University of Heidelberg

Department of Physics and Astronomy

How to unweight with GANs

Bachelor Thesis in Physics

submitted by

Mathias Backes

born in Neunkirchen (Germany)

2020

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

Abstract

Monte Carlo approaches for LHC event generation always pose the problem of weighted events, which have to be converted into unweighted events. Consequently, inefficient unweighting procedures cause a main bottleneck for this kind of event generation. In this thesis we show an approach to implement an unweighting procedure with the help of generative adversarial networks (GANs). To show that GANs are able to unweight data, we demonstrate the unweighting procedure with the well known Drell-Yan process.

Abstract (in deutscher Übersetzung)

Das Generieren von LHC Events mithilfe von Monte Carlo Methoden führt zu dem Problem, dass man gewichtete Events erhält, welche in ungewichtete Events konvertiert werden müssen. Infolgedessen verursachen ineffiziente Entwichtungen ein großes Problem für diese Art von Eventgeneration. In dieser Arbeit zeigen wir eine Möglichkeit, mittels Generative Adversarial Networks (GANs) zu entwichten. Um zu zeigen, dass GANs Daten entwichten können, demonstrieren wir das Entwichten anhand des allgemein bekannten Drell-Yan Prozesses.

Contents

1. Introduction **1**

2. Physical background **2**
2.1. Cross sections for two particle scattering 2
2.2. The parton model 3
2.3. Hadronic cross sections 5
2.4. Scattering kinematics with partons 5
2.5. Phase space integration and unweighting 7
2.6. Kinematic observables 8
2.7. Drell-Yan scattering 9

3. Neural networks **11**
3.1. Structure of an artificial neuron 11
3.2. Structure of a neural network 11
3.3. Activation functions 13
3.4. Training a neural network 15

4. Generative adversarial networks **19**
4.1. Structure of GANs 19
4.2. Usual application of GANs 21
4.3. Treating weighted data with GANs 23
4.4. Testing the unweighting procedure 23

5. Unweighting efficiency **26**
5.1. Implementing an efficiency 26
5.2. Testing the efficiency implementation 28

6. Cross check with VEGAS **30**
6.1. The VEGAS algorithm 30
6.2. Unweighting with VEGAS 31
6.3. Results for the camel function 32
6.4. Weight distribution after unweighting 33

7. Two dimensional toy model **34**
7.1. Generating toy data 34
7.2. Unweighting results 34
7.3. Efficiency calculation 35
7.4. VEGAS unweighting 36
7.5. Weight distributions 38

8. Drell-Yan process **39**

9. Summary and outlook **41**

1. Introduction

A main problem in LHC event generation are highly inefficient unweighting procedures, which are necessary to compare theoretical simulations with the experiments themselves. As an example we can consider a cross section of a simple partonic ($2 \to 2$) process [1],

$$\sigma_{\text{tot}} = \int \mathrm{d}\phi \int \mathrm{d}\cos\theta \int \mathrm{d}x_1 \int \mathrm{d}x_2 \, F_{\text{PS}} \, |\mathcal{M}|^2, \tag{1.1}$$

where $|\mathcal{M}|^2$ is the transition amplitude and F_{PS} is an appropriate function which will be specified later. To solve such an integral it is common to apply advanced Monte Carlo algorithms, which are based on importance sampling. These algorithms simulate events, but with an additional weight for every single event. These weights can be interpreted as a probability of the appearance of the event. Since in an experiment we can only distinguish between the appearance or absence of an event, we are not interested in weighted events. Therefore, we must find an algorithm to "unweight" this data, which means we want to gain a probability distribution depending only on our input configuration, not an additional weight. Until now, this problem causes a main bottleneck in LHC event generation, since most of the unweighting techniques are highly inefficient [1].

A really useful tool to handle LHC data are generative adversarial networks (GANs) [2]. It was already shown that GANs can be used to generate events [3], subtract event samples [4] or invert detector effects [5]. We want to build GANs that are able to unweight data. The difference to common unweighting methods is that we train on the weighted data and generate completely new, unweighted data. This new approach on how to unweight data might promise an improvement over other unweighting methods, as is shown in the thesis.

After recapitulating the basics of phase space integration (section 2) and neural networks (section 3), we introduce our theory on how to build GANs to unweight events and try it for different low dimensional toy models (sections 4, 7). The toy models are one- and two-dimensional and should be well reproduced to guarantee that we can expand in higher dimensions. We make sure that our unweighting is working well by defining an unweighting efficiency (section 5). To see the advantage of our unweighting method, we also compare it to the unweighting procedure of the well known VEGAS algorithm (section 6). In the last part of the thesis we show how our unweighting procedure deals with an actual physical application, in our case the Drell-Yan process (section 8).

2. Physical background

2.1. Cross sections for two particle scattering

We consider a scattering process with two input particles (p_A, p_B) and n output particles $(p_1, ..., p_n)$ with the four-momentum vectors $p = (E, \mathbf{p})$. Since we always have scattering processes with high relativistic energies, we neglect all the masses $(m_A = m_B = m_1 = ... = m_n = 0)$. For the total differential of the cross section we can calculate [6]

$$\mathrm{d}\sigma = \frac{1}{2E_A\, 2E_B\, |v_A - v_B|} |\mathcal{M}|^2 \mathrm{d}\Phi_{2\to n}. \tag{2.1}$$

The first factor, which is also called Moller-flux-Factor, can be expressed as

$$4E_A E_B |v_A - v_B| = 2s, \tag{2.2}$$

with s being the square of the center of mass energy. The second factor $|\mathcal{M}|^2$ is the matrix element of the process squared, which we can extract from perturbative Feynman diagrams. The last factor is the phase space element $\mathrm{d}\Phi_{2\to n}$, which in the relativistic case can be expressed as

$$\mathrm{d}\Phi_{2\to n} = (2\pi)^4 \delta^{(4)}(p_A + p_B - p_1 - ... - p_n) \prod_{j=1}^{n} \frac{\mathrm{d}^3\mathbf{p}_j}{(2\pi)^3} \frac{1}{2E_j}\bigg|_{E_j = \sqrt{\mathbf{p}_j^2}}. \tag{2.3}$$

Two particle final states

Now we will calculate this cross section explicitly. As a restriction we choose only two final state particles, following [6]. For this special case we can simplify the phase space factor by evaluating the integrals in the center of mass frame of the initial particles. The condition $\mathbf{p}_A + \mathbf{p}_B = 0$ for the input particles sets $\mathbf{p}_1 = -\mathbf{p}_2$ since

$$(2\pi)^4 \delta^{(4)}(p_A + p_B - p_1 - p_2) = (2\pi)^4 \delta(E_A + E_B - E_1 - E_2)\delta^3(\mathbf{p}_1 + \mathbf{p}_2). \tag{2.4}$$

We evaluate the $\mathbf{p}_2$ integral and get

$$\mathrm{d}\Phi_{2\to 2} = (2\pi)\delta(E_A + E_B - E_1 - E_2) \frac{\mathrm{d}^3\mathbf{p}_1}{(2\pi)^3} \frac{1}{2E_1 2E_2}\bigg|_{E_{1,2} = \sqrt{\mathbf{p}_1^2}}. \tag{2.5}$$

Now we can change the coordinate system to spherical coordinates

$$\begin{aligned} \mathrm{d}\Phi_{2\to 2} &= \frac{1}{(4\pi)^2} \frac{\delta\left(E_A + E_B - 2\sqrt{|\mathbf{p}_1|^2}\right)}{|\mathbf{p}_1|^2} |\mathbf{p}_1|^2 \mathrm{d}|\mathbf{p}_1| \mathrm{d}\Omega \\ &= \frac{1}{(4\pi)^2} \delta\left(E_A + E_B - 2\sqrt{|\mathbf{p}_1|^2}\right) \mathrm{d}|\mathbf{p}_1| \mathrm{d}\Omega, \end{aligned} \tag{2.6}$$

with $\mathrm{d}\Omega = \mathrm{d}\cos\theta \mathrm{d}\phi$. After executing the integration in $|\mathbf{p}_1|$-direction, we obtain

$$\mathrm{d}\Phi_{2\to 2} = \frac{1}{(4\pi)^2} \frac{1}{2} \mathrm{d}\Omega, \tag{2.7}$$

and therefore for the differential cross section

$$\frac{\mathrm{d}\sigma}{\mathrm{d}\Omega} = \frac{1}{64\pi^2}\frac{|\mathcal{M}|^2}{s}. \tag{2.8}$$

2.2. The parton model

The parton model was Richard Feynman's attempt to explain the Bjorken scaling at the SLAC experiment in the 1960s [7]. In this experiment electrons and nucleons collide with a beam energy of $\sqrt{s} \approx 20$ GeV. The cross section of this deep inelastic scattering (DIS) process can be expressed in terms of the structure functions $W_i(Q^2, \nu)$ [8]

$$\sigma_{\mathrm{DIS}} \sim \sigma_0 \Big(W_2 + 2W_1 \tan^2(\theta/2) \Big), \tag{2.9}$$

with the Mott cross section σ_0 of the scattering of a lepton on a point-like charged particle. Q^2 represents the negative of the squared four-momentum-transfer vector q of the exchanged virtual photon. $\nu = E - E'$ is the energy loss of the scattered electron, θ is the scattering angle and M is the invariant mass. For this deep inelastic scattering the structure functions $W_i(Q^2, \nu)$ exhibit a scaling in the asymptotic limit of Q^2 and ν, called Bjorken scaling

$$\begin{aligned} \lim_{Q^2,\nu\to\infty} W_1(Q^2,\nu) &= F_1(x), \\ \lim_{Q^2,\nu\to\infty} \nu W_2(Q^2,\nu) &= M F_2(x), \end{aligned} \tag{2.10}$$

with ν/Q^2 fixed. The form factors $F_i(x)$ only depend on the Bjorken scaling variable

$$x = \frac{Q^2}{2M\nu}. \tag{2.11}$$

Previous experiences contradicted these results since the nucleons have a finite size that corresponds to the idea of rapidly falling form factors $F_i(Q^2)$ in case of increasing Q^2. The fact that the Bjorken scaling was experimentally proven led to the hypothesis of a nucleon as a collection of point-like constituents for deep inelastic scattering.
Feynman named these point-like constituents partons and proposed the parton model. In this model he considers the partons to be in an infinite momentum frame (masses can be neglected), where the whole parton energy comes from the momentum in beam direction. By assumption these partons are incapable of exchanging large momenta among themselves through any kind of interaction. Every parton k carries a fraction x_k of the nucleons momentum p, i.e.

$$p_k = x_k p, \qquad \sum_k x_k = 1. \tag{2.12}$$

For each species of parton there is a function $f_i(x)$ which quantifies the probability that the nucleon contains a parton of this species. These functions are called parton

distribution functions (PDF) and can be connected to the Bjorken scaling with

$$F_2(x) = \sum_k q_k^2 f_k(x). \tag{2.13}$$

One way of estimating PDFs is by starting from a parametrisation of non-pertubative PDFs at a low scale and fitting them to experimental data, using the DGLAP evolution scheme [9]. The resulting PDFs depend on multiple parameters, for example the chosen order of perturbation, the choice of the input data, the treatment of heavy quarks or the correlation between α_s and the PDFs [10].
As an example we take a look at the parton distribution functions of the proton, calculated with next-to-leading order perturbative expansion. It is not surprising that we find u and d quarks to carry the majority of the proton's momentum, while the other quarks tend to have small longitudinal fractions.

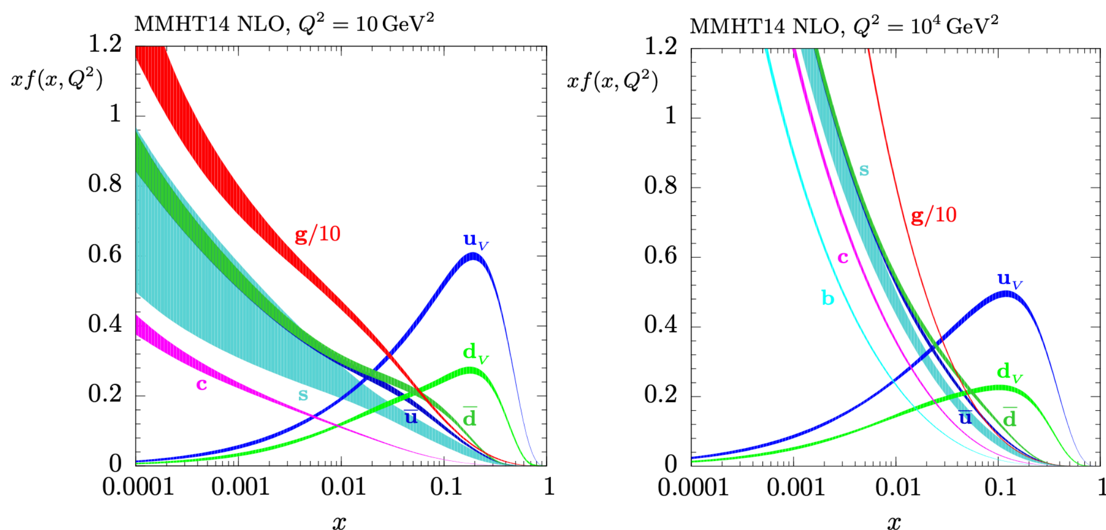

Figure 2.1: NLO PDFs at $Q^2 = 10[GeV]^2$ and $Q^2 = 10^4[GeV]^2$. Associated to the PDFs are 68% uncertainty bands. [source: [11]]

As one can see in figure 2.1, the desired independence of the PDFs concerning Q^2 is not given everywhere. The parton model cannot give us an explanation for these effects. Today the theory of Quantum chromodynamics (QCD) is the best explanation for nucleon configurations. The main difference between the two approaches is that in QCD the quarks are allowed to exchange energy with gluons. The violation of the Bjorken scaling can also be explained with higher order corrections in QCD [6].

