thanks to all of you. Not only for that, but all our shared time! I am truly grateful for my friends and especially my family, praying for me and my health. Thanks you all!

Declaration

I hereby formally declare that I have composed the present thesis myself and without use of any other than the cited sources and aids [1].

Heidelberg, 28 November 2023 .....

Nathanael Ediger

[1] The only additional aid used was DeepL and GPT 4.0 for spelling correction and improvement of my english.