Department of Physics and Astronomy
Heidelberg University

Bachelor Thesis in Physics
submitted by

Tobias Krebs

born in Annweiler am Trifels (Germany)

1999

GeneratINNg LHC events

Using invertible neural networks to generate LHC event samples with high precision.

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

Abstract

This thesis is about generating LHC events using invertible neural networks (INNs) trained on data which again is generated using standard Monte-Carlo approaches. Our main focus lays on the precise reconstruction of the Breit-Wigner peaks in the mass distributions of intermediate W and Z bosons and t quarks. We try to improve the generated mass distributions by employing MMD and Wasserstein losses as well as by training a discriminator to teach the generator network where it still performs badly. Furthermore we discuss different preprocessing steps one can do to achieve higher accuracy.

Zusammenfassung

Diese Arbeit behandelt die Generierung von LHC Events mittels invertierbarer neuraler Netzwerke (INNs), welche auf Monte-Carlo simulierten Daten trainiert werden. Dabei ist für uns vorallem die präzise Rekonstruktion der Breit-Wigner Verteilungen der Massen von W und Z Bosonen sowie t Quarks von Interesse. Wir versuchen diese Massenverteilungen zu verbessern, in dem wir MMD und Wasserstein Lossfunktionen verwenden und einen Diskriminator darauf trainieren, dem Generator Netzwerk zu zeigen, wo es noch Unterschiede zwischen echten Daten und Simulationen gibt. Zusätzlich diskutieren wir verschiedene Transformationen die wir auf den Input des Netzwerks anwenden, um die Genauigkeit der Rekonstruktion zu erhöhen.

Contents

1 Introduction

The Large Hadron Collider (LHC) at the CERN in Switzerland has currently been upgraded for Run 3 to a ten times higher luminosity at its LHCb experiment compared to the previous run 2. In the future it will undergo additional constructions resulting in even higher event rates before 2030[1]. Comparing these events with standard model calculations requires around as many simulated events as real ones which means that the simulation rate has to be sped up as well. Currently, event simulation is done using standard Monte Carlo techniques, as in the Pythia package[2] or Madgraph[3]. Those might however become too slow to match the ever growing production rate of the new LHC runs. To solve this problem, neural networks can come in handy, as they are, once fully trained, capable of sampling new events at a very high rate. However, as those do not calculate the events from theory but instead just mimic the distribution of given samples (which are still generated using Monte Carlos), the precision of those networks is not yet on a high enough level to really use them to probe the standard model. This shows most prominently in high dimensional, narrow, correlations of the inputs. Those correlations are notoriously hard to learn for a network, compared to the 1D distributions it gets as input. The aim of this thesis is to lower the deviations between the test set and generated events to as low as one percent in the bulk, and to the order of statistical fluctuations in the tails.
In the following we will start with the physics and machine learning background. We then present our baseline results using an INN and describe the different methods we employed on top to get better mass distributions. For most of those methods we first show how they work on easier low dimensional toy-models, then discuss their impact on training and whether they actually helped by improving the masses or not. Finally we summarize our results and give an outlook on further possible improvements to the event generation chain for even higher precision.

2 LHC Physics

Particle colliders like the LHC are used to probe the standard model of particle physics on high energy scales, searching for new physics like dark matter[4] or SUSY partners of already known particles[5]. This is mostly done by colliding massive particles like protons or electrons at high center of mass energies $\sqrt{s}$. At the LHC we reach around $\sqrt{s} = 14\text{TeV}$ for proton-proton collisions. This is enough to create pairs of top quarks which are the heaviest known particles at around 173GeV/c, or also Higgs particles whose detection is one of the most prominent findings at the LHC[6]. The data those experiments collect and which we want to simulate is given by the four momenta of the final state particles $p = (E, p_x, p_y, p_z)$ which is measured at the LHC using various detectors like calorimeters, vertex locators and trackers[7]. Those detectors also alter the signal and add unwanted detector effects to it like smearing. While event generation often also takes this into account and generates the data on detector level, in this thesis we chose to generate the data without those effects. Later on it can be added using detector simulations like Delphes[8], or instead the detector effects can be computationally removed from the

experimental data to compare experiment and theory on a pre-detector level. This so called detector unfolding can also be done using neural networks[9].
For this thesis we used three quite different processes to test our networks precision:

2.1 $t\tilde{t}$ Production

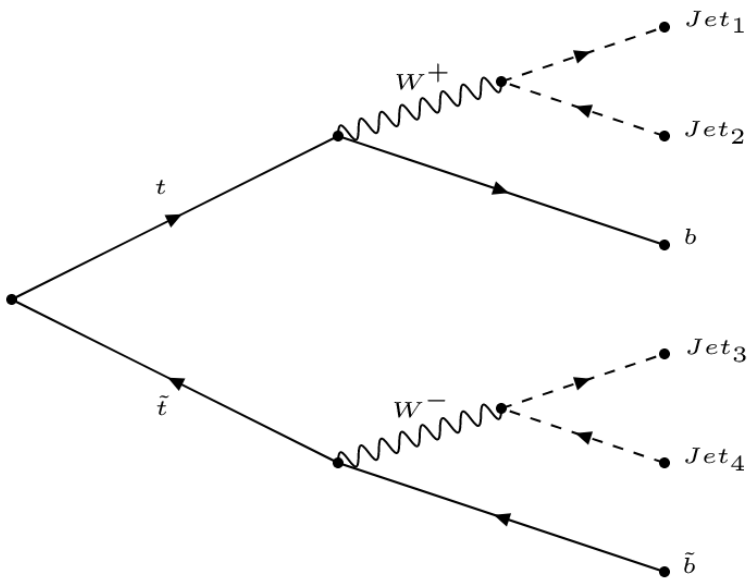

Fig. 1: Feynman diagram of the $t\tilde{t}$ process.

Our, for the generator, hardest data set consisted of $t\tilde{t}$ final state momenta. Those events are the result of proton proton collisions at the LHCb experiment: When two hadrons collide, their constituents scatter off of each other. One possible outcome is to create a $t\tilde{t}$ pair by e.g. gluon decay or quark-anti-quark annihilation[10]. Those t quarks have such a short lifetime of around $5 \cdot 10^{-25}$s that instead of producing more quarks or gluons in strong processes, they instead decay weakly into W bosons and an additional bottom, strange or down quark. We focus on processes where the t and $\tilde{t}$ decay in $W^+ + b$ and $W^- + \tilde{b}$ respectively as those are the processes the LHCb's detector architecture is designed for. The W bosons again decay in a lighter quark-anti-quark pair each, which will then produce whole jets of particles due to their strong interaction. Those jets produce many events in the detectors, and clustering the events back together to the jets they belong to can be done using many different algorithms like k_T[11], *Cambridge-Aachen*[12] or *anti-*k_T[13], but there are also approaches using neural networks[14]. As the t and $\tilde{t}$ are produced by constituents of the protons colliding which may have different shares of the proton momenta, we do not have momentum conservation in beam direction, however we do have momentum conservation perpendicular to the beam. As we can assume the hard jets to be massless at $E^2 \approx p^2$ and since we know the b quarks mass to high precision already, we can use energy conservation to reduce the phase space dimensionality from $6 \cdot 4 = 24$ to 18. Enforcing the momentum conservation can

reduce this further to 16 dimensions and we will test whether this reduction will help our network or not.

2.2 Drell-Yan Process

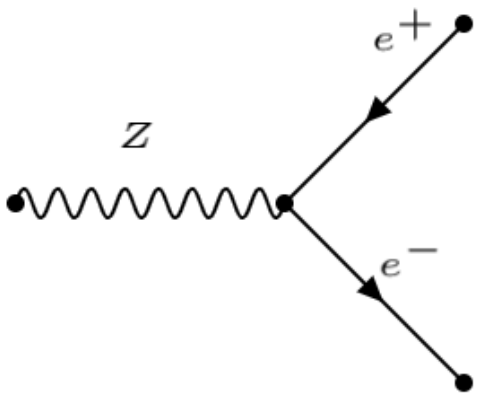

Fig. 2: Feynman diagram of the Drell-Yan process.

After a while of testing we realized that the $t\tilde{t}$ data set was to complex for our approach. We therefore also used an additional data set of Drell-Yan processes where a Z boson, created again by quark anti-quark annihilation in pp collisions, decays into an electron-positron pair. Having only two final state particles reduces the dimensionality to 8, given that we have again momentum conservation perpendicular to the beam direction as well as energy conservation, we can reduce this further to 4 dimensions. Also using that the global phase of the process is irrelevant as it is uniformly distributed, no direction in the xy-plane is favored, we can even reduce this to 3. Having such a simple data set helps in testing different architectures without having to train the network for sometimes even days before getting any results. Even though this process is so much simpler than the $t\tilde{t}$ decay, it still features a hidden Breit-Wigner peak in the Z mass distribution which means that we can test our full machinery developed to get such masses right here as well.

2.3 ZW Production

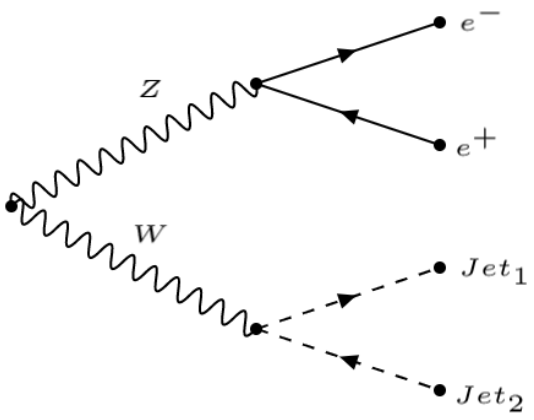

Fig. 3: Feynman diagram of the ZW process.

Our third data set consisted of ZW pairs also produced at the LHC. In this process, there is first a single W produced from (most of the time) quark anti-quark annihilation. The off-shell W then splits into a ZW pair, where the Z then subsequently decays into an e^+e^- pair like in the Drell-Yan, and the W decays into a pair of jets, as in the $t\tilde{t}$ data set. Like that we had three possible processes at hand, a very easy one with the Drell-Yan process, a pretty hard and high dimensional one which has been used for benchmarking various event generation frameworks in the past, the $t\tilde{t}$, and with the ZW an additional process which lays in between the previous ones in terms of complexity, with two hidden particle masses and a 12 dimensional phase space (already using energy conservation).