2.3. Hadronic cross sections

Now we want to calculate the hadronic cross section $\sigma_{k,l\to C}$ of two partons k, l scattering to a final state C. The partonic cross section of this process,

$$\hat{\sigma}_{k,l\to C}(\hat{s}), \tag{2.14}$$

depends on the squared partonic center of mass energy $\hat{s}$, which we calculate according to equation 2.12

$$\hat{s} = 2p_k p_l = x_k x_l s. \tag{2.15}$$

To calculate the hadronic cross section we simply weight the partonic cross sections with the partonic distribution functions and sum over all momentum fractions $x_{k,l}$, i.e. since these are continuous in $[0, 1]$ we integrate them [6] and have

$$\sigma_{k,l\to C} = \int_0^1 \mathrm{d}x_k \int_0^1 \mathrm{d}x_l f_k(x_k) f_l(x_l) \hat{\sigma}_{k,l\to C}(x_k x_l s). \tag{2.16}$$

If we want to consider the scattering of two hadrons which are constructed out of quarks, we will need to sum over all possible scattering quark combinations

$$\sigma_{k,l\to C} = \sum_{k,l} \int_0^1 \mathrm{d}x_k \int_0^1 \mathrm{d}x_l f_k(x_k) f_l(x_l) \hat{\sigma}_{k,l\to C}(x_k x_l s). \tag{2.17}$$

2.4. Scattering kinematics with partons

One consequence of the PDFs is the difference between the partonic and the hadronic center of mass system (CMS). If we want to transform between these systems, we need to transform one system into the other with a Lorentz boost into the direction of the beam axis.
For example we can take a look at two protons in the hadronic CMS with the momenta

$$p_A = \frac{\sqrt{s}}{2} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \qquad p_B = \frac{\sqrt{s}}{2} \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix}. \tag{2.18}$$

If we consider the partons of these protons in the hadronic CMS, we can describe them with the four-momentum vector

$$p_a = p_a^0 \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = x_1 p_A^0 \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \qquad p_b = p_b^0 \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix} = x_2 p_B^0 \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix}. \tag{2.19}$$

$x_{1,2}$ are momentum fractions. If we apply a Lorentz transformation

$$\Lambda^\mu_\nu = \begin{pmatrix} \gamma & 0 & 0 & \beta\gamma \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \beta\gamma & 0 & 0 & \gamma \end{pmatrix}, \tag{2.20}$$

we get the four-momentum vector in the partonical CMS

$$\hat{p}_a = x_1 p_A^0 \begin{pmatrix} \gamma + \beta\gamma \\ 0 \\ 0 \\ \gamma + \beta\gamma \end{pmatrix}, \qquad \hat{p}_b = x_2 p_B^0 \begin{pmatrix} \gamma - \beta\gamma \\ 0 \\ 0 \\ -\gamma + \beta\gamma \end{pmatrix}. \tag{2.21}$$

In the partonic CMS we have $\hat{p}_a^0 = \hat{p}_b^0$, so with $p_A^0 = p_B^0$ we can derive

$$\beta = \frac{p_b^0 - p_a^0}{p_b^0 + p_a^0} = \frac{x_2 - x_1}{x_2 + x_1}. \tag{2.22}$$

Scattering calculations

We now want to calculate the momenta of two incoming partons through the momenta of the outgoing particles of a $2 \to 2$ scattering. For the outgoing particles we have the four-momentum vectors

$$p_1 = \begin{pmatrix} E_1 \\ 0 \\ 0 \\ p_1^3 \end{pmatrix}, \qquad p_2 = \begin{pmatrix} E_2 \\ 0 \\ 0 \\ p_2^3 \end{pmatrix}. \tag{2.23}$$

We can also calculate the center of mass energy

$$E_{\mathrm{CMS}} = \sqrt{(E_1 + E_2)^2 - (p_1^3 + p_2^3)^2}. \tag{2.24}$$

We do not know the momentum fractions $x_{1,2}$, but we can make an ansatz in analogy to equation 2.19 and since we now know that in the partonic system equation 2.15 applies, we specify

$$p_a = \frac{x_1}{2} E_{\mathrm{CMS}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \qquad p_b = \frac{x_2}{2} E_{\mathrm{CMS}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix}. \tag{2.25}$$

Since we have momentum and energy conservation, we also have

$$\begin{aligned} E_1 + E_2 &= \frac{x_1}{2} E_{\mathrm{CMS}} + \frac{x_2}{2} E_{\mathrm{CMS}}, \\ p_1^3 + p_2^3 &= \frac{x_1}{2} E_{\mathrm{CMS}} - \frac{x_2}{2} E_{\mathrm{CMS}}, \end{aligned} \tag{2.26}$$

which can be rearranged to

$$x_1 = \frac{(E_1 + E_2) + (p_1^3 + p_2^3)}{E_{\mathrm{CMS}}}, \qquad x_2 = \frac{(E_1 + E_2) - (p_1^3 + p_2^3)}{E_{\mathrm{CMS}}}, \tag{2.27}$$

and leads to

$$p_a = \frac{(E_1 + E_2) + (p_1^3 + p_2^3)}{2} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \qquad p_b = \frac{(E_1 + E_2) - (p_1^3 + p_2^3)}{2} \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix}. \tag{2.28}$$

Finally, we are able to calculate the momenta of the incoming partons.

2.5. Phase space integration and unweighting

Since we are interested in the cross section of a physical process, we now want to take a look at the numerical phase space integration. If we consider scattering cross sections with multiple particles involved, we will get high dimensional phase space integrals. This multidimensionality makes it necessary to adopt advanced Monte Carlo integration methods. In these methods we sample a chain of random points $\mathbf{Y}_j$ with length N_Y that can be organized in any number of dimensions. To sample this chain we can define a smartly chosen, normalised function $\rho(\mathbf{y})$ which gives us the probability of finding $\mathbf{Y}_j \in [\mathbf{y}, \mathbf{y} + d\mathbf{y}]$. Since this probability function is normalised, we can calculate a d dimensional integral over the hypercube as

$$\int_0^1 \mathrm{d}^d y f(y) \approx \frac{1}{N_Y} \sum_j \frac{f(\mathbf{Y}_j)}{\rho(\mathbf{Y}_j)}. \tag{2.29}$$

Compared to other numerical approaches, which rely on binning of d dimensions, the Monte Carlo approach replaces these d dimensional arrays with one large chain, which is more efficient. The accuracy of this approach depends on the way we choose the function $\rho(\mathbf{y})$. The more this function resembles the integrand, the better our estimate for the integral will be. This task is called importance sampling. One way of solving this task is with the VEGAS algorithm, which will be discussed in section 6.
Now we go back to the $(2 \to 2)$ scattering cross section. First, we have to remap the cross section integration to the unit hypercube since it simplifies the integration and most of the Monte Carlo integrators are built for such integrals [1]

$$\begin{aligned} \sigma_{\text{tot}} &= \int \mathrm{d}\Omega \int \mathrm{d}x_1 \int \mathrm{d}x_2\, f_1(x_1) f_2(x_2)\, \frac{1}{64\pi^2} \frac{|\mathcal{M}|^2}{s} \\ &= \int_0^1 \mathrm{d}y_1 ... \mathrm{d}y_4\, J_{\text{PS}}(\mathbf{y})\, f_1(x_1(\mathbf{y})) f_2(x_2(\mathbf{y}))\, \frac{1}{64\pi^2} \frac{|\mathcal{M}(\mathbf{y})|^2}{s}. \end{aligned} \tag{2.30}$$

We need a tool which translates the integration variables $\mathbf{y} = (y_1, y_2, y_3, y_4)$ into external momenta. This tool is called a phase space generator. Since our phase space is not defined in terms of these $\mathbf{y}$, the phase space generator introduces the Jacobian J_{PS}. We also choose this transformation to be part of the Monte Carlo sampling process, so we implicitly encode the density function ρ in the Jacobian J_{PS}. According to the Monte Carlo methods, we can now calculate this integral as

$$\frac{1}{N_Y} \sum_j J_{\text{PS}}(\mathbf{Y_j})\, f_1(x_1(\mathbf{Y_j})) f_2(x_2(\mathbf{Y_j}))\, \frac{1}{64\pi^2} \frac{|\mathcal{M}(\mathbf{Y_j})|^2}{s}. \tag{2.31}$$

This procedure has an interesting interpretation. Once we compute the unique phase space configuration $\mathbf{x}_j = (p_A, p_B, p_1, p_2)_j$ corresponding to the mapped vector $\mathbf{y}_j$, the combined weight,

$$W_j = \mathrm{d}\sigma = \mathcal{N} J_{\text{PS}}\, f_1(x_1(\mathbf{y}_j)) f_2(x_2(\mathbf{y}_j))\, |\mathcal{M}(\mathbf{y}_j)|^2\, \mathrm{d}^d \mathbf{y}_j, \tag{2.32}$$

is the probability of the event $\mathbf{x}_j$, with the normalisation factor $\mathcal{N}$. This means what we do is not just a simple numerical integration; we in fact simulate events, more precisely weighted events [1].
As already mentioned in the introduction, these weighted events are not what experimentalists want since they want a probability distribution depending only on external momenta and not an additional weight. This means we have to find a way to translate this weighted event distribution into a distribution without or with a constant weight, i.e. we need to unweight the events. There are multiple ways of doing this:

Distributive method

This method uses the minimal weight W_{min} to express all other weights relative to W_{min}. Every event with weight W_j/W_{min} is now replaced by W_j/W_{min} events with unitary weights. In consequence, there are now a number of events, all without a weight. This method's problem is the binned phase space. There is an additional number of events for every bin, but no rule how to distribute the events in and around the bin. Therefore, this unweighting method is not very precise [1].

Hit-or-miss method

The idea of this method is to translate all the weights into a probability to keep the event or drop it. We divide every weight with the maximal weight $W_{\text{j,rel}} = W_j/W_{\text{max}}$, generate a random number $R \in [0, 1]$ and keep the event only if it satisfies

$$W_{\text{j,rel}} > R. \tag{2.33}$$

This is the most common method [1]. The problem with this unweighting method is that we lose a lot of events. Nevertheless, we will come back to it later since we will use elements of the hit-or-miss method to calculate an unweighting efficiency.

2.6. Kinematic observables

We want to introduce some kinematic observables for the $(2 \to 2)$ scattering which are invariant or transform easily under a Lorentz transformation in beam direction since the partonic and hadronic CMS are connected this way. There are some useful kinematic observables:

- Transversal momentum
$$p_T = \sqrt{(p^1)^2 + (p^2)^2}. \tag{2.34}$$

- Invariant mass of the final particles
$$M_{3,4} = \sqrt{(p_3 + p_4)^2}. \tag{2.35}$$

- Azimuthal angle ϕ of a final particle

$$\phi = \arctan\left(\frac{p^2}{p^1}\right). \tag{2.36}$$

- Rapidity

$$y = \frac{1}{2}\ln\left(\frac{p^0 + p_L}{p^0 - p_L}\right), \tag{2.37}$$

with $p_L = p^3$ being the momentum in beam direction.

- Pseudorapidity

$$\eta = \frac{1}{2}\ln\left(\frac{|\mathbf{p}| + p_L}{|\mathbf{p}| - p_L}\right), \tag{2.38}$$

which is in our case equal to the rapidity y since all masses are neglected.

Except the rapidities, all these values are Lorentz invariant. This is why we consider them to be good observables. The rapidity y transforms under Lorentz transformation as

$$\begin{aligned} y_L &= \frac{1}{2}\ln\left(\frac{\gamma p^0 + \gamma\beta p_L}{\gamma p^0 - \gamma\beta p_L}\right) \\ &= \frac{1}{2}\ln\left(\frac{p^0 + p_L}{p^0 - p_L}\right) + \frac{1}{2}\ln\left(\frac{1+\beta}{1-\beta}\right) \\ &= \frac{1}{2}\ln\left(\frac{p^0 + p_L}{p^0 - p_L}\right) + \frac{1}{2}\ln\left(\frac{x_2}{x_1}\right), \end{aligned} \tag{2.39}$$

with β following equation 2.22. We can also see that the rapidity difference between the outgoing particles is a Lorentz invariant observable

$$y^* = \frac{|y_3 - y_4|}{2}. \tag{2.40}$$

The pseudorapidity η is very similar to the rapidity, but easier to measure. With a few geometrical transformations one can derive the pseudorapidity as

$$\eta = -\ln\tan\frac{\theta}{2}, \tag{2.41}$$

where θ is the angle between the particle trajectory and the beam direction [12].

2.7. Drell-Yan scattering

Having figured out the essential theoretical basics, we now take a look at an explicit scattering process, the Drell-Yan process [13],

$$\mathrm{H}_a + \mathrm{H}_b \to l^- + l^+. \tag{2.42}$$

In general, two hadrons ($H_{a,b}$) collide to form a lepton-antilepton pair ($l^- + l^+$). It is one of the best explored processes at hadron-hadron colliders [14] since one can make precise theoretic predictions and the experimental signal consists of only two leptons. The Drell-Yan process was also used to design experiments through which the W and Z bosons [15] [16] were discovered, and was also important in the discovery of the top quark at Fermilab [17]. Today, the Drell-Yan process is still an important tool to observe the partonic stucture of hadrons since with the LHC beam energy of 14 TeV contributions for heavier quark flavours can be resolved.

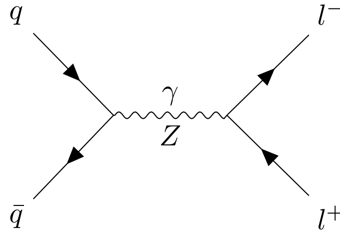

Figure 2.2: Feynman diagramm of a Drell-Yan process at leading order. The incoming particles are quarks; the outgoing particles are leptons.

We can draw the Feynman diagram in leading order (figure 2.2). The exchange particle can be a photon γ or a Z boson. For low energies, the only important term is the pure photon transition amplitude, but since we want to simulate a LHC process at high energies, we also have to resolve the contribution of the Z boson. We need to consider this when calculating the total transition matrix

$$\mathcal{M} = \mathcal{M}_\gamma + \mathcal{M}_Z. \tag{2.43}$$

The transition amplitude can be written as

$$|\mathcal{M}|^2 = |\mathcal{M}_\gamma|^2 + |\mathcal{M}_Z|^2 + 2\mathrm{Re}(\mathcal{M}_\gamma \mathcal{M}_Z^*). \tag{2.44}$$

Besides the pure photon or Z contribution we also got an interference term, which does not have to be positive. Therefore, we cannot be sure that every weight we will simulate is positive. We have to make sure that the neural network we build is able to deal with negative event weights.
The data we will use later is the result of a proton scattering

$$p + p \to \mu^- + \mu^+. \tag{2.45}$$

3. Neural networks

Artificial neural networks are an approach of deep learning inspired by the human brain, which is build of neurons. The biological neuron is constructed to release a chemical transmitter once a certain action potential in form of an electrical impulse is reached [18]. This release leads to another electrical impulse which is passed to other neurons and so on. This idea of interacting neurons is the basis on which we construct an artificial neural network.

3.1. Structure of an artificial neuron

The simplest possible neural network consists of one single artificial neuron which gets a number of inputs x_i (in figure 3.1 three of them) and produces one output y. The way this output is produced is shown diagrammatically in figure 3.1 [18].

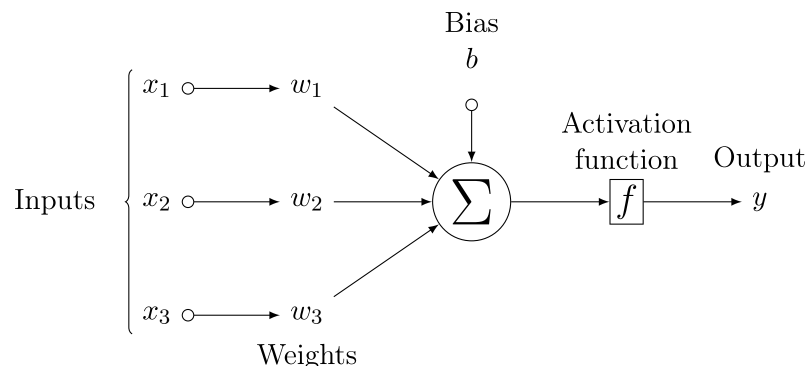

Figure 3.1: Structure of a neuron. The input is multiplied with weights w_i, summed up with an additional bias and a non-linear activation function is applied.

Mathematically, this can be expressed as

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right). \tag{3.1}$$

An artificial neuron can therefore be described by its weights, bias and the non linear activation function. The output of a single neuron is always one dimensional. To get a more dimensional output we have to arrange multiple neurons next to each other. This structure is called a layer (even if it contains only a single neuron).

3.2. Structure of a neural network

A neural network consists of several neurons, which are typically organised in layers. We differentiate between three kinds of layers: the input layer at the beginning, the output layer at the end and the hidden layers in between [19] (figure 3.2).

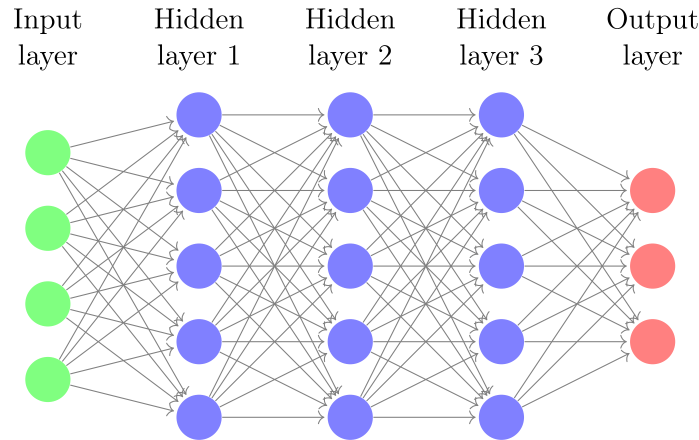

Figure 3.2: Exemplary structure of a generic neural network. Each circle illustrates one neuron. The green layer on the left side is called the input layer, the blue layers in the center are called the hidden layers and the red layer on the right is called the output layer.

The input layer just takes the input information and sends it to the first hidden layer. For more advanced neural networks, the input data might already be preprocessed to improve the neural network or to fit the neural network specifications.
At this point we need to specify the idea of the weights w_i. One weight is not assigned to a single neuron, but it expresses the "importance" of one connection between two neurons of consecutive layers (in figure 3.2 shown as an arrow) [19]. Every single neuron of one layer now takes this input and produces an output according to equation 3.1. This procedure is repeated for every hidden layer. The hidden layers usually have more neurons per layer than the input or output layer to guarantee that no input information needs to be compressed in the neural network. Later we will refer to the number of neurons in one hidden layer as units. The output layer needs to have the exact number of neurons to fit the requested dimensionality of the output since every neuron itself has one dimensional output.
We can compare the whole neural network to a Taylor series of an irrational function. The idea of a Taylor series is to distribute an irrational function into a sum of rational functions, for example for the sinus function we get

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} \mp \dots \,. \tag{3.2}$$

In figure 3.3 we show how the Taylor series approximates the sinus function better with every additional term. In analogy to this idea, we want all (or at least most of the) neurons to get a small piece of information which, combined, should express the complicated distribution we want to reproduce.

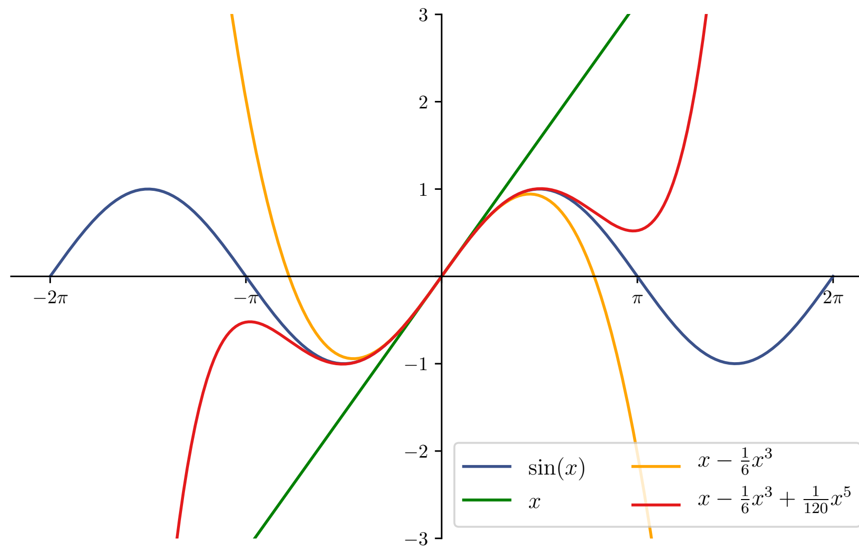

Figure 3.3: Taylor series expansion at zero of the sinus function up to the third order. The series expansion approximates the irrational function better with every order.

At this point, we also understand why we have to apply a non-linear activation function. If we had no activation function at all or if we chose a linear activation function, equation 3.1 would become linear as it is a composition of linear functions. Following this thought, if every neuron could be represented by a linear function, we would be able to replace all the neurons with one single neuron because consecutive linear functions can be reduced to one linear function. Since we do not want this to be the case, we need to have a non-linear activation function [18].

3.3. Activation functions

There are several types of activation functions, an overview can be found in [20]. In the following, we only describe the activation functions which are used in our network and plot them (figure 3.4).

The ReLU activation function (**Re**ctified **L**inear **U**nit) has been the most used activation function since its proposal in 2010 [20]. It is defined by

$$\mathrm{ReLU}(x) = \max\{0, x\} = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}. \tag{3.3}$$

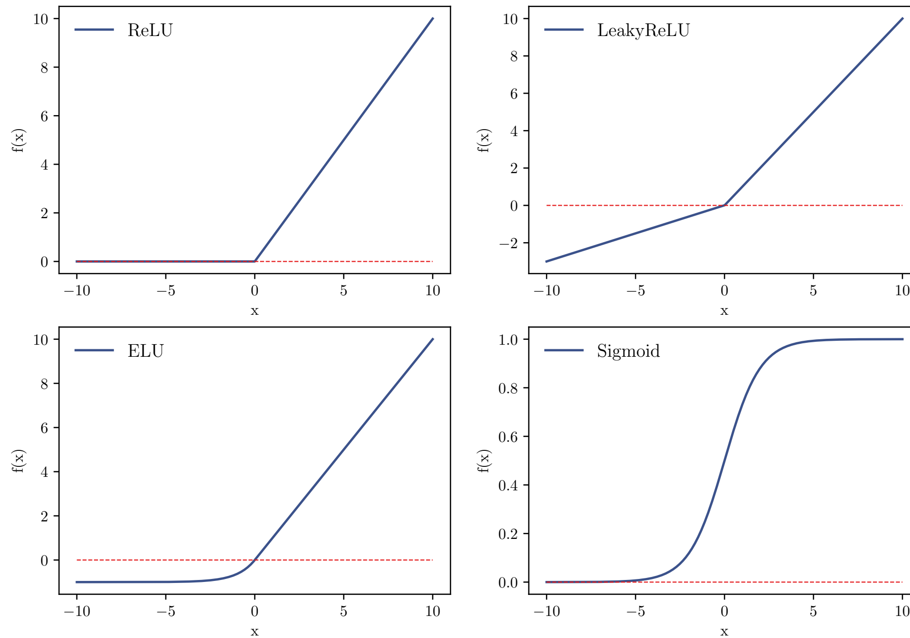

Figure 3.4: Plots of activation functions. On the upper left the ReLU activation function, on the upper right the LeakyReLU activation function with $\alpha = 0.3$, on the lower left the ELU activation function with $\alpha = 1$, and on the lower right the sigmoid activation function.

This activation function, which is nearly linear, has the advantage that there is no need to calculate exponentials or divisions for example. Therefore, the computation is much faster compared to other activation functions. Problems with this activation function can occur with "dying" neurons, which means that the output of a neuron is constantly set to zero and therefore is basically ignored. To solve this problem one can use the LeakyReLU function.

The LeakyReLU activation function has a similar shape as the ReLU activation function, it is just not constantly zero for negative inputs

$$\text{LeakyReLU}(x) = \begin{cases} x & x > 0 \\ \alpha x & x \leq 0 \end{cases}. \tag{3.4}$$

The LeakyReLU hyperparameter α is in our case set to 0.3.

The ELU activation function (**E**xponential **L**inear **U**nit) is also very similar to the ReLU activation function, but for negative inputs there is an exponential shape

$$\mathrm{ELU}(x) = \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0 \end{cases}. \tag{3.5}$$

As we will see later, the exponential shape of the ELU activation function will be useful if we want to generate exponentially shaped data. The ELU hyperparameter α is usually set to one [20].

The sigmoid activation function is another important activation function

$$\mathrm{Sigmoid}(x) = \frac{1}{1 + e^{-x}}. \tag{3.6}$$

The non-zero centred output can cause the gradient updates to propagate in different directions [20]. Nevertheless, we will be using the sigmoid activation in our neural network to map values strictly to the interval $[0, 1]$.

3.4. Training a neural network

If we now want to train the neural network, we need to adjust all the layer-connecting weights and the biases to reproduce the data we trained on. To reach this goal, we first have to discuss the loss function.
A loss function is designed to increase if the output predicted by the neural network (y) and the actual output value ($\hat{y}$) become different, and to decrease if they get very similar. One way of obtaining such a function is the mean squared error (here for an n dimensional output) [18]

$$L_{\mathrm{MSE}} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2. \tag{3.7}$$

The predicted output of the neural network only depends on the parameters w of the neural network (with the biases implied) and the input. The input is fixed, so we can interpret the loss as a function of the neural net parameters $L = L(w)$. The best case would be if the network performed perfectly, i.e. if the loss function would be as low as possible, which means that the predicted and actual output are very similar. If the parameters are all in perfect configuration w_∞, the loss function will arrive at its global minimum. Therefore, we want to find the parameters which minimize the loss function [18]

$$w_\infty = \arg\min_w L(w). \tag{3.8}$$

As long as we neglect boundary effects, a necessary condition at the global minimum of the loss function is

$$\nabla_w L(w) = 0. \tag{3.9}$$

To achieve this numerically we use gradient descent. An exception from this idea of finding the global minimum can occur if we regularise the weights in a bad way. If the hyperspace of the weights is restricted to a certain area with the global minimum outside, there is the possibility that the weight configuration which minimizes the loss lies at the edges of the restricted hyperspace. In this case we cannot find the global minimum at all, so we need to make sure to choose eventual weight restrictions wisely.

Stochastic gradient descent is a useful tool to obtain this configuration. The neural network is first initialised with random parameters. For the first prediction we will therefore have a completely arbitrary result. After every prediction, the parameters are redefined [18]

$$w_{t+1} = w_t - \alpha \nabla_{w_t} L, \tag{3.10}$$

with w_t being the previous weight. The gradients of the loss function are calculated with backpropagation; more information about this algorithm can be found in [21]. This redefinition of the parameters after each iteration will change the weights towards the minimizing weights. The factor to quantify the size of the step, α, is called the learning rate. It is important to find a good value for α, because if it is to low, the neural network might get stuck in a local minimum; if it is to high, the network will never converge to the global minimum (figure 3.5) [18].