2.4 Parametrization

While our data sets were given as four-momenta (E, p_x, p_y, p_z) of the final state particles, it can be beneficial to change the basis and we therefore introduce some more common choices of phase space coordinates: Choosing the direction of the colliding beams to be the z direction, the part of the momentum which is perpendicular to the beam axis then becomes

$$p_T = \sqrt{p_x^2 + p_y^2} \tag{1}$$

called the transverse momentum. We call the angle between beam and particle θ and the angle between the x axis and the particles momentum projected on the xy plane ϕ. The pseudo rapidity is then given as $\eta = -\ln\left(\tan\frac{\theta}{2}\right)$. We can exchange the energy with the particles mass and obtain e.g. (m, p_T, η, ϕ) as coordinates. The choice of parameters also has an influence on training performance and we will test whether using four momenta, (m, p_T, η, ϕ) or a mix like (m, p_T, p_z, ϕ) as basis works best. As the final state particles have given fixed masses, we do not use them as inputs but instead reduce the coordinates given to the network to (p_T, η, ϕ) etc.

3 Machine Learning

Computer programs which are built to get better by learning from provided data are summarized under the term machine learning (ML). This class of programs can be further distinguished in supervised and unsupervised learning. While in the case of supervised learning the program not only gets the input data but also the expected output, unsupervised learning only provides the unlabeled data to the computer. As we only impose a fixed prior distribution on the output, but do not give unique labels to the data, our approach belongs to the unsupervised learning strategies.

3.1 Neural Networks

Nowadays many ML applications are implemented using neural networks. Those are built upon many neurons which take an input, perform very basic functions on it and output the result. While those single neurons are pretty limited in what they can do, the combination of multiple neurons can become a powerful tool to map pretty much any input data to any desired output[15]. Most of the time the multiple neurons are structured in layers with transformations in between which distribute the output of the previous layer onto the inputs of the next layer. There are multiple types of transformations in use like convolutions or up-sampling, which are often used for e.g. image processing, but in our application we only use basic dense layers, which are connected using affine transformations. The neurons themselves only apply the so called activation function to their input. They need to be non linear as otherwise the whole network would only be a combination of the linear transformations and therefore the network as a whole would be just a linear transformation, rendering the multi layer approach useless. A basic example of an activation function is the rectified linear unit (ReLU):

$$\mathrm{ReLU}(x) = \max(0, x) \tag{2}$$

Other popular activation functions are the leaky ReLU, softplus and sigmoid functions. A comparison of their graphs is given in fig. 4.

3.2 Network Training

The parameters (or weights) of the transformations are first initialized. While there are many different methods out there to choose the initialization[16], in this thesis we stick to the default version of the pytorch package, which simply draws the parameters of a linear layer from a uniform distribution in the interval $\left(-\sqrt{\frac{1}{n_{in}}}, \sqrt{\frac{1}{n_{in}}}\right)$ with n_{in} the number of inputs of the given layer. During training, the network then changes the weights in order to minimize a so called loss which is computed using the network output and, in the case of supervised learning, the known true output. Possible losses are for example the mean squared distance between N true outputs $\hat{z}_i$ and the network outputs z_i,

$$\mathrm{MSE}(\hat{z}, z) = \frac{1}{N} \sum_i (\hat{z}_i - z_i)^2 \tag{3}$$

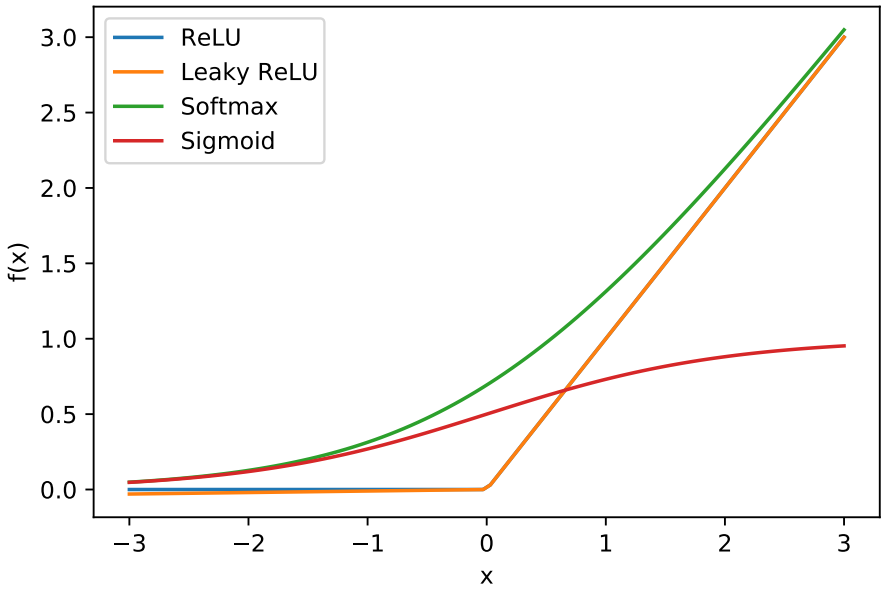

Fig. 4: Comparison of different activations.

In our studies we use the Kullback-Leibler Divergence as a loss to measure how far the distribution of the network output varies from the prior distribution we impose on the output space (in our context also called latent space). Given probability distributions p and q, the Kullback-Leibler Divergence is defined as:

$$D_{KL}(p,q) = \int_X p(x) \log\left(\frac{p(x)}{q(x)}\right) dx \tag{4}$$

Where the integration is done over the underlying space X, in our case this would be the final state phase space of the process at hand. Note that this formulation in general is non-commutative regarding the distributions: $D_{KL}(p,q) \neq D_{KL}(q,p)$. Using $D_{KL}(p_{true}, p_{gen})$ as loss, minimizing this term becomes equivalent to maximizing the log-likelihood of the generated data. Using the change-of-variables formula, we can calculate this directly on the latent distribution:

$$\log(p_{gen}(x)) = \log(p_{latent}(z(x))) + \log(J) \tag{5}$$

where J denotes the jacobian of the network. For a standard normal distribution on the latent space, this simply becomes

$$\log(p_{gen}(x)) = \frac{-z(x)^2}{2} + \log(J) + const. \tag{6}$$

During training we then pass a batch of input data through the network, calculate the loss for each output and average over them. Using the torch.autograd package we can calculate the gradients of the networks weights w.r.t the loss, which basically tells us which change to the weights might result in a lower loss for the next iteration. The weight updates are done using an optimizer, which takes the current weights

w and the gradients ∇w to calculate the new weights w'. In its most simple form, the update can be performed as:

$$w \to w' = w - \tau \nabla w \tag{7}$$

This specific method is called gradient descent and is the foundation of most other, more involved, optimizers[17]. The parameter τ is called the learning rate and determines how much the weights change in one iteration. Using this simple procedure can be pretty slow, as the gradients are taken over all training set points and also the gradients can change much between different iterations, making the weight values fluctuate instead of converging towards the local optimum. Those problems can be solved by computing the gradient only on subsets of the training set called batches and by introducing a momentum which also takes into account the previous gradient, which makes the gradients and therefore the training more stable[18]. In this thesis we use the ADAM[19] optimizer which adds some changes to the momentum approach to make the training even faster. Additionally, one can choose different behaviors for the learning rate: While a large learning rate might at the beginning be very useful to not get stuck in a bad local minimum of the loss right from the start, later on during training a large learning rate might prevent the loss from converging closer towards the found minimum, as the update steps can then be larger than the distance to the minimum, making the weights shoot over their optimal values. During training this can be seen as a plateau in the loss value. A simple approach is to reduce the learning rate by a fixed percentage after a given amount of epochs. Another common approach is to reduce the learning rate automatically when the loss plateaus. We chose to use the one-cycle learning rate scheduling during our trainings, which first raises the learning rate and then lets it fall off again to a much smaller value. We chose to start at a learning rate of $\frac{1}{25}$ of the maximal learning rate, and end with a learning rate only $\frac{1}{10000}$ of it. This kind of learning rate scheduling can speed up the training convergence significantly[20].

3.3 Invertible Neural Networks

For the loss introduced above, it is already clear that our network architecture needs a tractable, fast to compute log-jacobian. To really use the trained network afterwards, we also need to be able to compute the inverse of the network equally fast. The basic architecture of dense layers with ReLU activations does not full-fill those requirements, as ReLUs are neither invertible nor have a well defined derivative at 0. While other activations like the softmax do not have those problems, they are still not a good choice for an invertible network as their jacobian is expensive to calculate. We solve this issue using coupling blocks. The basic idea here is to split the input of a layer up into two parts. While the first part gets through the block unaltered, we perform a transformation on the second part. The parameters θ of this transformation f are chosen based on the first part using a dense network:
For f we can use any differentiable and invertible function like an affine function $f : x \to x' = Ax + b$ with parameters $\theta = (A, b)$ from the dense network. It is easy to see that this block can be used backwards without the need of inverting the dense

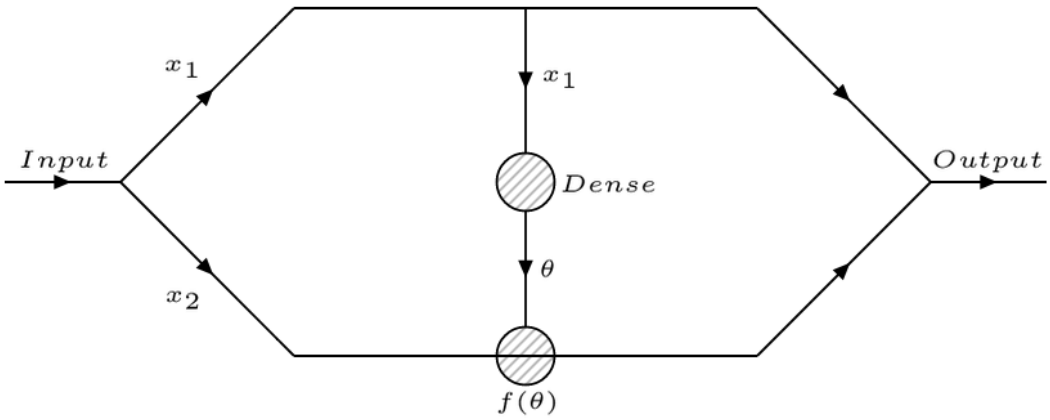

Fig. 5: Coupling Block

network, just the transformation f needs to be inverted. Additionally, the jacobian is independent of the dense network and for simple functions like polynomials it is easy to calculate. As the upper half of the input is unchanged by such a block, we also need to add many blocks up with some mixing of the input dimension in between. This can be done by eg. just permuting the output vectors or by using a rotation matrix. For our networks we always use the rotation matrices. Both transformations are again easily invertible and have a tractable jacobian. To implement the INN architecture, we rely on the FrEIA framework by the Visual Learning Lab Heidelberg[21], with some minor changes to improve stability. In this thesis we will primarily use coupling blocks where the transformation is built of a cubic spline, which is just a piece wise cubic function with differentiable transitions between the pieces[22]. How those splines interpolate between given data points can be seen for some example functions in fig. 6.