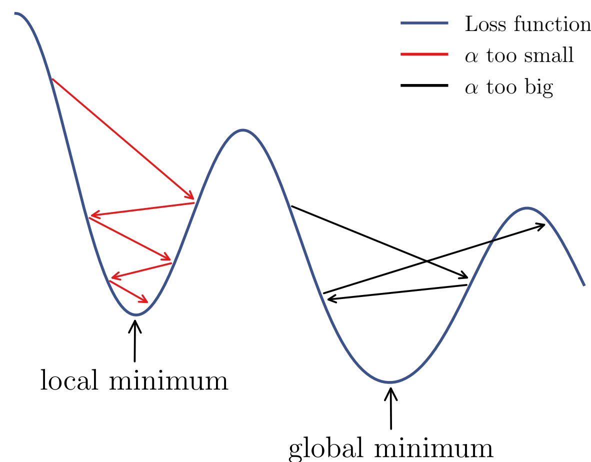

Figure 3.5: Learning process of a neural network with different learning rates. If the learning rate is too high, the network will never arrive in any minimum; if it is too low, the network might get stuck in a local minimum.

As one can imagine, the gradient descent method is useful, but not the best solution. Finding the global minimum of the loss function requires a tool to prevent overstepping it, i.e. an optimizer for the size of the steps [22].
Since the loss function can have local minima, we need our algorithm to make sure we

find the global minimum. This feature is called exploration. After finding this minimum, the neural network needs to approach it in well chosen steps. This feature is called exploitation. There are a lot of optimizers to reach this goal [22].

ADAM is the optimizer of our choice. The first idea implemented in ADAM is a momentum m_t which is added to the gradient term of the stochastic gradient descent method [23]

$$\begin{aligned} m_{t+1} &= \rho m_t + \alpha \nabla_{w_t} L, \\ w_{t+1} &= w_t - m_{t+1}. \end{aligned} \tag{3.11}$$

ρ is a factor in $[0, 1]$ to add a fraction of the previous update to the next update. This additional term smooths out the updates, helps to prevent getting stuck in a local minimum and is useful in areas where we have big differences between the gradients in several directions. In such a case, neural networks without momentum tend to take a lot of sidesteps instead of going straight to the minimum. This problem is solved if we include the calculated momentum. A disadvantage is that it is easier to overshoot the minimum. ADAM uses two different kinds of momenta and there are soma additional other features; more informations about the optimizer can be found in [24].

The decay of the learning rate is another way to improve the learning process. If we are on a good way to converge to a global minimum of the loss function after several training epochs (with a number of iterations), we want to have a smaller learning rate to obtain smaller steps and therefore more precision. There are several ways to implement a decay. In this work we reduce the learning rate at the beginning (α_0) after N epochs to

$$\alpha(N) = \frac{\alpha_0}{1 + N \cdot \gamma}, \tag{3.12}$$

with γ being the decay parameter.

Overfitting is another problem that can occur. Since our training data is limited, we will not be able to avoid several training iterations with the same data. This risks overspecialisation or overfitting [25], meaning that the neural network learned details about the training data which are specific to the training data (see figure 3.6). Therefore the network is able to distinguish between points of the training data and other points, without depending on whether they fit the underlying structure. The bigger the network, the easier it can overfit. So for complex shaped data which requires a certain size of the network one will need to reduce the possibility of overfitting without reducing the size of the network.
One possibility to prevent overfitting is a dropout of neurons [26]. The idea is to omit each hidden neuron in each presentation of each training case with a dropout probability, for example 0.5. If we speak of a neuron dropping out, we mean that the activation or other features of the neuron are set to zero. This procedure can be seen as "a very efficient way of performing model averaging with neural networks" [26] since we have a

different combination of active neurons in every training iteration but all these "subnetworks" share the same weights. The neurons cannot rely on other hidden units, so we take away the ability of the network to focus on random properties of the training data. The dropout probability must not be chosen too low. Otherwise, we would not gain the desired effect. At the same time, it must not be too high because it might reduce the ongoing training.
Another possibility to prevent overfitting is an early stopping of the learning process [27]. If there is a possibility of overfitting, the desired minimum of the loss function is no longer the global minimum since then the network will have learned special properties of the training data. The desired local minimum in which the network has learned the process might be reached before the training is finished. At this point, we need the network to stop its training and stay in the local minimum, i.e. perform an early stop. To check if this minimum is reached, we can simply evaluate the loss of additional external testing data. We quit once the loss of this testing data starts to rise as the network is overfitting.

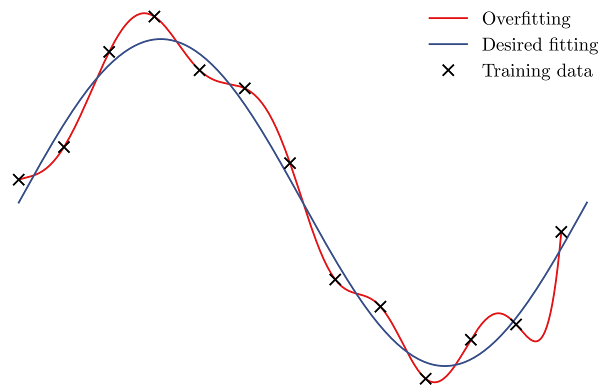

Figure 3.6: Overfitting of a neural network shown in a graphic example. The underlying structure is a sinus function

4. Generative adversarial networks

4.1. Structure of GANs

A GAN [2] is built with two different neural networks which compete with each other. On the one hand there is the generator G, which tries to mimic the input data, on the other hand there is the discriminator network D, which tries to distinguish between the real and the generated data [3]. The generator G is a multilayer neural network whose output has the same dimensionality as the training data; the discriminator D outputs a single scalar. These networks play against each other to improve the generator by adjusting its parameters in regions, where the discriminator can distinguish easily. The structure of the network is shown in figure 4.1.

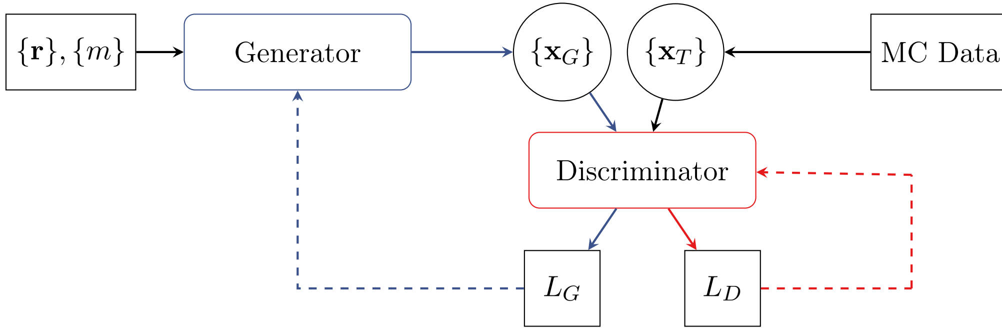

Figure 4.1: Structure of generative adversarial networks. The generator tries to mimic the data and the discriminator tries to distinguish between true and generated data. During the training procedure the loss functions should be minimized.

The input for our neural network consists of random numbers $\{\mathbf{r}\}$, the "true" data $\{\mathbf{x}_T\}$ we want to imitate and later the masses $\{m\}$ of the final particles of a physical example. The generator, whose weights are initialized with random numbers, maps the input $\{\mathbf{r}\}$ to a distribution $P_G(\mathbf{x})$ that has the same dimensionality as the true data. In this first step, the distribution $P_G(\mathbf{x})$ is like the input $\{\mathbf{r}\}$ completely arbitrary. Now the discriminator can take samples $\{\mathbf{x}_T\}$ and $\{\mathbf{x}_G\}$ from the real distribution $P_T(\mathbf{x})$ and the predicted distribution $P_G(\mathbf{x})$. These sets are provided in batches $\{\mathbf{x}_{T,G}\}$ and are now getting labelled by the discriminator. Therefore, we define the discriminator output $D(\mathbf{x}) \in (0,1)$. The perfect discriminator is trained to label true data with $D = 1$ and generated data with $D = 0$. To make sure that the discriminator only produces values in this interval, we use the sigmoid as activation function in the output layer. We now want to find a loss function which has a global minimum if every point of the distributions is labelled correctly. To get a better sensitivity for $D \to 0$ we evaluate the logarithm of the discriminator output and find the loss function [3]

$$L_D = \langle -\log D(\mathbf{x}) \rangle_{\mathbf{x}\sim P_T} + \langle -\log(1 - D(\mathbf{x})) \rangle_{\mathbf{x}\sim P_G}. \tag{4.1}$$

In case of a perfect generator, the discriminator output will constantly be 0.5, which means the loss function will become $L_D = -2\log(0.5) \approx 1.4$.
The generator also relies on the function $D(\mathbf{x})$ during his training. In opposition to the discriminator it tries to maximise L_D by increasing its second term. Instead of increasing the second term we can also minimize the generator loss

$$L_G = \langle -\log D(\mathbf{x}) \rangle_{\mathbf{x}\sim P_G}. \tag{4.2}$$

To train the network we need to alternate between updates of the discriminator and the generator parameters. This training procedure is presented in Algorithm 1.

Algorithm 1 Training of the generator and discriminator

for number of training iterations **do**
  for number of discriminator updates relative to generator updates **do**
    Sample minibatch of m noise samples $\{\mathbf{r}^{(1)}, ..., \mathbf{r}^{(m)}\}$.
    Sample minibatch of m true data examples $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$.
    Update the discriminator by ascending its stochastic gradient with learning rate α:

$$w_D \to w_D - \alpha \nabla_{w_D} L_D. \tag{4.3}$$

  end for
  Sample minibatch of m noise samples $\{\mathbf{r}^{(1)}, ..., \mathbf{r}^{(m)}\}$.
  Update the generator by ascending its stochastic gradient with learning rate α:

$$w_G \to w_G - \alpha \nabla_{w_G} L_G. \tag{4.4}$$

end for

This algorithm was extracted from [2].
The training of the network has to be well balanced since the generator is only as good as the discriminator is able to distinguish. On the other hand, if the discriminator is nearly perfect, the loss function vanishes, which reduces the gradient and therefore slows down the training significantly. This interplay of the two networks often leads to stability issues in the training [3].
One way to stabilize this is by defining the monotonous logit function of the discriminator variable $D(\mathbf{x})$,

$$\phi(\mathbf{x}) = \ln \frac{D(\mathbf{x})}{1 - D(\mathbf{x})}, \tag{4.5}$$

on which we can apply the gradient

$$\nabla \phi = \frac{1}{D(\mathbf{x})} \frac{1}{1 - D(\mathbf{x})} \nabla D(\mathbf{x}). \tag{4.6}$$

This gradient is very sensitive in the regions $D \to 0$ and $D \to 1$. We want to implement this gradient in the loss function of the discriminator L_D in a way that it applies to regions where the discriminator labels $D \approx 0$ to true data and $D \approx 1$ to generated data.

Therefore, we add it to the loss function with a prefactor λ_D and obtain the regularized Jensen-Shannon loss [28]

$$L_D \to L_D + \lambda_D \langle (1 - D(\mathbf{x}))^2 |\nabla\phi|^2 \rangle_{\mathbf{x}\sim P_T} + \lambda_D \langle D(\mathbf{x})^2 |\nabla\phi|^2 \rangle_{\mathbf{x}\sim P_G}. \tag{4.7}$$

The combination of these regularisations leads to a more stable training. There are other methods to stabilise the training of a GAN, for example the use of the Wasserstein distance [29]. Since this thesis is based on the the work of [3], we choose the regularised Jensen-Shannon GAN.

4.2. Usual application of GANs

We want to see if and how well the neural network can reproduce data without a weight. First, we have to generate toy data. Thus, we sample points according to a given function, in our case we choose a "camel-shaped" distribution (figure 4.2)

$$\begin{aligned} C(x) &= 0.3\, G(x,\, \mu = 0,\, \sigma = 0.5) + 0.7\, G(x,\, \mu = 2,\, \sigma = 0.5), \\ G(x,\, \mu,\, \sigma) &= \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}. \end{aligned} \tag{4.8}$$

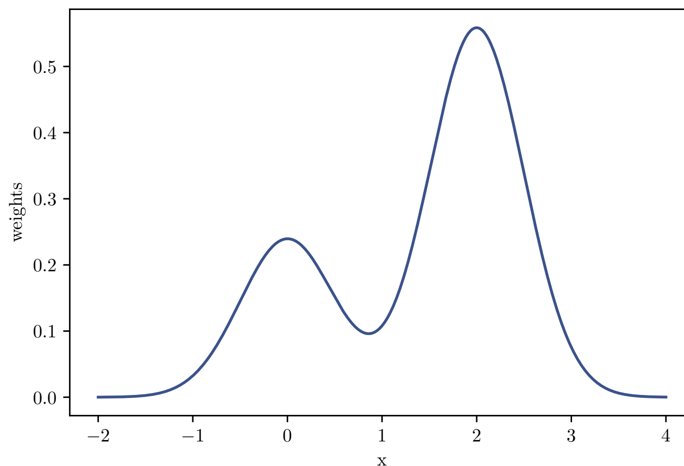

Figure 4.2: Plot of the camel function. This distribution will be the central toy example for further applications.

After obtaining one million points distributed by this function, we can try to reproduce them with our neural network.
First we have to choose the way we sample the noise $\{\mathbf{r}\}$. The noise sampling for the generator has an impact on our generated data. It is reasonable to sample gaussian distributed noise if we want to generate a gaussian-like distribution as well as it is reasonable

to sample uniformly distributed values as noise to generate a rectangle-like distribution, i.e. we choose gaussian noise for the camel function.
As an activation function in the hidden layers of the generator we choose the ELU-function. A major advantage of the ELU function is the already exponential shape. It is again obvious that this fits our problem pretty well, since we want to obtain an exponential shape (sum of two Gaussians). For the discriminator we choose the LeakyReLU in the hidden layers and the sigmoid in the output layer as activation functions. As was mentioned before, the sigmoid is necessary to make sure that the discriminator returns only values in $[0, 1]$.
Another important aspect is the training ratio between discriminator and generator. In this low-dimensional example, the generator has the easier job to do, so the discriminator can be tricked easily. To prevent this from happening, we train the discriminator more often than the generator, which leads to better and more stable results. In our case we choose a training ratio of 4 : 1, since we got good results with this setting.
We can now set up the networks by choosing different numbers of layers, units, learning rates and learning decays. For a neural network with 128 units in four hidden layers we get the reproduction shown in figure 4.3 after 800 training epochs.

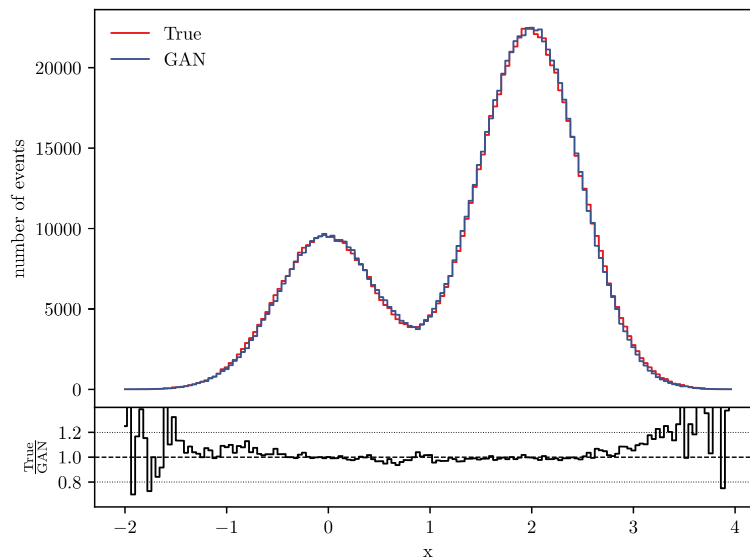

Figure 4.3: True and generated version of a camel distribution without weights. Their similarity is shown by dividing the distributions bin-wise in the lower half of the picture.

In the upper plot we see the "true distribution" and the distribution generated by our neural network. In the lower plot the similarity between these distributions is shown by dividing them. Since these values are close to one most of the time, we can say that

we get a good reproduction. The biggest problems of the ratio plot occur at the edges, where only a few data points are in each bin. We will come back to this later when we are going to calculate unweighting efficiencies.

4.3. Treating weighted data with GANs

To get unweighted data with our neural network, we change the loss function of the generator and the discriminator. Assuming we have $\{\mathbf{x}_T\}$ as input data with weights $\{w_T\}$ (which must not be confused with the weights in the neural network). We want to include these weights in our training procedure, to be precise, in the loss function. For high weights we want to gain a high importance in the loss function; for low weights a low importance. Therefore we define a relative weight for every datapoint, i.e.

$$w_{\text{rel}} = w_T / w_{T,\max}\,, \tag{4.9}$$

with $w_{T,\max}$ being the maximal weight in $\{w_T\}$. With this weight we can redefine our discriminator loss function, so the points with small relative weights have only little impact

$$L_{D,\text{uw}} := \langle -w_{\text{rel}} \log(D(\mathbf{x}))\rangle_{\mathbf{x}\sim P_T} + \langle -\log(1 - D(\mathbf{x}))\rangle_{\mathbf{x}\sim P_G}. \tag{4.10}$$

We are assuming that the desired weight of the generated data is constantly equal to one. If we wanted to generate a different weight distribution, we would also have to introduce weights for the generated data w_{gen}, modifying both loss functions to

$$L_{D,\text{uw}} := \langle -w_{\text{rel}} \log(D(\mathbf{x}))\rangle_{\mathbf{x}\sim P_T} + \langle -w_{\text{gen}} \log(1 - D(\mathbf{x}))\rangle_{\mathbf{x}\sim P_G}, \tag{4.11}$$

$$L_{G,\text{uw}} := \langle -w_{\text{gen}} \log(D(\mathbf{x}))\rangle_{\mathbf{x}\sim P_G}. \tag{4.12}$$

As an alternative to 4.10 it is also possible to use the weighted mean instead of the ordinary mean value. In this case we have no changes in L_G and for the discriminator loss we define

$$L_{D,\text{uw}} := \frac{\sum\limits_{\mathbf{x}\sim P_T} -w_{\text{rel}} \log(D(\mathbf{x}))}{\sum\limits_{\mathbf{x}\sim P_T} w_{\text{rel}}} + \langle -\log(1 - D(\mathbf{x}))\rangle_{\mathbf{x}\sim P_G}. \tag{4.13}$$

4.4. Testing the unweighting procedure

To test the unweighting procedure for a one dimensional toy model, we need to generate some toy data again. We sample uniformly distributed points in the interval $[-3, 5]$ and give them a weight. This weight is calculated with the "camel function" (equation 4.8). We can now use this data as input data for our neural network. Again we need to get a good combination of the hyperparameters, but we can just take the same as for the classical application of the GAN since we are dealing with a similar problem. So, again we get a good result for a neural network with 128 units in four layers. As we can see in figure 4.4, our neural network is able to unweight data.

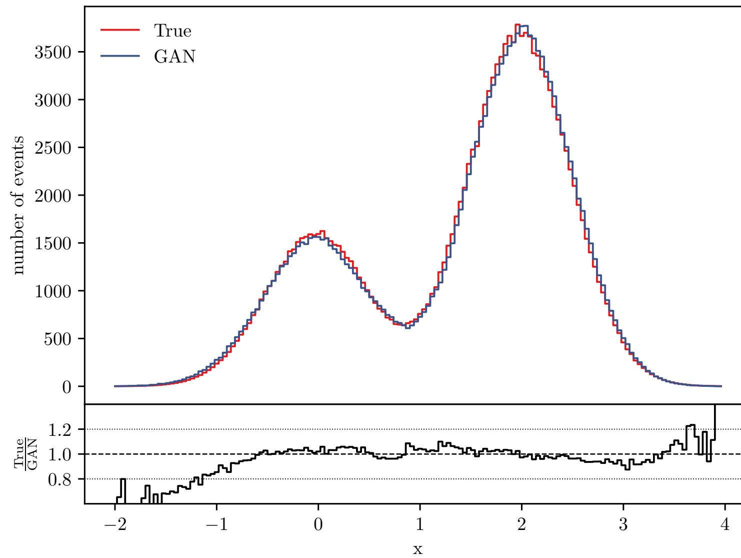

Figure 4.4: True and unweighted camel distribution function. Again the biggest uncertainties occur at the edges where only a few events are in each bin.

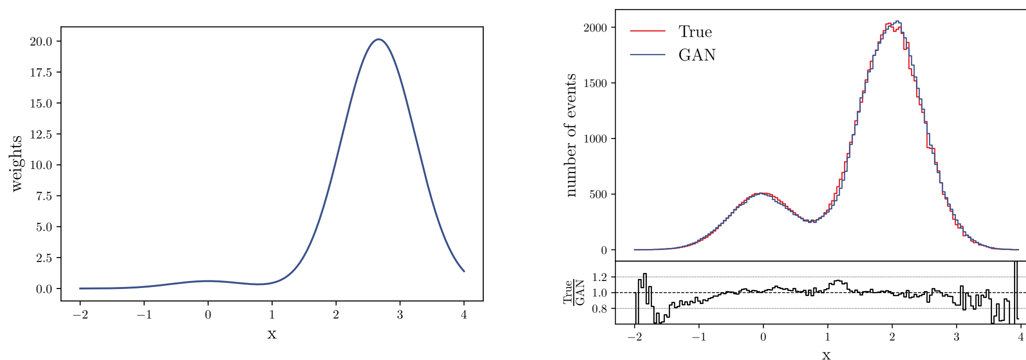

Figure 4.5: The left plot shows the weight distribution of our input data with the reduced camel function. The right plot shows the true and unweighted distribution after sampling the x values of the true data according to a Gaussian. The information is therefore encoded in both the distribution and the weights.

Realistic unweighting procedures

We have now shown that our neural network is able to reproduce a distribution independent of whether the information is encoded in the distribution itself or in an additional weight. We can now also test if the neural network is able to adapt information which is

encoded in both of it as this will be the case in later physical applications. Therefore, we sample Gaussian distributed points ($\mu = 0$, $\sigma = 1$) and weight it with a "reduced camel function" to obtain a camel shape. With equation 4.8 we define this function as

$$C_{\text{red}}(x) = \frac{C(x)}{G(x,\, \mu = 0,\, \sigma = 1)}. \tag{4.14}$$

The reduced camel function is shown on the left of figure 4.5. If we now unweight this distribution, we can see that the neural network has no preferences whether the information is encoded in the weights or the distribution (figure 4.5, right).

Overfitting

If we consider this neural network, we do not usually have to worry about overfitting. Overfitting in the discriminator is unlikely to happen since the input for the discriminator is always new data out of the generator, so the discriminator can only overfit concerning on the true data. Finally, the discriminator would not gain anything from this overfitting because it has to compare the true data with the generated data afterwards.
Overfitting in the generator is also unlikely to happen since the generator only generates unweighted data. This unweighted data can -in our setup of the loss function- not be distributed as the input data because the information is also encoded in the weights. Therefore, it is also for the generator highly unlikely to overfit.

Now that we have a properly working neural network for our unweighting procedure, we need to find a way to tell how well our neural network is working to compare it to other unweighting methods.

5. Unweighting efficiency

5.1. Implementing an efficiency

A standard unweighting technique is the hit-or-miss method, which was already mentioned in section 2.5. In detail this method consists of the following steps:

1. Generate data points $\{\mathbf{x}\}$ in a given Volume $V \subset \mathbb{R}^n$ with weights $s(\mathbf{x})$.
2. Determine maximal weight $s_{\max}$.
3. Generate a random point $\mathbf{x}_R$ in V and calculate the relative weight
$$s_{\text{rel}} = \frac{s(\mathbf{x}_R)}{s_{\max}}. \tag{5.1}$$
4. Generate a random number R in $[0,1]$ and compare it to the relative weight s_{rel}:
   - $s_{\text{rel}} > R$: "hit"
   - $s_{\text{rel}} < R$: "miss"
5. Only keep the "hit" events.

After this unweighting procedure one can simply divide the number of hits by the total number of points to get an unweighting efficiency. As one can imagine, this procedure is highly inefficient, leading to a massive data loss.
In our case we want to introduce a concept of an efficiency based on the same idea as the technique above. The whole neural network itself can be seen as one mapping function $\mathbf{f}$ of the noise $\{\mathbf{r}\}$ to the unweighted distribution $\{\mathbf{x}\}$. To calculate an efficiency as above, we need to determine the distribution function of the generator. One useful tool to obtain this is the following Lemma.

Lemma 1. *Let A, B be n-dimensional sets linked by a mapping function $\mathbf{f}: A \to B$ and $g_A(\mathbf{r})$ a probability density function on set A with $\mathbf{r} \in A$. The probability density $q_B(\mathbf{x})$ with $\mathbf{x} \in B$ resulting if $\mathbf{r}$ is sampled in A according to g_A and mapped to B is given by*

$$q_B(\mathbf{f}(\mathbf{r})) = \frac{g_A(\mathbf{r})}{|\partial \mathbf{f}(\mathbf{r})/\partial \mathbf{r}|}. \tag{5.2}$$

Proof. We can see that the subspace $[r_i, r_i + \mathrm{d}r_i]^n$ is mapped to the subspace $[f_i(\mathbf{r}), f_i(\mathbf{r}) + \mathrm{d}x_i]^n$ by the function $\mathbf{f}$. The volume of the first subspace is $\mathrm{d}V_r = \mathrm{d}r_1 \mathrm{d}r_2 ... \mathrm{d}r_n$, the volume of the second subspace is $\mathrm{d}V_x = \mathrm{d}x_1 \mathrm{d}x_2 ... \mathrm{d}x_n$. The total probability over the first subspace in A must be equal to the total probability over the second subspace in B, i.e.

$$P_{\text{tot}} = g_A(\mathbf{r}) \mathrm{d}V_r = q_B(\mathbf{x}) \mathrm{d}V_x. \tag{5.3}$$

Now we can replace $\mathbf{x}$ with $\mathbf{f}(\mathbf{r})$ and solve the equation according to $q_B(\mathbf{f}(\mathbf{r}))$

$$q_B(\mathbf{f}(\mathbf{r})) = \frac{g_A(\mathbf{r})}{\mathrm{d}V_x / \mathrm{d}V_r}. \tag{5.4}$$

The connection between the two sets A and B is per definition given by the coordinate transformation which affects the infinitesimal shifts by multiplying the Jacobi determinant $|\partial\mathbf{f}(\mathbf{r})/\partial\mathbf{r}|$ of the transformation function $\mathbf{f}$, i.e.

$$\mathrm{d}x_1\mathrm{d}x_2...\mathrm{d}x_n = |\partial\mathbf{f}(\mathbf{r})/\partial\mathbf{r}|\mathrm{d}r_1\mathrm{d}r_2...\mathrm{d}r_n \qquad \Leftrightarrow \qquad \frac{\mathrm{d}V_x}{\mathrm{d}V_r} = |\partial\mathbf{f}(\mathbf{r})/\partial\mathbf{r}|. \tag{5.5}$$