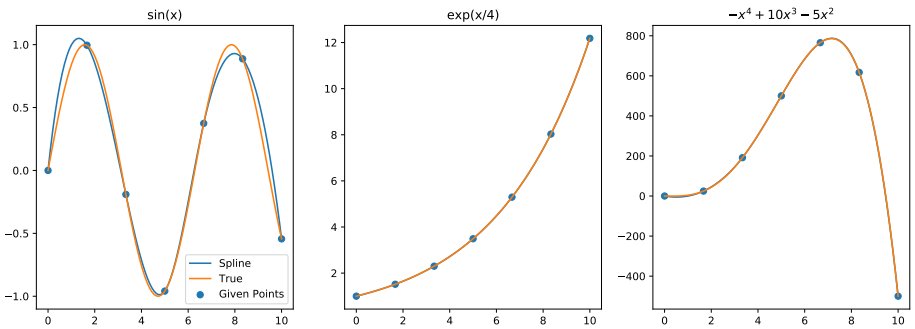

Fig. 6: Spline interpolation for some example functions, using 7 data points each for the fit.

3.4 GANs

A common strategy for training a generator is doing so by letting it compete against an additional discriminator which tries to find the flaws in the generator outputs. This setup is commonly called a generative adversarial network (GAN) and is normally used with a standard dense network as generator. Those GANs have already been used for LHC event generation[23], image super-resolution[24] or image generation[25] to only name a few applications. The discriminators goal is then to differentiate between generator output and true data, mapping true events to 1 and fake events to 0. In contrast, the generators goal is to make this task as hard as possible by trying to change the discriminators output for fake events to 1 as well. Those goals can be encoded in the loss functions of both networks. Based on N samples x_i from the generator output and samples y_i from the true data, we can write the losses as:

$$\begin{aligned} Loss_D &= -\frac{1}{N}\sum_i \log(1 - D(x_i)) - \frac{1}{N}\sum_i \log(D(y_i)) \\ Loss_G &= -\frac{1}{N}\sum_i \log(D(x_i)) \end{aligned} \tag{8}$$

The discriminators output layer is a sigmoid to restrict the output to $[0, 1]$. As the combination of sigmoid and the logs in the loss function cancel out some computations, we do not use the sigmoid layer when computing the loss. Instead we use a numerically more stable and slightly altered version of the loss function to accommodate the cancellations and which is already implemented in pytorch. In addition to making the generator learn the right distributions, the discriminators can also be used to reweight the generated data post-hoc. If the GAN is not in the Nash-equilibrium after training, where the discriminator can not discriminate anymore between true and fake events, we can therefore use the left over information in the discriminator to better the generators precision, however at the cost of getting weighted events. When using an INN as the generator with its own maximum log-likelihood loss, we have to couple both loss terms together:

$$Loss = Loss_{INN} + \lambda_{adv} Loss_G \tag{9}$$

3.5 Regularization

Neural networks in general, and the GANs discussed later on even more so, are prone to exploding weights and therefore unstable training. Excessively large weights can be due to overfitting to some minor details in the data distributions. Very large networks can also tend to "memorise" the training data instead of really learning the underlying structures, which can also lead to such large weights and bad performance on new data. To counter this, there are some common regularization which limit the growth of the weights:

- **Dropout:** During training, one can "shut down" random subsets of the neurons by setting their inputs to zero, which mostly prevents that the network clusters into sub-parts each memorising certain aspects of the training set.[26]

- **Weight Decay:** Another option is to reduce the network weights by some chosen percentage at each update step, which limits their growth and also reduces overfitting.[27]

- **Spectralnorm:** Interpreting the weights of an e.g. linear transformation between an n dimensional and an m dimensional layer of neurons, as an $n \times m$ matrix, one can divide this weight matrix by its spectral norm (which is its largest singular value) as an alternative way of regularizing it. This has been proven to bound the Lipschitz constant of the network to be smaller than one which is especially important for the discriminator when training a GAN[28], but it helps stabilizing the training also for other architectures.

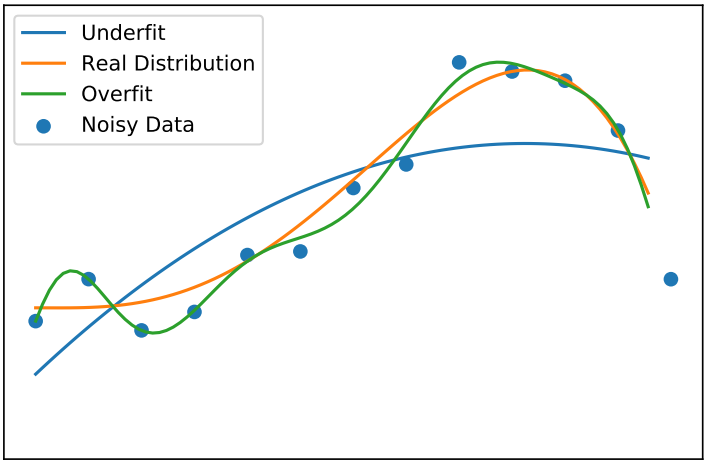

Fig. 7: Example of how a model can overfit or underfit to data points. In this toy example we sample points from the 4th order polynomial also shown in fig. 6 with some noise added and fit a simple quadratic (underfit) and a 10th order polynomial (overfit).

Those regularizations come however with the cost of lower expressiveness of the network, and they therefore need to be used carefully. E.g. a high weight decay can make training nearly impossible, and deactivating almost all of the neurons during training is obviously not a good choice either.

4 Preprocessing

Using the (p_T, ϕ, η) vectors as direct network input is a bad idea as all of them are in different intervals. For example the ϕ distribution is a uniform distribution between $-\pi$ and π which comes with steep edges at the borders. Those steep edges are generally hard for the network to learn, and we can speed up the training process by instead feeding $\tanh^{-1}\left(\frac{\phi}{\pi}\right)$ into the network, a comparison between the preprocessed and unpreprocessed pdfs can be seen in 8.

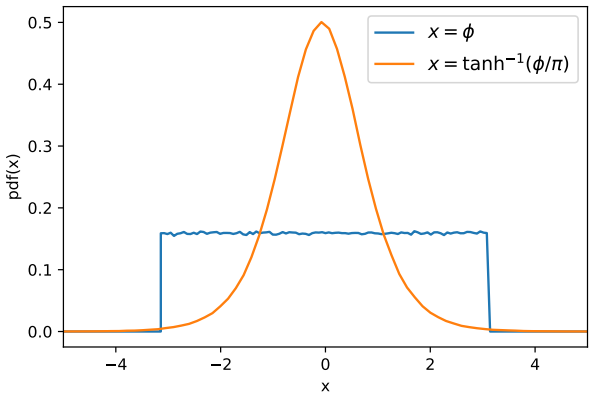

Fig. 8: Comparison between the unpreprocessed and preprocessed pdf of ϕ.

We also transform the p_T to $\ln(p_T - \min(p_T))$ which reduces the long tails of the p_T distributions and makes them more similar to a Gaussian distribution. Subtracting the minimal p_T value also has the added benefit, that the network does not have to learn cutoffs at low p_T values which are present in e.g. the Drell-Yan data. Those cutoffs are due to restrictions in the perturbative expansion used for calculating the training data, which fail for low momenta.

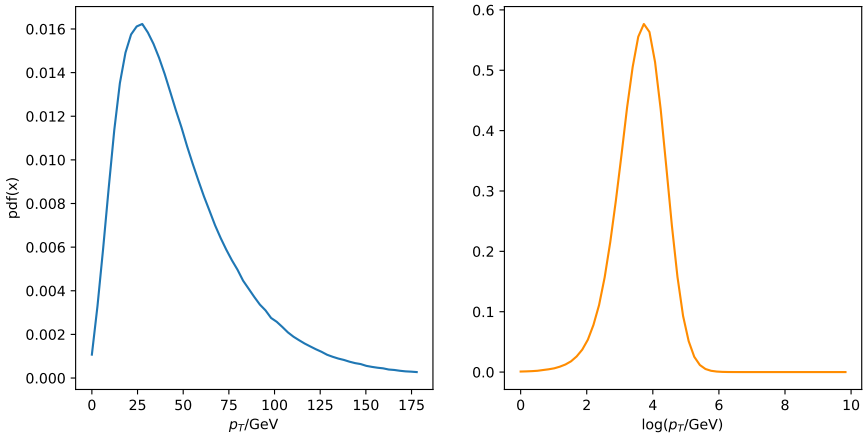

Fig. 9: Comparison between the unpreprocessed and preprocessed pdf of p_T.

Additionally we normalize the input by subtracting the mean and dividing by the standard deviation. Like that the input becomes already more similar to the latent Gaussian which also comes with a mean of zero and standard deviation of one. To reduce correlations between the inputs we also employ a whitening technique based on performing a principle component analysis on the set of input vectors. After this whitening step the covariance matrix of the input data should be close to the unit matrix. To demonstrate how well this works in a toy example, we sample from a Gaussian with randomly initialized covariance, perform the whitening and then plot both covariance matrices next to each other in fig. 10.

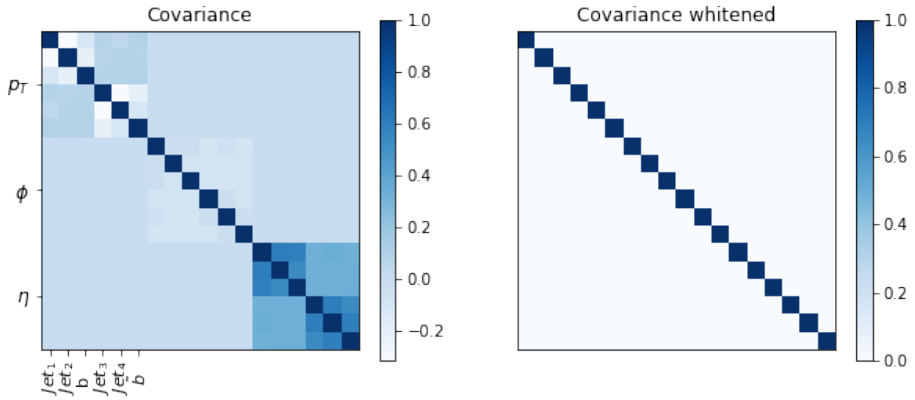

Fig. 11: Covariances before and after whitening for the preprocessed $t\tilde{t}$ events. Observable type is indicated on the y axis for both plots, the further splitting into the Jets and bs is indicated on the x axis.