Therefore, we can conclude

$$q_B(\mathbf{f}(\mathbf{r})) = \frac{g_A(\mathbf{r})}{|\partial\mathbf{f}(\mathbf{r})/\partial\mathbf{r}|}. \tag{5.6}$$

□

This means that if we can get the Jacobian of our generator mapping function $\mathbf{f}$, we can calculate our generative distribution $q(\mathbf{x})$ from the noise distribution $g(\mathbf{r})$. In our case the noise distribution $g(\mathbf{r})$ is a Gaussian with the parameters $\mu = 0$ and $\sigma = 1$. To obtain the generator distribution $q(\mathbf{x})$, we generate an unweighted camel distribution $\mathbf{x} = \mathbf{f}(\mathbf{r})$ from random numbers $\{\mathbf{r}\}$ with the generator. Using the Jacobian of the generator mapping, we are now able to calculate the generative distribution according to Lemma 1 as

$$q(\mathbf{x}(\mathbf{r})) = \frac{g(\mathbf{r})}{|\partial\mathbf{f}(\mathbf{r})/\partial\mathbf{r}|} = \frac{G(\mathbf{r},\, \mu = 0,\, \sigma = 1)}{|\partial\mathbf{f}(\mathbf{r})/\partial\mathbf{r}|}. \tag{5.7}$$

Now we can also calculate the desired unweighted distribution $p(\mathbf{x})$, according to the camel function (equation 4.8) as

$$p(\mathbf{x}) = C(\mathbf{x}) = 0.3\, G(\mathbf{x},\, \mu = 0,\, \sigma = 0.5) + 0.7\, G(\mathbf{x},\, \mu = 2,\, \sigma = 0.5). \tag{5.8}$$

From these distributions we define an array of values by dividing them

$$s(\mathbf{x}) = p(\mathbf{x})/q(\mathbf{x}). \tag{5.9}$$

The closer the two distributions are together, the better our neural network unweighting algorithm works; in a perfect case $s(\mathbf{x})$ would be equal to one. We divide p by q because we want the generating distribution q never to undershoot the distribution p significantly since it is easier to cut an overshooting than producing additional events. We can interpret the array s as the resulting weight array after our unweighting procedure. In a perfect case, all weights would be constantly one. In reality, we always have values very close to one, so we consider the data to be unweighted. In the following we will refer to s as "weight distribution after the unweighting procedure." Nevertheless, we still consider the generated data as unweighted since s is just a tool to estimate the quality of our unweighting procedure.
Now we want to apply the hit-or-miss method to this distribution: we determine the maximum of the function $s(\mathbf{x})$, sample a random vector $\mathbf{x}_R$ from our given n dimensional subspace and sample another random number R in the interval [0,1]. Now we continue as described above with the hit-or-miss method and obtain an efficiency.
This method is only possible if the whole information is encoded in the weights. For a "hybrid" as discussed in section 4.4, we need to modify p by multiplying with the distribution of the x values with which we sampled our true data.

5.2. Testing the efficiency implementation

We can now calculate the unweighting efficiency for the camel function. Our expectation is a high efficiency because we got good fitting distributions after the unweighting (see figure 4.4).
However, we get an unweighting efficiency between 40% and 50% over several runs, which is lower than expected. The reason for that is obvious if we take a look at the distributions p and q in figure 5.1.

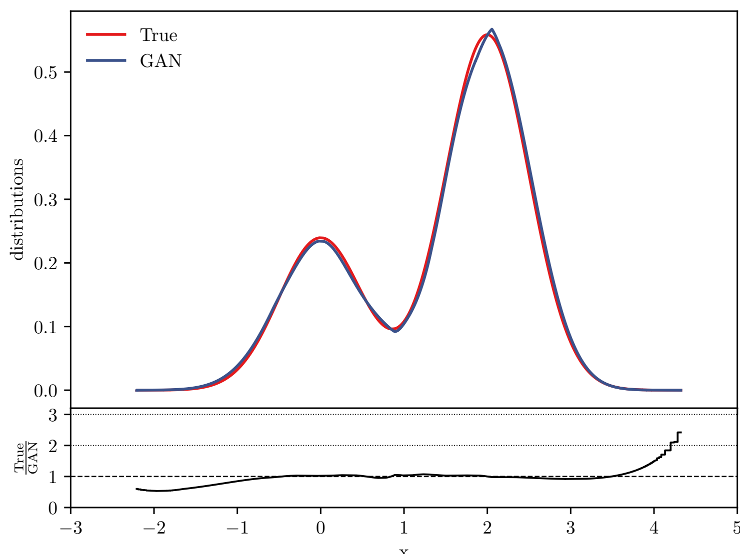

Figure 5.1: Desired distribution function p (labelled as "True") and generated distribution function q (labelled as "GAN"). In the upper plot both distributions are shown; in the lower plot the similarity between them is shown by dividing them. The curves are smooth because p and q are functions, not distributions.

Although the distributions look very similar, there are problems at the edges. There, the weight array $s = p/q$ reaches values up to $s_{\text{max}} = 2.5$. Since the efficiency calculation with the hit-or-miss method is strongly connected with this maximum, we get a low unweighting efficiency.
To solve this problem we need to cut off the edges by restricting the input noise $\{\mathbf{r}\}$ of the neural network. Until now, we have sampled this noise according to a gaussian distribution with the parameters $\mu = 0$ and $\sigma = 1$. Now we will again sample according to this gaussian distribution, but this time with the restriction that every value is in the interval $[\mu - 3\sigma, \mu + 3\sigma]$. The mistake we make through this condition is very small, because statistically speaking 99.73% of the points should be already in the interval $[\mu - 3\sigma, \mu + 3\sigma]$. This small adjustment has a huge effect. We now get an unweighting efficiency of 93%. In figure 5.2 we can see the cut-off at the edges.

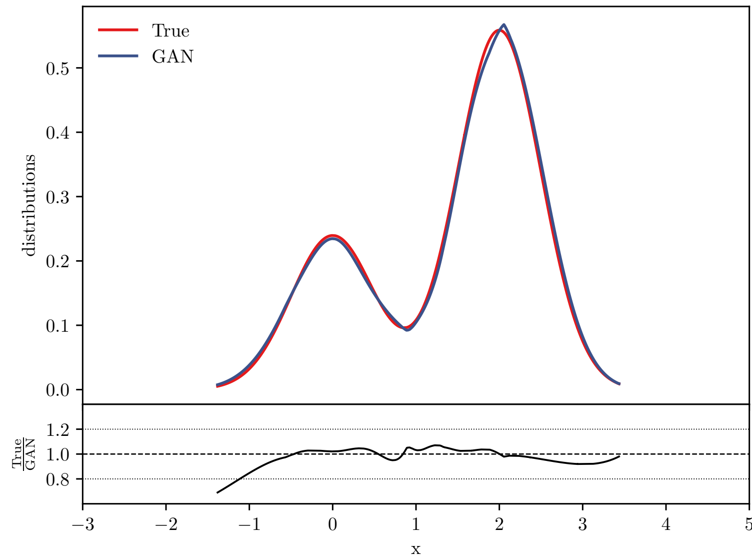

Figure 5.2: Desired distribution p and generated distribution q with a cut-off at the edges. The unweighting efficiency has to be higher since the ratio plot is very close to one.

This simple cut-off leads to a good unweighting efficiency. Nevertheless, we remark that our unweighting efficiency is an unstable value. For now this problem is easily solved by cutting of the edges, but later we will not be able to do so, for example if a distribution has a tail in the center which we cannot cut off. We have to prevent getting a false impression because of a misleading unweighting efficiency. Therefore, we will (after a cross check with VEGAS) take a look at the shape of the distribution s itself.

6. Cross check with VEGAS

6.1. The VEGAS algorithm

In this section we want to compare our efficiency results to the efficiency when we use the VEGAS [30] algorithm for unweighting. Therefore, we first take a brief look at the VEGAS algorithm itself, whose actual purpose is to calculate integrals. For example if we consider a one-dimensional integral

$$I = \int_a^b f(x)\,\mathrm{d}x. \tag{6.1}$$

The main idea of VEGAS is to put a uniformly distributed grid in the integration area; in the one dimensional case we divide the integration interval $[a, b]$ into n bins [31]

$$\begin{aligned} &x_0 = a, \\ &x_1 = a + \Delta x_0, \\ &\dots \\ &x_{n-1} = b - \Delta x_{n-1}, \\ &x_n = b. \end{aligned} \tag{6.2}$$

At first, we choose the length of the sectors $\Delta x_i = \frac{b-a}{n}$ to be the same. These sectors form a grid with n lines (where each line separates two sectors). We can give each line a value in the interval $[0, 1]$ by dividing the section number with n to

$$r_i = i/n\,. \tag{6.3}$$

It is now possible to perform a discrete coordinate transformation [31]

$$x_i \to r_i, \qquad x_i = a + (b-a)\frac{i}{n}. \tag{6.4}$$

The connecting Jacobian is piecewise constant, so we can calculate the integral in our new coordinate system as

$$\begin{aligned} &\int_0^1 f(x(r))\,J(r)\,\mathrm{d}r, \\ \text{with} \qquad &J(r) = J_i = n\Delta x_i, \qquad \frac{i}{n} < r < \frac{i+1}{n}. \end{aligned} \tag{6.5}$$

A simple Monte Carlo estimate with M sampling points r_i for the integral is given by

$$I = \frac{1}{M}\sum_{i=1}^{M} f(x(r_i))J(r_i). \tag{6.6}$$

The variance of this integral estimation with M different points is defined [31] as

$$\begin{aligned}\sigma_I^2 &:= \frac{1}{M}\left(\int_0^1 J^2(r)f^2(r)\mathrm{d}r - I^2\right)\\ &= \frac{1}{M}\left(\int_a^b J(r(x))f^2(x)\mathrm{d}x - I^2\right)\\ &= \frac{1}{M}\left(\sum_{i=1}^n J_i \int_{x_i}^{x_{i+1}} \mathrm{d}x f^2(x) - I^2\right).\end{aligned} \tag{6.7}$$

Trivially, this standard deviation is minimized when the following condition is fulfilled for all i

$$J_i \int_{x_i}^{x_{i+1}} \mathrm{d}x f^2(x) = n\Delta x_i \int_{x_i}^{x_{i+1}} \mathrm{d}x f^2(x) = \text{ constant.} \tag{6.8}$$

The VEGAS algorithm is programmed to adjust the grid via the increments Δx_i until this condition is nearly fulfilled. In the case of a perfectly placed grid with an infinite number of increments, the Monte Carlo estimate gives us the exact value of the integral. We want to achieve a tight grid with many sectors where the function comes to a maximum and few sectors in regions where the function is close to zero. As an example we show the binning for the camel function in figure 6.1.

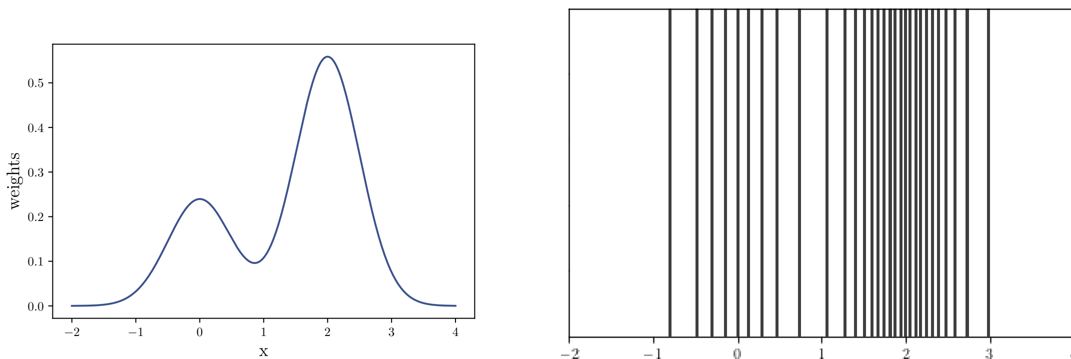

Figure 6.1: VEGAS binning for camel function. On the left the camel function is shown once again to compare it with the plot on the right, the grid that the VEGAS algorithm applies to the camel function.

6.2. Unweighting with VEGAS

Having set up a VEGAS integrator, we are now also able to unweight data by executing the following steps [31]:

1. Take the distribution function f (in our case the camel function) as input.
2. Train a VEGAS grid to fit the function f properly and obtain the Jacobian $J(r)$.
3. Generate uniformly distributed points $\{r\}$ in the interval $[0, 1]$.

4. Obtain x_{uw} from $\{r\}$ with the grid-mapping.

With the VEGAS algorithm we now have an array of unweighted points which are distributed according to the input function f. In analogy to our GAN efficiency we calculate an unweighting efficiency for VEGAS as well. According to Lemma 1, we calculate the generative distribution $q(x)$ with the Jacobian of the mapping. This leads to a weight array s in analogy to equation 5.9, with which we can calculate an efficiency according to the hit-or-miss method

$$s = \frac{p(x)}{q(x(r))} = \frac{f(x)}{q(x(r))} = \frac{f(x)}{\frac{1}{J(r)}} = f(x)\,J(r). \tag{6.9}$$

6.3. Results for the camel function

Now we can apply this VEGAS unweighting algorithm to the camel function. After some runs we can see that the efficiency gets better until it reaches 93%, but if we go on training the integrator, the efficiency gets lower again until it stagnates at around 80%. To understand this, we have to search for the connection between the training and the improvement/deterioration of the efficiency. The VEGAS integrator is designed to get a very tight grid in the highs, and a very wide grid in the tails. This means, the longer we train, the more points are taken away from the tails, which is fine to evaluate the integral, but bad if we want to get an unweighting efficiency close to one. The problem in the tails can be seen properly if we plot the ratio of the true and generated data (figure 6.2).