Fig. 10: Covariances before and after whitening for a Gaussian with randomly initialized Covariance matrix.

We also plot the covariance for the preprocessed $t\tilde{t}$ data, and again the whitened covariance becomes a unit matrix as can be seen in fig. 11. In this case the non-whitened covariance matrix also gives us some insights on the given data: It is clearly separated into quadratic blocks, where the upper left belongs to the p_T distributions, the middle block to the ϕ and the lower left to η. Between those variable groups the covariance tends to vanish or at least becomes way smaller in comparison. The p_T and η blocks again split up in two sub-blocks, corresponding to the decay products of the t and $\tilde{t}$.

5 Event Generation

Using the previously described INNs, it becomes easy to generate new events by sampling from the latent space distribution and then feeding those samples backwards through the network. We then also have to undo the aforementioned preprocessing steps, but as those were just basic transformations this is a simple and computationally fast step. To make the high dimensional generated data accessible for examination, we marginalize over all dimensions but one or two and then plot 1D histograms for the resulting p_T, η, ϕ, etc. distributions and 2D histograms for the various correlations between pairs of them. We also look at the latent distribu-

tions to see how well they match the imposed Gaussians, again only regarding the 1D and 2D marginals.

5.1 Architecture

While we experimented a lot with different network architectures, learning rates, optimizers etc, for the presented runs we used the same standardized network setups with cubic splines, only the node amount and training durations were depending on the data set used:

Input Observables	$(p_{T,l_1}, \eta_{l_1}, \eta_{l_2})$
Optimizer	$ADAM$
Number of Blocks	10
Layers per Block	3
Nodes per Layer	128
Activation	ReLU
Bounds	10
Parameters to Optimize	330k
Spectralnorm	No
Weight Decay	0.0
Learning Rate	$2 \cdot 10^{-4}$ (OneCycle)
Batchsize	4096
Epochs	50

Tab. 1: Generator setup for the Drell-Yan data set.

Input Observables	(p_T, η, ϕ)
Optimizer	ADAM
Number of Blocks	24
Layers per Block	3
Activation	LeakyReLU
Nodes per Layer	128
Bounds	10
Parameters to Optimize	330k
Spectralnorm	No
Weight Decay	0.0
Learning Rate	$2 \cdot 10^{-4}$ (OneCycle)
Batchsize	4096
Epochs	500

Tab. 2: Generator setup for both $t\tilde{t}$ and ZW.

The discriminator we only used on the Drell-Yan and the ZW data set, and the parameters were chosen based on the amount of inputs:

Inputs	1	2	4	12
Optimiser	ADAM	ADAM	ADAM	ADAM
Layers	3	8	8	8
Nodes per Layer	64	128	128	256
Parameters to Optimize	4.5k	100k	100k	400k
Spectralnorm	No	No	No	No
Weight Decay	0.0	0.0	0.0	0.0
Learning Rate	$1 \cdot 10^{-4}$	$2 \cdot 10^{-5}$	$2 \cdot 10^{-5}$	$2 \cdot 10^{-5}$

Tab. 3: Discriminator setups for varying input dimensions.

6 Getting the Mass Right

While our network architecture is already capable of producing events with accurate p_T, ϕ and η distributions from the start, it has some issues with predicting the Breit-Wigner peaks in the mass distributions of the intermediate t quarks and $W^{\pm}$, Z bosons. As already mentioned this behavior is expected as those masses only appear as high dimensional correlations of the network inputs. E.g. in the M_W distributions, all information of the two daughter jets is combined which makes them a 6D correlation of those quantities. For the top masses this is even worse, as they only appear hidden behind such a W and an additional b. Most previous methods which used networks to generate $t\tilde{t}$ or similar events with intermediate particles also had those mass-related issues[29]. A baseline run of a plain INN without anything added to improve the masses is shown in fig. 12 and fig. 13 for the ZW data set.

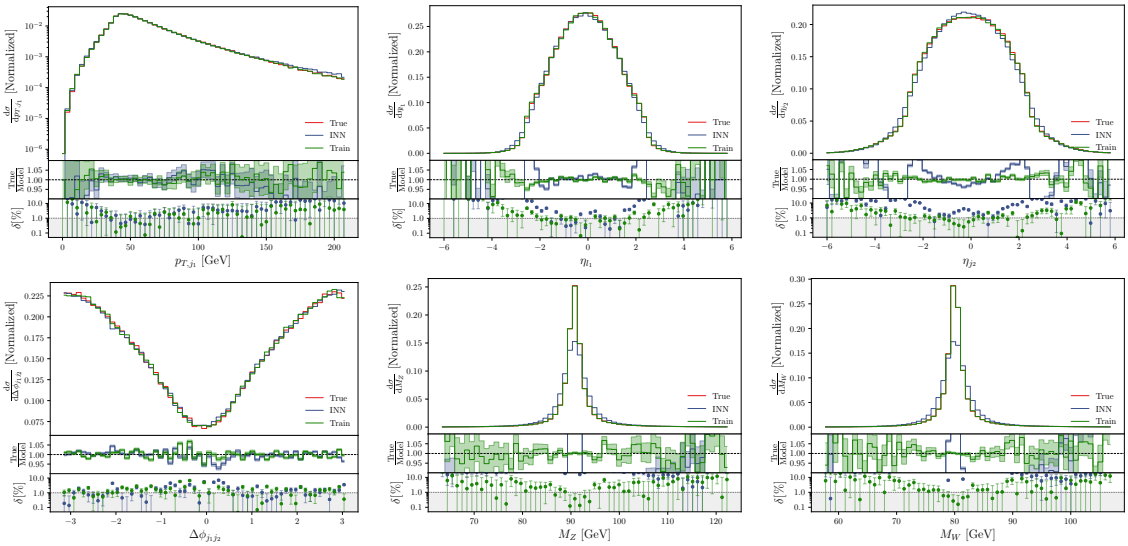

Fig. 12: Example distributions for a baseline INN run on the ZW data set. Upper panel: Distributions of the training set, the test set (label "True") and the INN output. Second panel: Binwise ratio between the test set and the generated and training set respectively. Below: Percentage deviation of generated and training from test set. While the relative error of most other distributions is already on the level of the training statistics, the mass distributions are clearly still too broad.

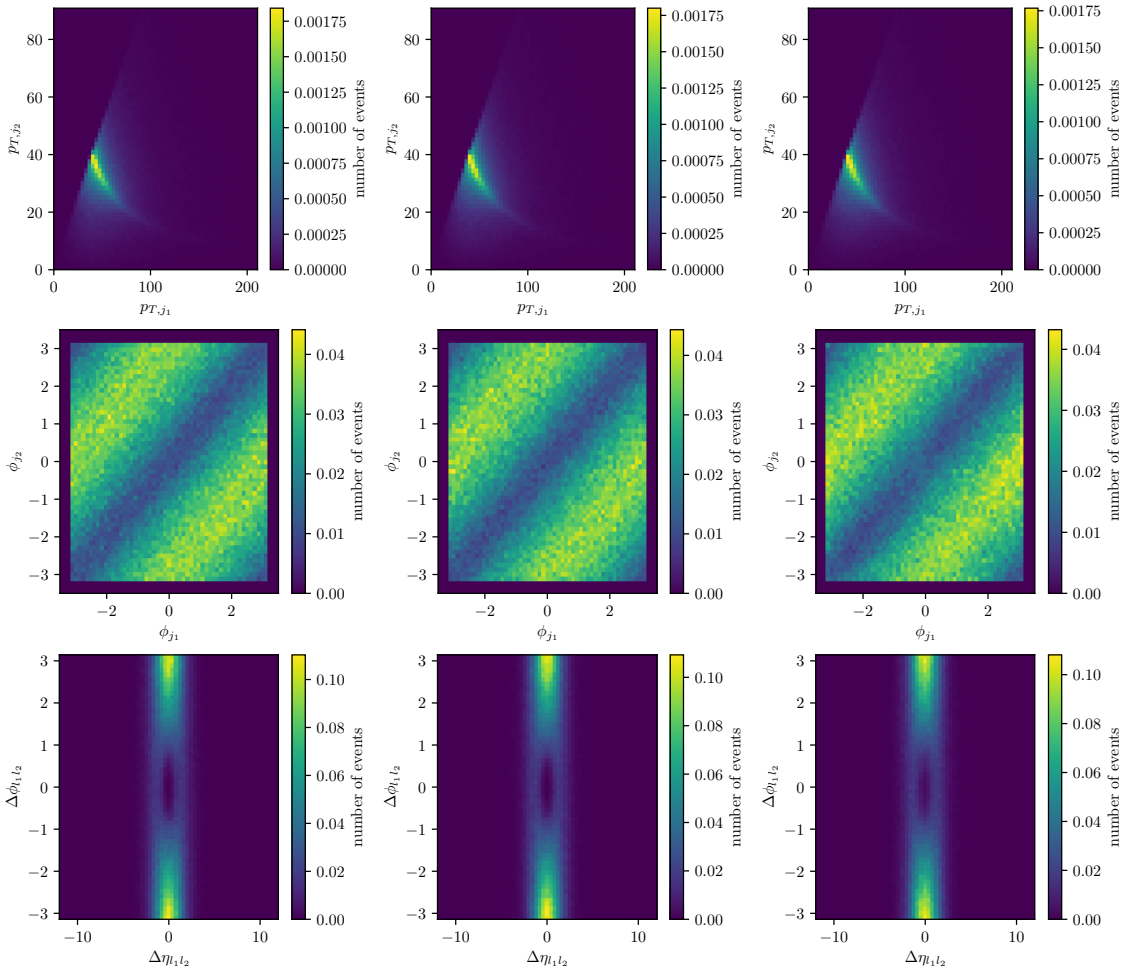

Fig. 13: Additional 2D correlations for the ZW baseline. From left to right, the panels are correlations on the test, training and generated data sets. Only the rings in the $\Delta\phi\Delta\eta$ correlations are in the generated set not as sharp as in the test and train sets.

6.1 The right parametrization

For our simplest data set, the Drell-Yan, choosing a parametrization which captured the symmetries of the process was enough to get masses which deviated from the test set only on the order of statistical fluctuations. We chose a single p_T and the two leptons' ηs. Having only two final state particles, the momentum conservation could therefore be implemented symmetrically, by using that both particles have to have the same p_T. Momentum conservation also implied that the particles need to be emitted back to back, so we could just set ϕ for one of them to 0 and for the other one to π. This procedure was however not so easily applicable to the other data sets, as the choice of picking a single jet, a b Quark or another lepton, and then using momentum conservation to get its p_x and p_y, introduced some asymmetry. For example fixing the momentum of the positron in the ZW data, the W mass actually got better, but the Z mass got worse as also the small errors in the jet momenta got added in the positrons momentum which in turn were reflected in the Z mass getting broader. Omitting the global uniform phase by turning e.g.

a jet to $\phi = 0$ introduced a similar bias, and had no effect on training anyways. We therefore did not use momentum conservation nor did we subtract the global phase for the ZW and $t\tilde{t}$ in the final results. As inputs we experimented with using (p_x, p_y, p_z), (p_T, ϕ, η) and (p_T, ϕ, p_z), and we found that the combination of (p_T, ϕ, η) with our above described preprocessing worked best. Looking at the distributions this is pretty much expected as, after preprocessing, all those observables were almost Gaussian shaped. The Cartesian momenta (p_x, p_y, p_z) however were not and we also had no simple preprocessing like for p_T or ϕ to make them "more" Gaussian. η has an easy shape from the start without any preprocessing needed. Training on η was however very unstable on the ZW data set, until we found out that this was just due to two single events in the training data which had an η of ∞ from the start due to rounding errors in the calculation of η. Removing those two points stabilized training and we could stick to (p_T, ϕ, η) also for the ZW. Regarding the whitening we employed, we saw that it had a positive effect on all the trainings, which showed mostly in better mass distributions. We expected this to happen, as the masses only show as high dimensional correlations between the four momenta, and the whitening absorbed some of these correlations which the network therefore did not have to explicitly learn.

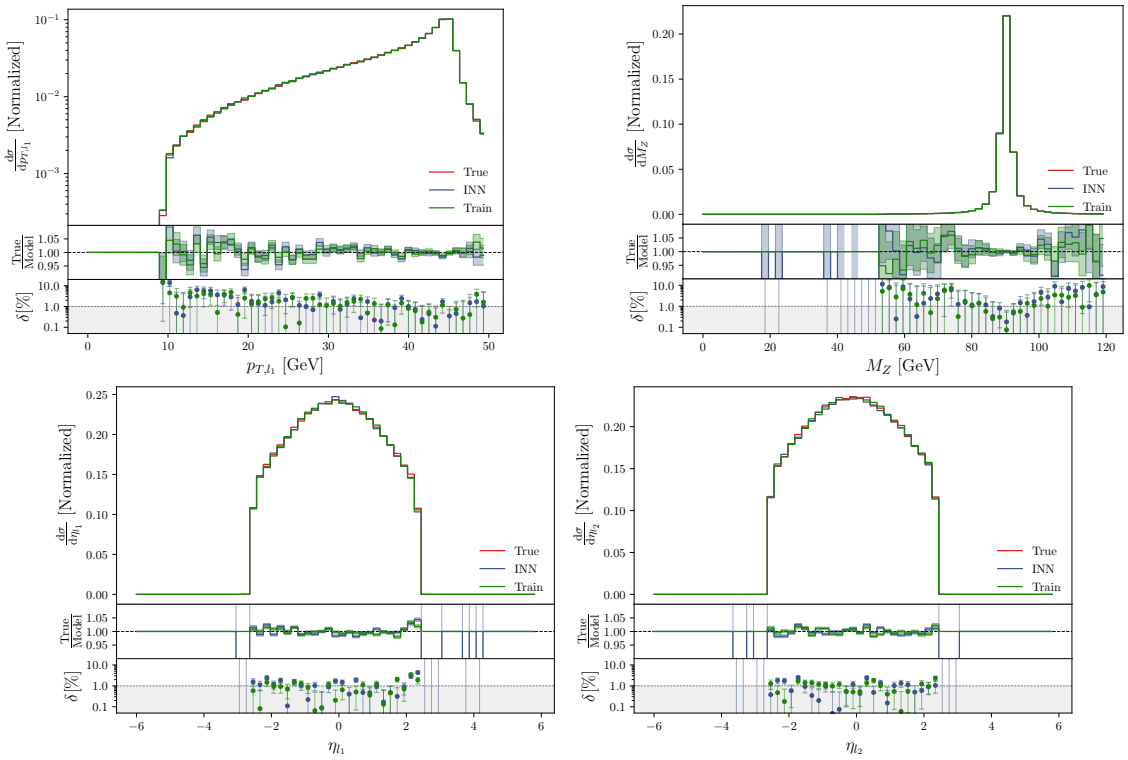

Fig. 14: Observable Distributions for the Drell-Yan process.

For the ZW data set, we also compared having the two jets ordered after their p_T or not. While we expected the unordered inputs to yield better results, as this would retain the symmetry between both jets, we found out that in fact the p_T ordering actually helped the network. On a second thought this makes sense, as

the ordering made some minor correlations between high p_T and the η distributions easier to learn for the network, which would otherwise be hidden in some higher dimensional correlations instead of the the 1D distributions of the jet ηs. In fig. 15 we show those differences between the η distributions for the p_T ordered jets.

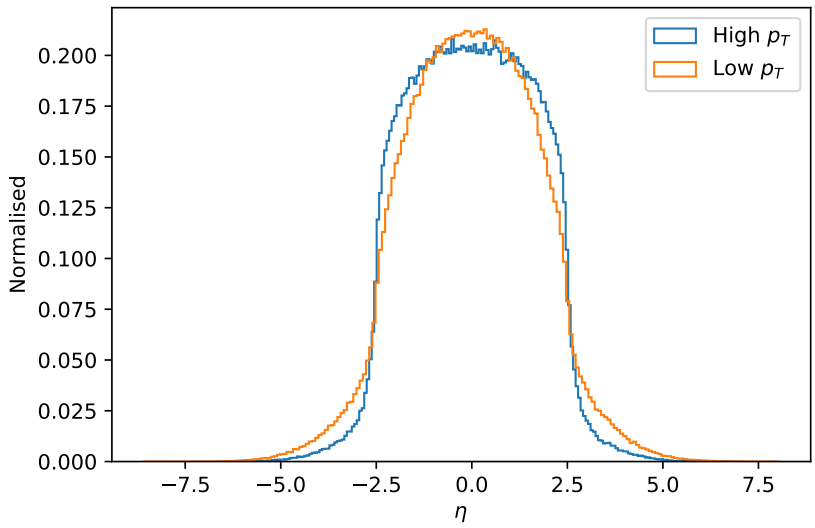

Fig. 15: Differences in the jet η distributions for p_T ordered jets. In all other plots, the jet with higher momentum is labeled jet_1, the jet with lower p_T is labeled jet_2.

6.2 MMD-Loss

A common tool to measure the discrepancy between different distributions is the so called maximum mean discrepancy (MMD). Given N samples x_i from a distribution and M samples y_i from another, the squared MMD is approximately calculated via:

$$MMD^2 = \frac{1}{N^2}\sum_i\sum_j k(x_i, x_j) + \frac{1}{M^2}\sum_i\sum_j k(y_i, y_j) - \frac{2}{N*M}\sum_i\sum_j k(x_i, y_j) \tag{10}$$

Here k denotes a kernel function we apply to pairs of samples. The kernel function allows us to set a scale, using the kernel width σ_k, on which the two distributions are compared. Common choices of kernel functions are the Gaussian kernel and the Breit-Wigner kernel. They are given as:

$$k_{Gaussian}(x, y) = e^{\frac{(x-y^2)}{\sigma_k^2}} \tag{11}$$

$$k_{Wigner}(x, y) = \frac{\sigma_k^2}{(x-y)^2 + \sigma_k^2} \tag{12}$$

The scale becomes crucial during training as we need the gradients of the MMD to train our network: When the generated and true distributions are at first completely different, we need a large kernel width to resolve those differences. Later on

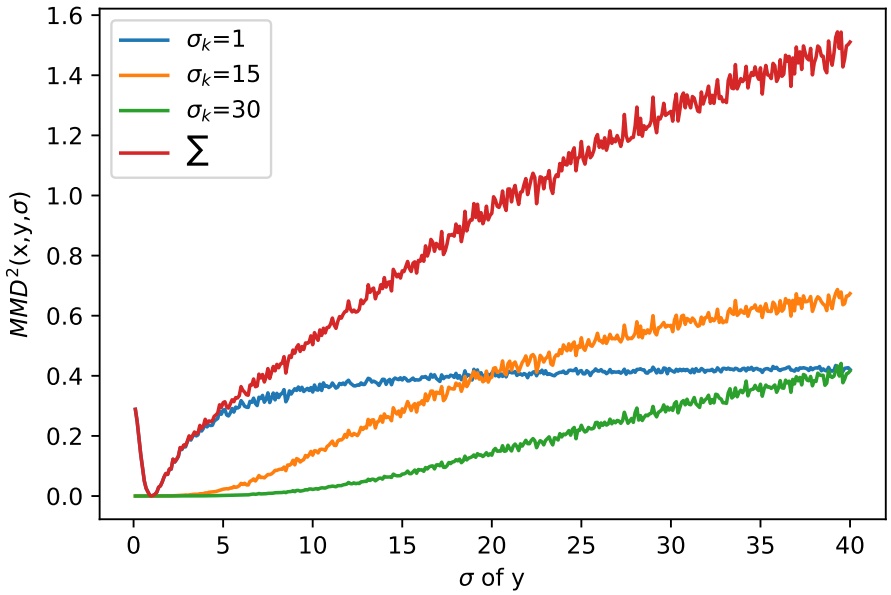

Fig. 16: MMD^2 for different kernel width σ_k of a Gaussian kernel and varying width of the Gaussian input data, where one distribution is fixed to a width of one and the other width is noted on the y axis. We choose 4096 data points which matches the batch-sizes used during the network training.

during training when the fake distribution has further converged towards the real distribution, a small kernel width is needed to resolve the now smaller differences between the two distributions. To visualize the differences, we created data points x_i sampled from a standard normal Gaussian. The y_i data points were sampled from Gaussians with different widths σ_y and we used a Gaussian kernel for this toy model. We then plotted the MMD^2 value as a function of the width of the y_i distributions in fig.fig:mmd.