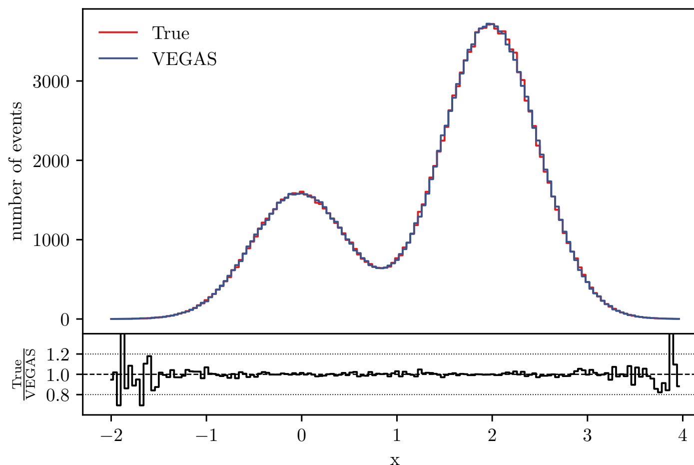

Figure 6.2: Desired and unweighted distribution with the VEGAS algorithm. In the lower plot again the ratio of the distributions.

The crucial point is the undershooting in the tails at the edges. Again, this leads to a low unweighting efficiency if we apply the hit-or-miss method.

Once more we can see that our defined unweighting efficiency does not necessarily tell us something about the quality of the unweighting procedure. Instead of focussing on getting every single weight close to one we should take a look at the weight distribution in total to get an impression of the unweighting quality.

6.4. Weight distribution after unweighting

Since the unweighting efficiency is not a reliable quantity, we plot the weight distributions s themselves for our neural network and for VEGAS (figure 6.3). In doing so, we can see how close we are getting to a constant weight of one.

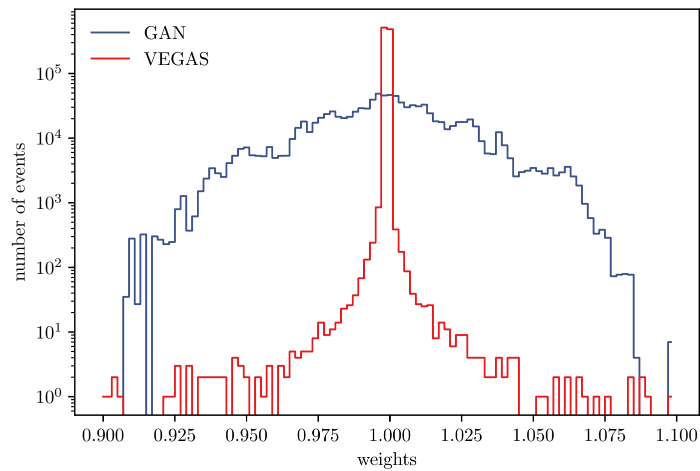

Figure 6.3: Weight distributions after the unweighting procedure with our neural network and with VEGAS.

We can conclude that in the one dimensional case VEGAS leads to a much better result since its weight distribution is tighter around one than the weight distribution of our neural network.

7. Two dimensional toy model

In the last sections we kept referring to the one dimensional camel function. Now we want to show that for higher dimensions our GAN can still perform a proper unweighting. This time we also expect a better performance than VEGAS since we are not restricted to the Cartesian coordinate system. Before getting into this in detail, we first construct a two dimensional toy example.

7.1. Generating toy data

To produce two dimensional toy data we generate uniformly distributed tupels $(x, y) \in [0, 1] \times [0, 1]$ and give them a weight. We choose the weight in a way that the points, which have a distance of $r_0 \approx 0.25$ to the point $(x_0, y_0) = (0.5, 0.5)$, have the highest weights, i.e we want to have a circular distribution of the generated data afterwards. We get these weights with the function

$$w_{2\mathrm{D}}(x, y) = \mathcal{N}_{2\mathrm{D}}\, e^{-\frac{1}{2\sigma^2}\left(\sqrt{(x-x_0)^2+(y-y_0)^2}-r_0\right)^2}. \tag{7.1}$$

In this function σ is the standard deviation and $\mathcal{N}_{2\mathrm{D}}$ is a normalisation factor, which we can determine via integration over the $\mathbb{R}^2$. For $r_0 = 0.25$ and $\sigma = 0.05$ we get $\mathcal{N}_{2\mathrm{D}} \approx 5.079$. We show the graph of the function in figure 7.1.

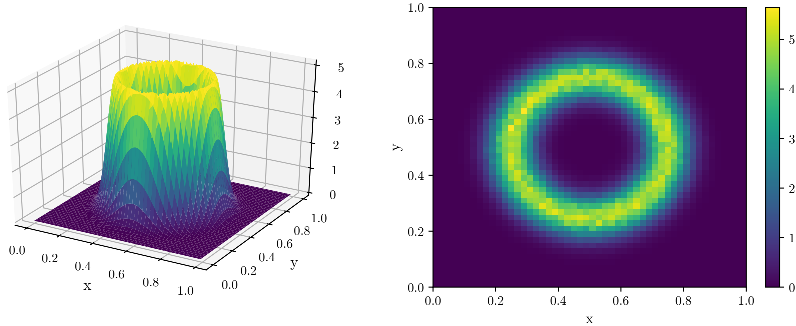

Figure 7.1: Surface plot (left) and two dimensional histogram (right) for the two dimensional toy model.

7.2. Unweighting results

We can now try to unweight the two dimensional toy model. As we still have data with gaussian peaks, we again choose gaussian distributed noise as input. The biggest difference to the one dimensional case is the choice of the activation function in the hidden layers of the generator. We choose the ReLU (equation 3.3) instead of the ELU (equation 3.5) function because we are now preparing some runs with thousands of epochs. In long runs, the training with the ReLU function is more stable than the training with the ELU

function. For a network construction with 256 units in eight layers we got good results for our unweighting procedure as we can see in figure 7.2.

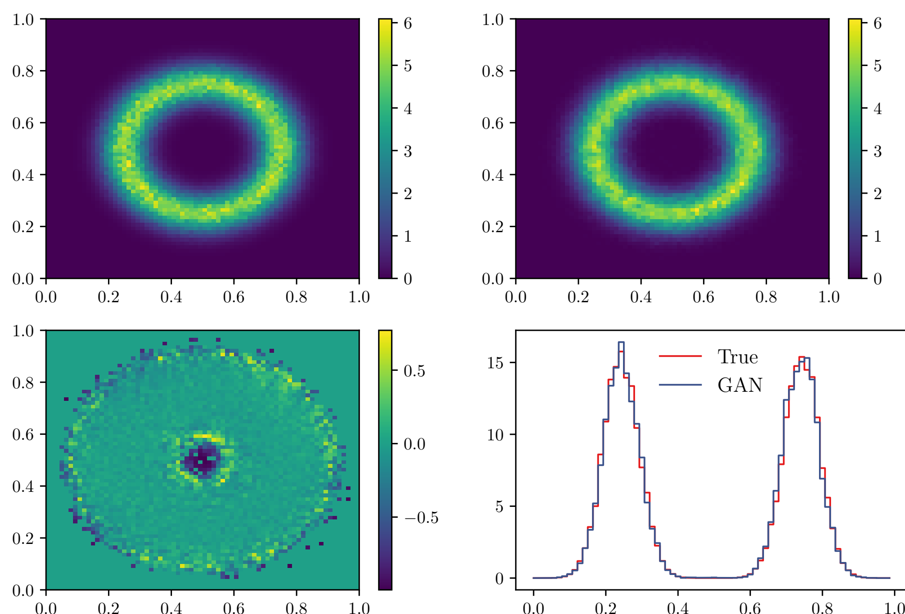

Figure 7.2: Results for the unweighting procedure in the two dimensional case. On the upper left and right the true and generated data are shown. On the lower left one can see a ratio plot to see how similar the two distributions are. On the lower right there is a slice at $x = 0.5$ to once again show similarities.

The ratio plot in the lower left was calculated as

$$\text{Value} = \frac{\text{GAN} - \text{True}}{\text{GAN} + \text{True}}. \tag{7.2}$$

If there are only generated and no true points in one bin, we set the value automatically to zero to prevent distractions far outside the ring.
In the lower right we plot a slice of the figure at $x = 0.5$. Here we can see how well the true and generated data match. We thus get also in the two dimensional case a very good result for our neural network too.

7.3. Efficiency calculation

The calculated efficiency (see section 5) is not as high as it was for the camel function because we have a tail in the middle. This tail is slightly undershooted and cannot be cut of with any σ condition for the input noise because the outer edges are not the problem.

Looking at the distribution of p and q in figure 7.3, we can see that our assumption about the problems arising in the center is correct. Therefore, we have to be satisfied with an unweighting efficiency of only 45%. Again we can see that our unweighting efficiency is not the best value for estimating the quality of our unweighting procedure.

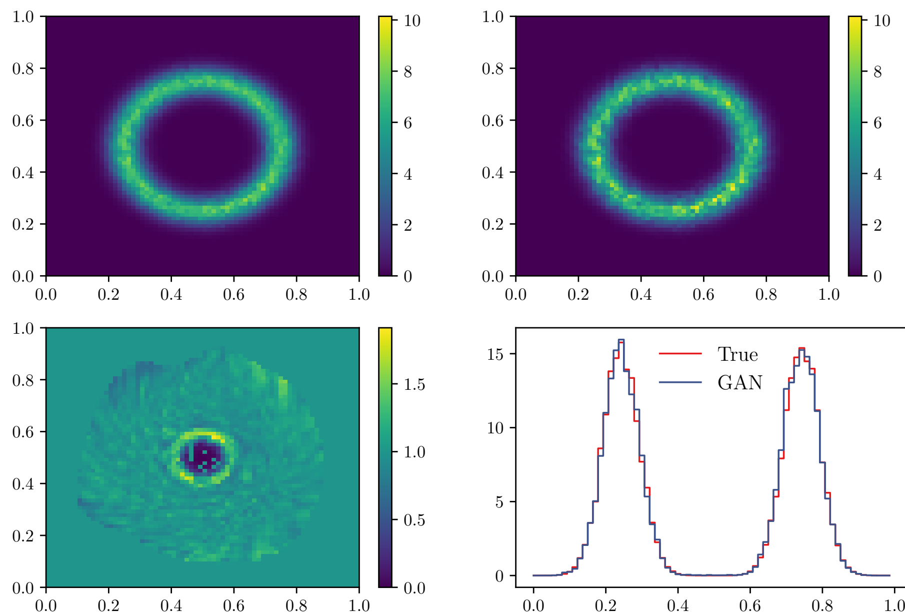

Figure 7.3: Desired distribution p of the true data and generated distribution q for the two dimensional toy model. On the upper left and right the distributions p and q are shown. On the lower left there is the fraction $s = p/q$ where one can see that the maximal values are reached in the center of the circle. On the lower right there is a slice at $x = 0.5$ of p and q.

7.4. VEGAS unweighting

The VEGAS algorithm in the two dimensional case has some obvious difficulties, which is why we expect VEGAS to perform a lot worse than our neural network. As we can see in figure 7.4, getting an optimal grid to approximate the circle is nearly impossible as we can only use the Cartesian coordinate axes to apply our grid. Therefore, the VEGAS integration has a lot of areas where the grid is very tight although there is not much information (for example at $(x_0, y_0) = (0.5, 0.5)$).
Now we apply the VEGAS unweighting algorithm. As we expected, the integrator is not capable of detecting the right shape. We can see in the impact of the applied grid in the final result in figure 7.5. Instead of the desired circle we get more of a quadratic plot.

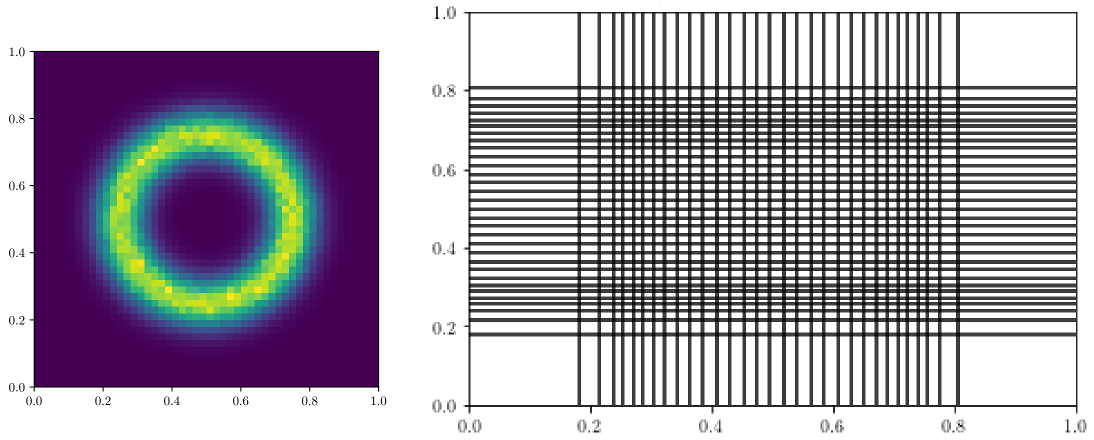

Figure 7.4: VEGAS binning for the two dimensional circle. On the left again the two dimensional toy model; on the right the VEGAS grid where a proper binning is not possible.