For different kernel width, the results varied greatly: For large σ_k, the MMD^2 had no minimum at $\sigma_y = 1$ where the two distributions are the same. During training, this means that σ_k much larger than the width of the true data would push the fake distribution too narrow. Choosing $\sigma_k = \sigma_x$ on the other hand has a steep minimum at $\sigma_y = \sigma_x$, but if the fake distribution is way broader than the true distribution, the MMD^2 does not change much when varying σ_y which means that when our fake distribution initially is very flat, the MM^2 does not generate a large gradient towards the true distribution. We therefore also added the kernels with different widths up, shown in fig. 16 with label "$\sum$". All kernels combined now have non vanishing gradients for flat fake distributions and still remain the minimum at $\sigma_y = \sigma_x$. While this toy model is a huge simplification in comparison to the real problem at hand which does not have a fake distribution with fixed shape and σ_y as its only parameter, this model can still give us an insight on how to choose the kernel widths during training:

- **Fixed, multiple kernels:** Our first approach was to use multiple kernels added up which was also shown in the toy model plot. We chose the scales

such that σ of the first kernel matched the true width of the Breit-Wigner distributions of the W and t masses which can be calculated from theory. For the second kernel we chose σ to match the initial width of the fake distribution which we got from printing it out during a normal training run without MMD enabled. This ensured good gradients on all scales between the initial width and the true width. While we plotted three different kernels added up in the toy model, we found out that two where enough to retain gradients everywhere and therefore we sticked to using two kernels only.

- **Variable scale:** Next we tried changing the scale during training, by calculating the current width of the fake distribution and adapted the scale accordingly. Like that only one kernel was enough to get good gradients for broad fake distribution as well as for fake distribution at the end of the training which already matched the true distribution more closely. To estimate the current width we tried fitting a Breit-Wigner distribution to the data as well as reading the width off of the data using the distance between the $\frac{1}{4}$ and $\frac{3}{4}$ quantiles. We also tried an estimation free version, were we chose an exponential decay for the width with fixed decay rate.

Additionally, we tried schedulers for varying the λ_{MMD} which couples the MMD term to the normal latent loss. Our reasoning here was that at the beginning, when the masses, and all other observables as well, were still really bad, the MMD should not contribute or only slightly, while later on when the other observables where already learned, the MMD term could then be activated to push the network towards better mass distributions. The schedulers we used are:

- **0 to const:** The first simple approach was to just choose an epoch and only add the MMD after this epoch, giving the network time to learn the other distributions and then activating the MMD at once.

- **tanh:** We also tried slowly rising the λ_{MMD} from 0 up to a predefined value over multiple epochs, using $\lambda_{max} \cdot \tanh\left(\frac{epoch}{epoch_{max}}\right)$ to interpolate between vanishing λ_{MMD} at the beginning and λ_{max} later during training.

- **Reference Distribution:** As the MMD had a huge impact on the other observables, making them way worse, we added a reference distribution like $p_{T_{jet1}}$ and measured how far this distribution varied from the true data. When the difference was to big, we lowered the λ_{MMD} until the network learned the reference distribution again well enough and then we raised λ_{MMD} back up. We also tried to couple it instead to the latent loss as a reference value, meaning high latent loss $\rightarrow$ low λ_{MMD}, low latent loss $\rightarrow$ high λ_{MMD}.

We experimented with many different combinations of the aforementioned widths and λ schedulers, different learning rates, different amount of epochs, batch-sizes, network depths etc, but all of them had in common that they only produced one of two different results:

- **Good masses, bad** (p_T, η, ϕ) Choosing a large λ_{MMD}, the network became exceedingly good at predicting correct mass distribution, but at the same time completely destroyed all other observable distributions. Example plots for this behavior can be seen in fig. 18

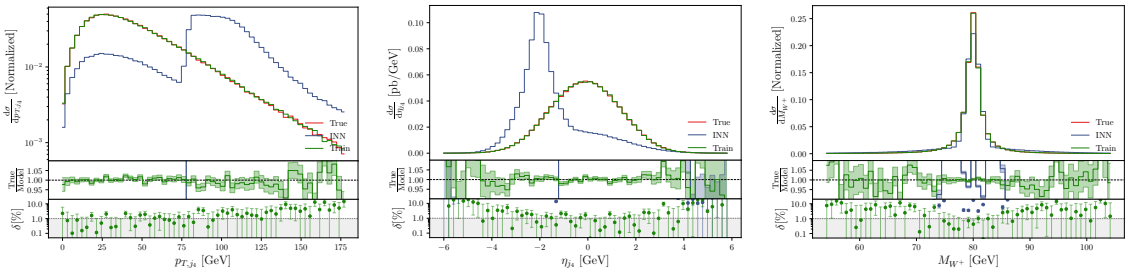

Fig. 17: Example distributions for a training with large $\lambda_{MMD} = 10^{-2}$ on the $t\tilde{t}$ data set.

- **Good** (p_T, η, ϕ)**, bad masses:** On the other hand, λ_{MMD} too low or strong coupling to a reference distribution which pulled λ_{MMD} down during training had the opposite result: The MMD did not change the result at all, (p_T, η, ϕ) stayed the same as well as the masses.

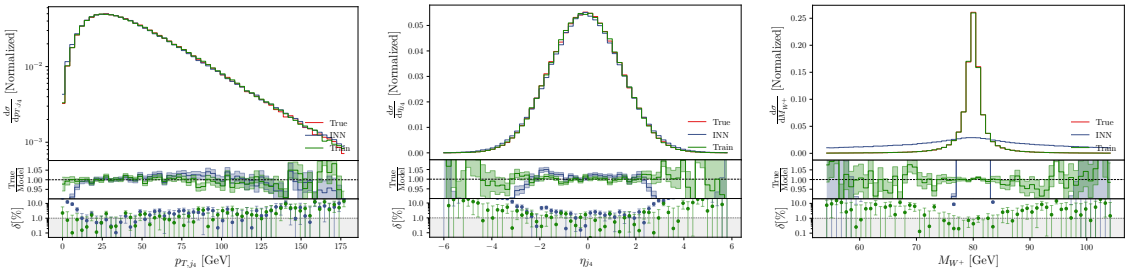

Fig. 18: Example distributions for a training with small $\lambda_{MMD} = 10^{-5}$ on the $t\tilde{t}$ data set.

Even though we thoroughly searched for hyperparameters which resulted in a balance of good masses and good (p_T, η, ϕ) at the same time, somehow we did not find them. Our assumption is that the MMD pushes the loss towards a different local minimum at which the other observables are very distorted and there is no intermediate loss region in between at which all observables are good.

6.3 Wasserstein

Another approach for getting the intermediate masses right, akin to using an MMD term, is to instead use the Wasserstein metric as a measure between true and fake distributions and then feeding this number back into the network. It has the added benefit of being interpretable as the minimal distance which events have to be shifted to make the two distributions equal. In comparison to shifting dirt from one pile to another in the most effective way, it is also called the earth mover distance (EMD). While the EMD becomes computationally unfeasible in higher dimensions, in 1D it actually only involves sorting the samples and is therefore of

order $\mathcal{O}(N \log N)$ in the batch-size N, whereas the MMD is $\mathcal{O}(N^2)$ as we there had to calculate all N^2 differences between true and fake events. The EMD of order p in 1D is approximately given for sorted samples as:

$$\mathrm{EMD}_p(X, Y) = \left(\frac{1}{N} \sum_i (x_i - y_i)^p \right)^{\frac{1}{p}} \tag{13}$$

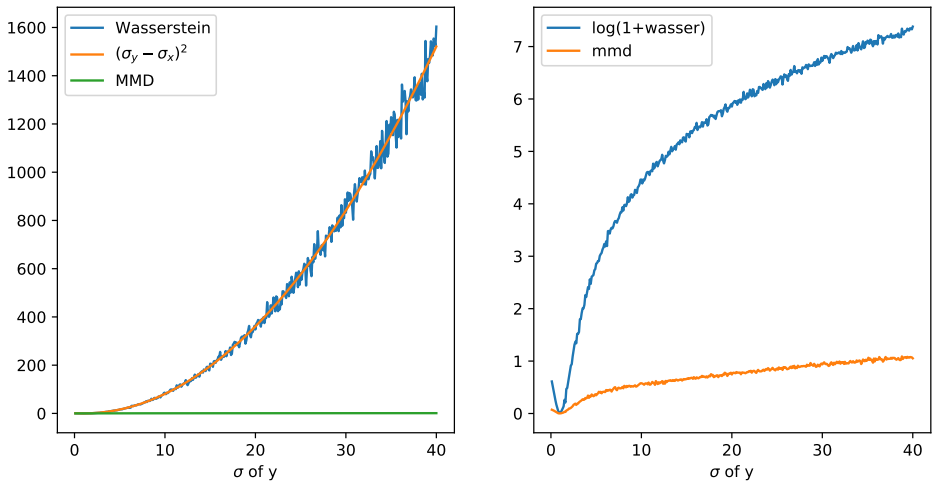

Fig. 19: EMD and $\log(1 + \mathrm{EMD})$ for varying width of the Gaussian data, where one distribution is fixed to a width of one and the other width is noted on the y axis. For reference, also the MMD^2 is plotted using kernels with width 1 and 30 added up.

Using the same plots as used to compare the gradients of different MMD kernel sizes, we can also see that the EMD produces larger gradients on all scales. In fact it even produces too large gradients which make the training unstable and we therefore also tryed $\log(1 + \mathrm{EMD}_2)$ as loss term to regularize the steep EMD at the beginning of training, where the two distributions have a large separation. The EMD of the Gaussian toy distributions can actually be calculated in closed form to be $(\sigma_x - \sigma_y)^2 + (\mu_x - \mu_y)^2$. We plot this on top of the EMD we get from the samples and the curves match well on the scale of statistic fluctuations of the samples.

Again this method only produces the two results already mentioned in the discussion of the MMD. Since the MMD and EMD approaches are fairly similar, this behavior was expected. Nevertheless we checked this approach to rule out some implementation errors we could have had in the MMD.

6.4 Discriminator

We next introduce an additional discriminator like in a GAN setup: At first we only train the generator as before, without any discriminator or MMD input and only afterwards train a discriminator on the generator output while keeping the generators weights fixed. Training the discriminator solely on the intermediate masses

and then reweighting the events as described yields already better distributions of those masses, as can be seen in fig. 20.

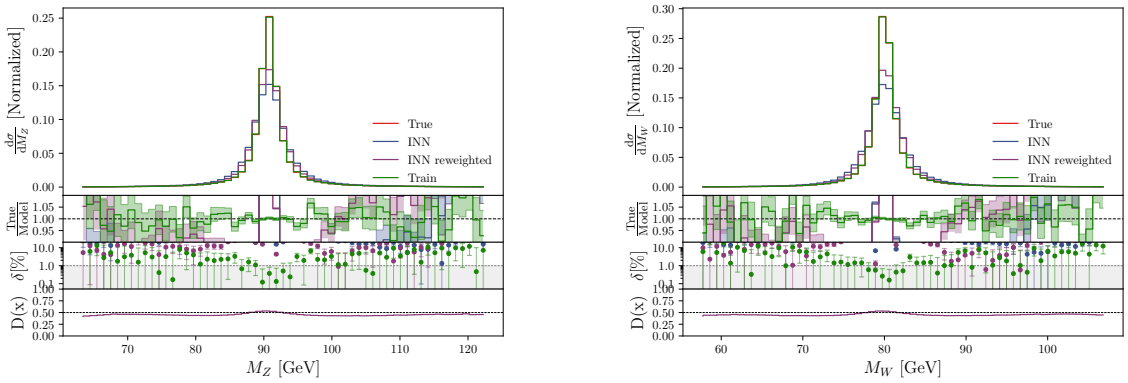

Fig. 20: Reweighted mass distributions for the baseline run in fig. 12, discriminator trained on M_Z and M_W.

An important step to enhance the reweighting was to introduce some additional preprocessing of the discriminators input: Normalizing the M_W and M_Z distributions boosted our reweighting precision a lot, shown in fig. 21, 22

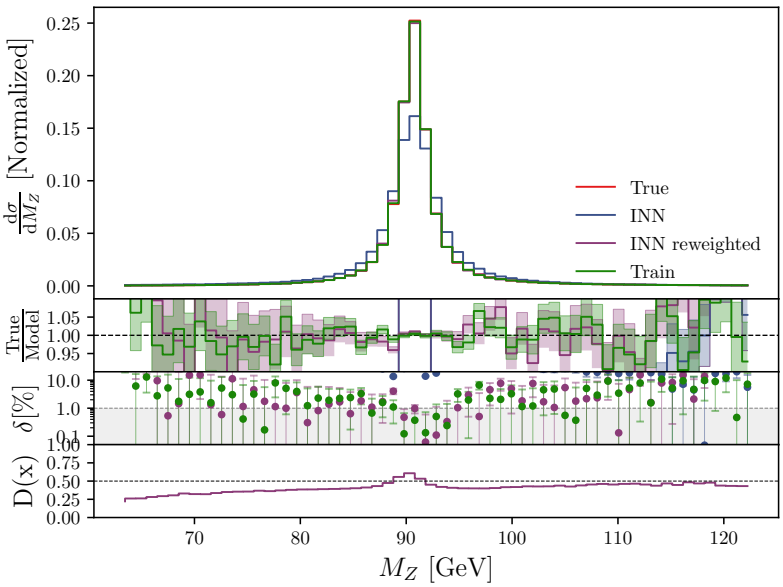

Fig. 21: Example plot for the reweighted M_Z distribution, training was exactly the same as in fig. 20, but with normalized masses as discriminator input. The reweighted curves are hardly visible as they overlap with the True and Train distributions.

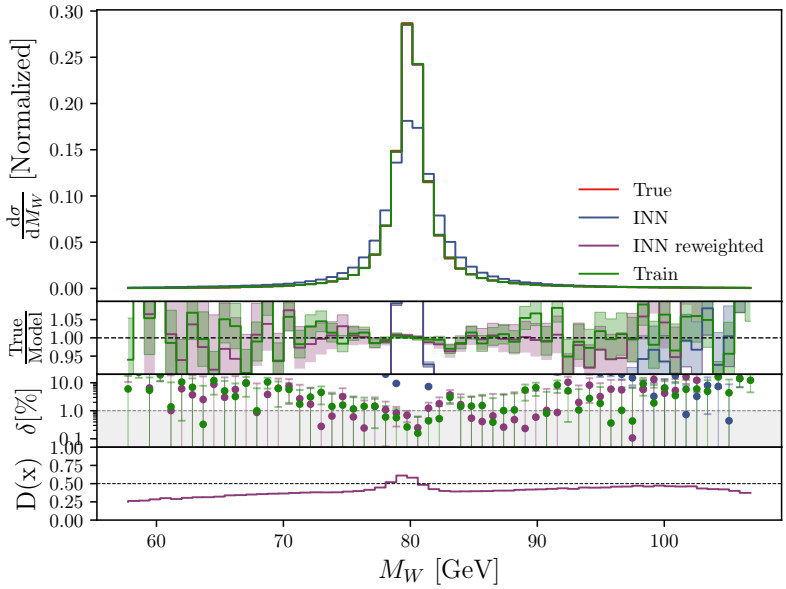

Fig. 22: M_W for the same run as in fig. 21

Using this reweighting, we can get the mass distributions for the ZW up to an error of around only 1%. It is notable that training the discriminator only on the masses also resulted in a precision increase of the other observables, most prominently in the p_T tails, shown in fig. 23

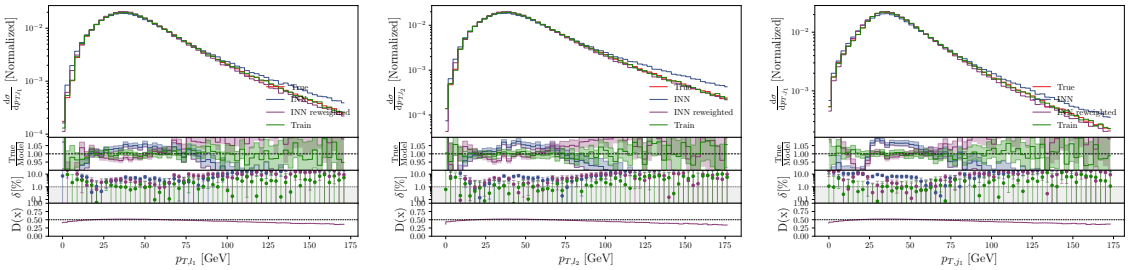

Fig. 23: Reweighted p_T tails of the baseline run, with the discriminator only trained on the W and Z masses.

Afterwards the weighted data we produce can then be unweighted via an additional neural network[30] or by standard unweighting techniques like rejection sampling. As our generated data is already close to the real data, the weights only span a range of around 0.3 to 1.7 which would result in a rather efficient unweighting. The main goal here is however to directly produce unweighted data and we therefore feed the discriminator output back into the the generator using the GAN loss (8), with a coupling λ_{adv} to control the balance between the maximum likelihood and the adversarial loss terms. Training the discriminator solely on the masses again results in the scheme known from the MMD, where either the adversarial loss term does not contribute at all or it helps producing precise mass peaks, but on the cost of worse p_T, ϕ etc. depending on the value of λ_{adv}. In the region between good masses and good other observables, the training simply becomes unstable with some runs resulting in good masses and bad other observables,

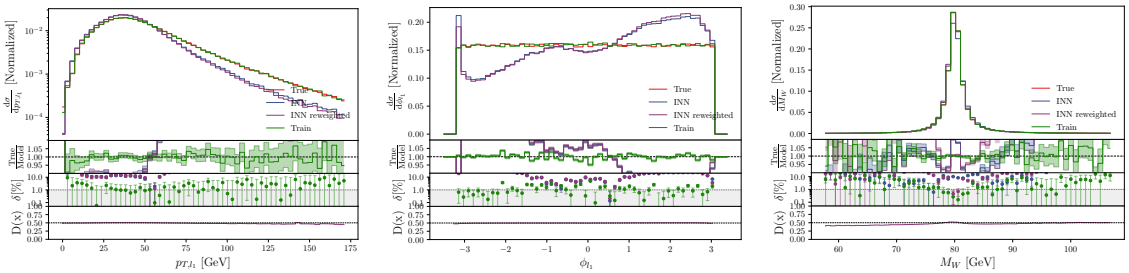

Fig. 24: Example distributions for a training with large $\lambda_{adv} = 1$ on the ZW data set, discriminator trained on both masses.

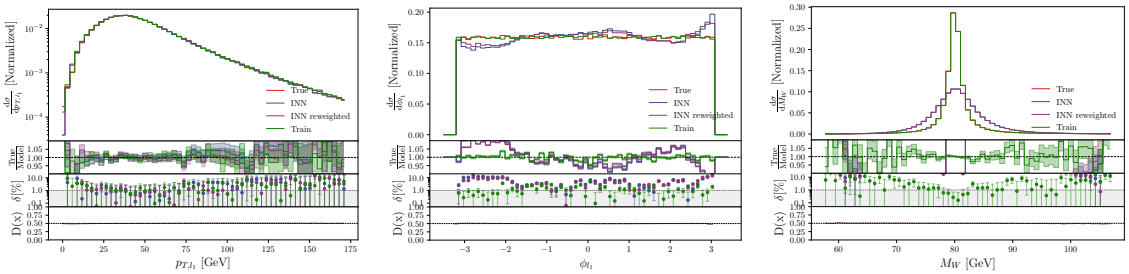

Fig. 25: Example distributions for a training with $\lambda_{adv} = 1$ on the ZW data set, discriminator trained directly on the discriminators output without undoing the preprocessing.

while other runs with the exact same parameters result in bad masses and also bad other observables. However with the discriminator we also have the possibility to train it on other observables in addition to the masses, to make it learn the underlying correlations which produce the mass peaks as well, therefore also producing correct other observables. Additionally we tried feeding the generated, still preprocessed data directly into the discriminator instead of first undoing the preprocessing, calculating the observables we wanted to use like the masses, and then using an additional preprocessing before giving it to the discriminator. While both methods, adding some other observables to the discriminators input and inputting the raw generated data, reduce the artifacts in the p_T, η, ϕ distributions, they still overall reduce the precision of the network compared to the baseline run in fig. 12, especially in the mass distributions as shown in fig. 25. While this might just be a problem with some hyperparameters we have not fine tuned enough, it could also hint for a general incompatibility of the latent and adversarial loss terms, like we also suspect for the MMD and EMD.

As all previous attempts of MMD, Wasserstein and adversarial loss terms were all based on introducing the second loss term on the physical input side, rather than the latent space on which the log-likelihood loss was defined, we thought that the previous problems might all be tied to having different losses on both sides of the network. We therefore shifted our focus towards using the discriminators feedback instead on the latent space. First we just tried to train the discriminator on the latent space itself, to make it learn differences between the generated latent space and the imposed Gaussian, and then use the discriminator in the normal adversarial loss, but now on the same space as the log-likelihood loss. This approach did not work out, most probably because the differences on the latent space are way smaller

and harder to detect than the obvious differences between true and generated mass distributions on phase space. Next we tried to train the discriminator on the phase space again, but now we used its feedback as a weight on the latent space during training: By simply weighting each events latent representation by 1 over its weight w in the log-likelihood loss term, we made the generator focus more on the areas where it was still performing badly:

$$\mathcal{L}_{mod}(x)) = \frac{z(x)^2}{2w} - \frac{\log(J)}{w} \tag{14}$$

The generators goal of minimizing this modified loss function would then cause the discriminator weights to converge to one, which is also the goal when training it as a GAN, but without the need of introducing a second loss term at all. This approach yielded good results out of the box. Training the discriminator solely on the masses and using this feedback in the modified loss term resulted in sharper mass peaks compared to the baseline model in fig. 12, without distorting the other observables.

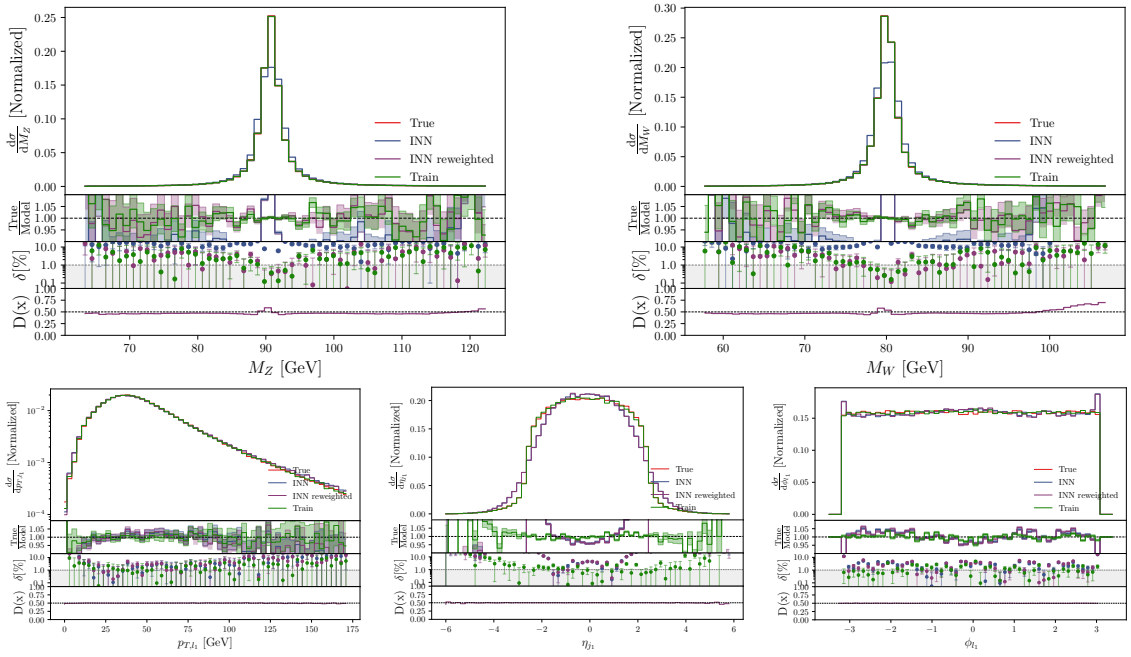

Fig. 26: Example distributions for a training with latent space reweighting.

While there are still some problems in the jet ηs and the ϕ distributions also have wrong boarders, those problems occurred in the baseline runs without the discriminator as well. We can most probably fix the η distributions in the future by training the discriminator additionally on the ηs. For these runs we can also see that the reweighting still adds precision, which means that there is still information left in the discriminator. Additionally looking at the loss curve for the training in fig. 27, we see that the training had not yet converged which means that there really is the possibility of further improvements with higher epoch counts, different learning rate etc. We will do the necessary testing in the near future and the results will be published in our upcoming paper.

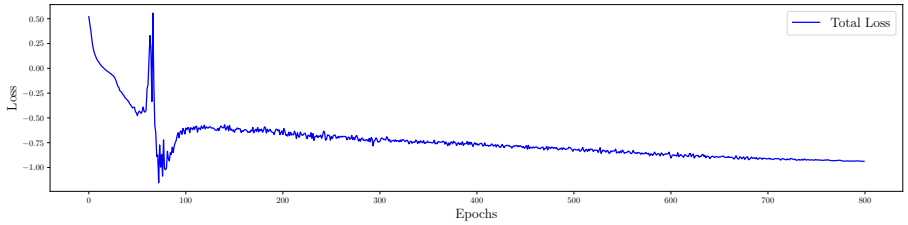

Fig. 27: Loss curve for the training with latent reweighting. Here 800 epochs correspond to training the generator for 400 epochs, as half of the training steps are used on the discriminator. While only declining very slowly, it is still visible that the loss has not yet plateaued and further improvements with more training time seem realistic.

7 Conclusion/Outlook

This thesis was basically all about studying whether an INN for event generation can profit from an additional loss function on phase space or not. While we have used various different losses like MMD, Wasserstein and adversarial, our final conclusion has to be that at least for our application, it is very hard to combine the different losses on different sides of the network in a constructive way if not impossible. We have not found a theoretical obstruction why this would be the case though, so it is completely possible that we just did not use the right hyper-parameters. For us it is at least plausible that the back-propagation of different losses from both sides of the network is the cause of this unstable behavior.

However using the discriminators information gathered on phase space in the latent loss function, instead of introducing an additional loss on phase space, worked really well. Combining our optimal parametrizations using (p_T, η, ϕ), our preprocessing and the discriminator we were finally able to generate:

- Weighted ZW events with an error of less than 1% in the bulk and errors comparable to the statistical fluctuations in the tails. It is also notable that the weights lay in a close range around 1, compared to Monte-Carlo sampling where the weights can span multiple orders of magnitude. This means that unweighting of our events could be done very efficiently compared to the unweighting needed in Monte-Carlo based approaches.

- Non-weighted ZW events with the discriminator solely used to reweight the latent loss during training. While we are not yet on the same level of precision for those events compared to the reweighted events, we are positive that we can achieve the same precision with some more tuning in the near future.

- Non-weighted Drell-Yan events with errors comparable to the statistical fluctuation of the training data in all phase space regions.

In the future we will try to employ our event generation methods from the ZW also again on the $t\tilde{t}$ and an additional $Z + Jets$ data set, to see how well our gained insights on event generation translate to more complicated data sets.

8 References

[1] *Lhc schedule*, https://home.cern/news/news/accelerators/new-schedule-lhc-and-its-successor.

[2] T. Sjöstrand, "The pythia event generator: past, present and future", Computer Physics Communications **246**, 106910 (2020).

[3] J. Alwall, R. Frederix, S. Frixione, V. Hirschi, F. Maltoni, O. Mattelaer, H.-S. Shao, T. Stelzer, P. Torrielli, and M. Zaro, "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**, 10.1007/jhep07(2014)079 (2014).

[4] M. Felcini, "Searches for dark matter particles at the lhc", (2018).

[5] A. Canepa, "Searches for supersymmetry at the large hadron collider", Reviews in Physics **4**, 100033 (2019).

[6] S. Chatrchyan, V. Khachatryan, A. Sirunyan, A. Tumasyan, W. Adam, E. Aguilo, T. Bergauer, M. Dragicevic, J. Erö, C. Fabjan, and et al., "Observation of a new boson at a mass of 125 gev with the cms experiment at the lhc", Physics Letters B **716**, 30–61 (2012).

[7] "Lhcb detector performance", International Journal of Modern Physics A **30**, 1530022 (2015).

[8] J. de Favereau, C. Delaere, P. Demin, A. Giammanco, V. Lemaître, A. Mertens, and M. Selvaggi, "Delphes 3: a modular framework for fast simulation of a generic collider experiment", Journal of High Energy Physics **2014**, 10.1007/jhep02(2014)057 (2014).

[9] M. Bellagente, A. Butter, G. Kasieczka, T. Plehn, and R. Winterhalder, "How to gan away detector effects", SciPost Phys. **8**, 070 (2020).

[10] N. Kidonakis, "Top quark production", (2013).

[11] S. D. Ellis and D. E. Soper, "Successive combination jet algorithm for hadron collisions", Physical Review D **48**, 3160–3166 (1993).

[12] Y. Dokshitzer, G. Leder, S. Moretti, and B. Webber, "Better jet clustering algorithms", Journal of High Energy Physics **1997**, 001–001 (1997).

[13] M. Cacciari, G. P. Salam, and G. Soyez, "The anti-ktjet clustering algorithm", Journal of High Energy Physics **2008**, 063–063 (2008).

[14] X. Ju and B. Nachman, "Supervised jet clustering with graph neural networks for lorentz boosted bosons", Physical Review D **102**, 10.1103/physrevd.102.075014 (2020).

[15] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators", Neural Networks **2**, 359–366 (1989).

[16] M. Skorski, A. Temperoni, and M. Theobald, "Revisiting initialization of neural networks", (2020).

[17] S. Ruder, "An overview of gradient descent optimization algorithms", (2017).

[18]N. Qian, "On the momentum term in gradient descent learning algorithms", Neural Networks **12**, 145–151 (1999).

[19]D. P. Kingma and J. Ba, "Adam: a method for stochastic optimization", (2017).

[20]L. N. Smith and N. Topin, "Super-convergence: very fast training of neural networks using large learning rates", (2018).

[21]L. Ardizzone, J. Kruse, S. Wirkert, D. Rahner, E. W. Pellegrini, R. S. Klessen, L. Maier-Hein, C. Rother, and U. Köthe, "Analyzing inverse problems with invertible neural networks", (2019).

[22]C. Durkan, A. Bekasov, I. Murray, and G. Papamakarios, "Cubic-spline flows", (2019).

[23]R. P. Winterhalder, "How to GAN : Novel simulation methods for the LHC", PhD thesis (U. Heidelberg (main), 2020).

[24]P. Baldi, L. Blecher, A. Butter, J. Collado, J. N. Howard, F. Keilbach, T. Plehn, G. Kasieczka, and D. Whiteson, "How to GAN Higher Jet Resolution", (2020).

[25]I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks", (2014).

[26]G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors", (2012).

[27]I. Loshchilov and F. Hutter, *Decoupled weight decay regularization*, 2019.

[28]T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida, "Spectral normalization for generative adversarial networks", (2018).

[29]A. Butter, T. Plehn, and R. Winterhalder, "How to GAN LHC Events", SciPost Phys. **7**, 075 (2019).

[30]M. Backes, A. Butter, T. Plehn, and R. Winterhalder, "How to GAN Event Unweighting", SciPost Phys. **10**, 089 (2021).

List of Figures

9 Acknowledgments

I would like to thank our whole group for the great atmosphere and nice lunch talks. Namely I would like to thank Tilman and Anja, who were great supervisors and helped us stay on track when we got lost in trying out way to much different stuff at the same time. Also to Theo and Armand who wrote the codebase and always had an open ear for our questions. Last but not least I would like to thank Ramon for listening to my "innovative" ideas which mostly turned out to be rubbish and to Sander, with whom working was always a delight, especially when discussing Gumbel related distributions.

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 8. August 2021,