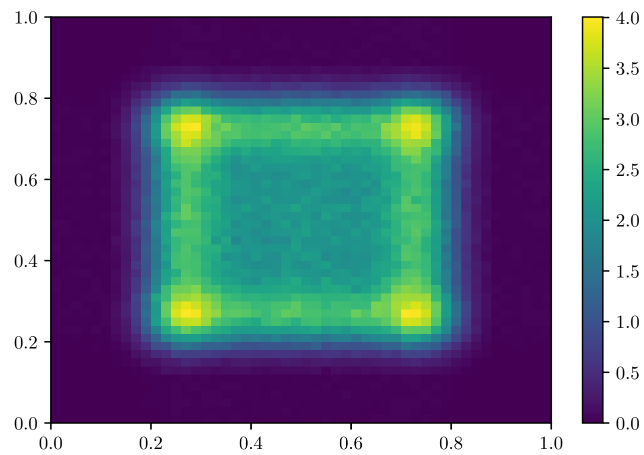

Figure 7.5: Result for the unweighting procedure with VEGAS for the two dimensional toy model.

Nevertheless, we have an unweighting efficiency of 15% for the VEGAS algorithm. This shows once more that the efficiency is not a useful value to describe the quality of the unweighting. We once again have to take a look at the weight distribution.

7.5. Weight distributions

Similar to the one dimensional case (section 6.4) we can plot the resulting weight array s of our neural network and compare it to the weights we obtain with VEGAS (figure 7.6). As expected, we get a gaussian-like result for the neural network and a highly irregular distribution for VEGAS, according to the unweighted distributions. A plot of the evolution of the GAN weight distribution with increasing epochs can be found in appendix A.1.

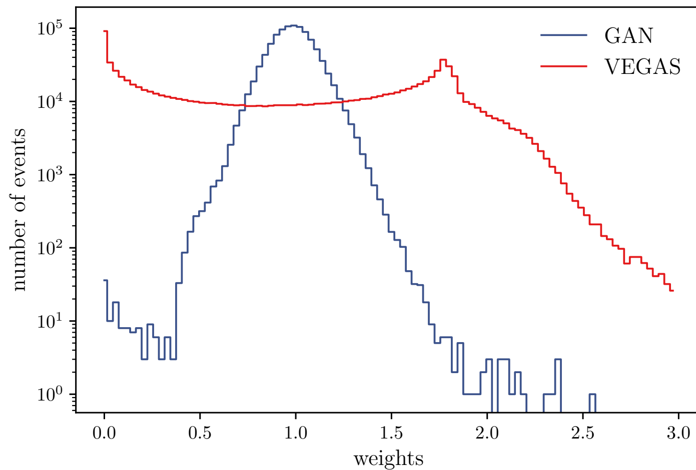

Figure 7.6: Weight distributions of the GAN compared to VEGAS in the two dimensional case. The neural network outperforms VEGAS by far. The VEGAS distribution was cut off; it would reach up to ten.

We can conclude that our neural network is clearly outperforming VEGAS in the two dimensional toy model. For higher dimensions the unweighting procedure is still working very well. An unweighting example of a three dimensional toy model can be found in appendix A.2.

Finally, having considered a lot of theory and two different toy examples with cross checks to the VEGAS algorithm, we can conclude that our neural network is in fact capable of unweighting data and is ready to be applied to a physical process.

8. Drell-Yan process

As was already mentioned in the introduction, we want to unweight events generated for the Drell-Yan process occurring during the scattering of

$$p + \bar{p} \to \mu^- + \mu^+. \tag{8.1}$$

First we generate weighted events with SHERPA (**S**imulation of **H**igh **E**nergy **R**eactions of **PA**rticles) [32]. We have to apply some cuts to the data because otherwise there would be disturbances in low energetic regions, coming from photonic scattering. These disturbances would cause our neural network to concentrate only on the shapes of low energetic regions and ignore the interesting, high energetic regions. In our case we chose the restrictions

$$p_{T,\mu} > 25\text{GeV}, \qquad |\eta_\mu| < 2.5, \tag{8.2}$$

with the transverse momentum $p_{T,\mu}$ and the pseudorapidity η_μ of the outgoing muon. Since the transverse momentum is cut we can conclude that the invariant mass $M_{\mu\mu}$ is also bounded from below

$$M_{\mu\mu} > 2\, p_{T,\mu,\text{min}}. \tag{8.3}$$

Since we are looking at a physical process, we want all outgoing particles to be on on-shell and momentum conservation to be fulfilled. As on-shell condition in the high relativistic case we get

$$E_{\mu^\pm} = \sqrt{(p^1_{\mu^\pm})^2 + (p^2_{\mu^\pm})^+ (p^3_{\mu^\pm})^2}. \tag{8.4}$$

Because the incoming particles only carry a momentum in beam direction (p^3), the momentum conservation leads to

$$p^1_{\mu^-} = -p^1_{\mu^+}, \qquad p^2_{\mu^-} = -p^2_{\mu^+}. \tag{8.5}$$

For the true data this is always guaranteed since it is implemented in SHERPA. To make sure that the generated data also fulfills the conditions, we implement a layer in front of the output layer, which sets the parameters $p^1_{\mu^+}$, $p^2_{\mu^+}$, E_{μ^+} and E_{μ^-} according to the conditions. We can now apply our unweighting procedure to this data and plot some observables of the process.
The distribution of the transverse momentum $p_{T,\mu}$ of the outgoing muons (figure 8.1, upper left) looks very good, only in the area around the cut (25GeV) some problems occur. This was to be expected since our neural network is not yet able to resolve such hard cuts in a better way. The distribution of the invariant mass $M_{\mu\mu}$ (figure 8.1, upper right) is slightly too low. In fact, this peak is hard to resolve for GANs in general. One way to deal with this would be by introducing an MMD (maximum mean discrepancy) loss [33], which has already been shown in [3]. Nevertheless, we can see the mass of the Z boson

$$M_Z = (91.1876 \pm 0.0021)\text{GeV [34]}. \tag{8.6}$$

Similar to the transverse momentum, the distribution of the pseudorapidity η_μ of an outgoing muon (figure 8.1, lower left) looks very good in general. Only in the area of the

cuts ($\eta = \pm 2.5$) the ratio plot deviates significantly from one.
Since the incoming particles only have a momentum in beam direction, the distribution of the azimuthal angles ϕ_μ of the outgoing muons (figure 8.1, lower right) is nearly constant. This property is also conserved after our unweighting procedure. In general, the distributions look all pretty good and we conclude that our neural network can also deal with a physical process and its restrictions.

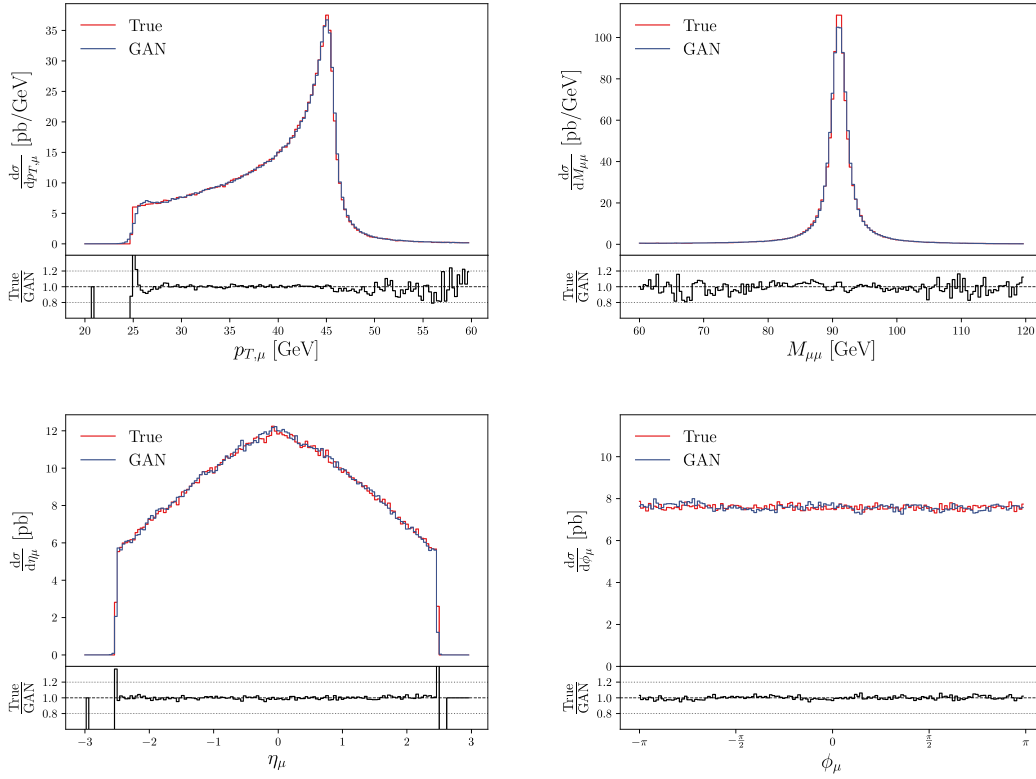

Figure 8.1: Plots of observable distributions. On the upper left the transverse momentum $p_{T,\mu}$ of the outgoing muon is shown, on the upper right the invariant mass $M_{\mu\mu}$ of the muons. On the lower left there is the pseudorapidity η_μ, on the lower right the azimuthal angle ϕ_μ of the outgoing muons.

9. Summary and outlook

The aim of this thesis was to build an unweighting procedure using generative adversarial networks. The weight of the input data was used to change the loss function of the discriminator to obtain a modified learning process. We were able to show that in low dimensional toy models our unweighting procedure returned excellent results. In the next step we compared our neural network with VEGAS, so we had to define an efficiency first. As it turned out, the efficiency we defined is not the best value to characterize the quality of the unweighted data, since it overrates problems in the tails of a distribution. Nevertheless, having plotted the weight distribution, we saw that VEGAS outperforms our neural network in the one dimensional case, but in higher dimensions our approach works better. Finally, we showed that it is possible to unweight simulated data for the Drell-Yan process.

Until now, the whole approach of using GANs to unweight data has been a success, but there have also been some difficulties. The most obvious problem is the setting of the hyperparameters since there are a lot of them. Of course, it is possible to have good guesses and with some experience one can find a good setting quicker. Nevertheless, this is the most time consuming step of this unweighting procedure.

After figuring out the unweighting process in the physical case, the next step would be to obtain the weight distribution. The challenge here is to get the desired distribution p of the unweighted data since we have to extract it from SHERPA. We are currently working on that.
Another challenge arises if we reconsider the plots in the physical case. We can see that the cuts in the transvers momentum and the pseudorapidity lead to bad results in these areas. Solving this problem needs a lot more effort and is another project our group is currently working on.

A. Appendix

A.1. Evolution of the weight distribution

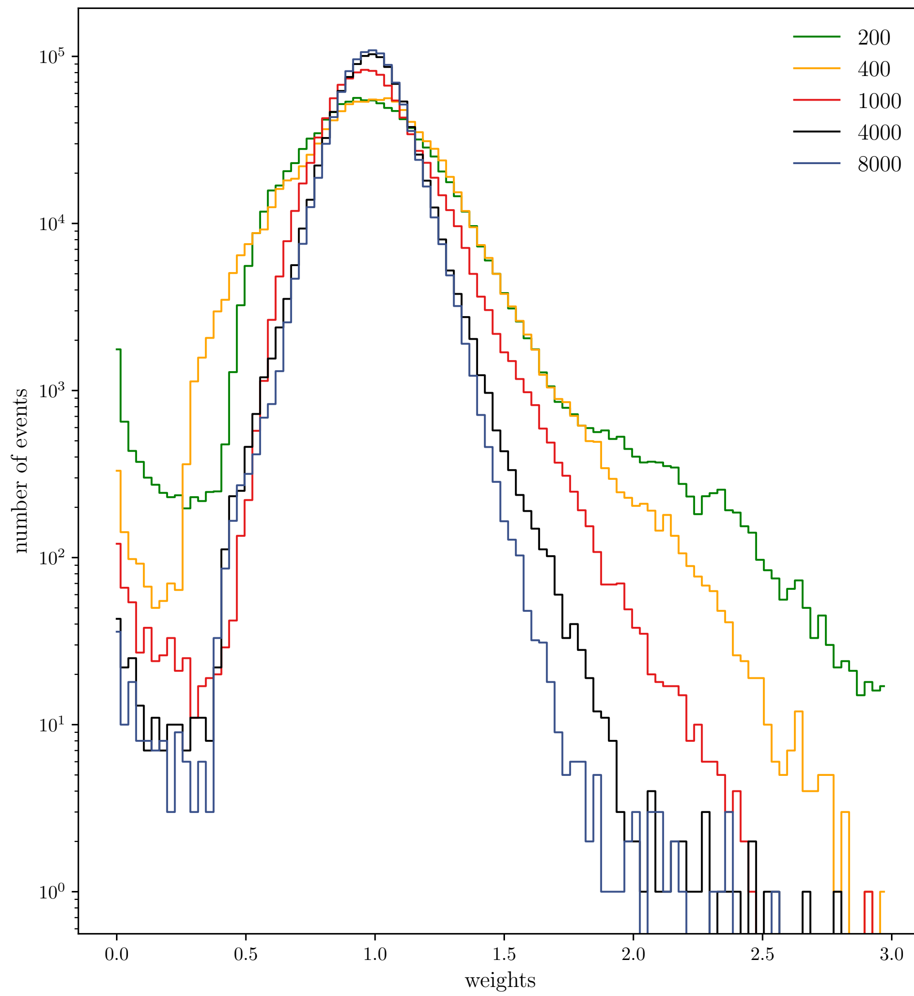

Figure A.1: Evolution of the weight distribution after different epochs for the two dimensional toy model. One can clearly see that the weight distribution approximates a gaussian distribution around one and gets thinner during the training.

A.2. Unweighting of a 3D toy model

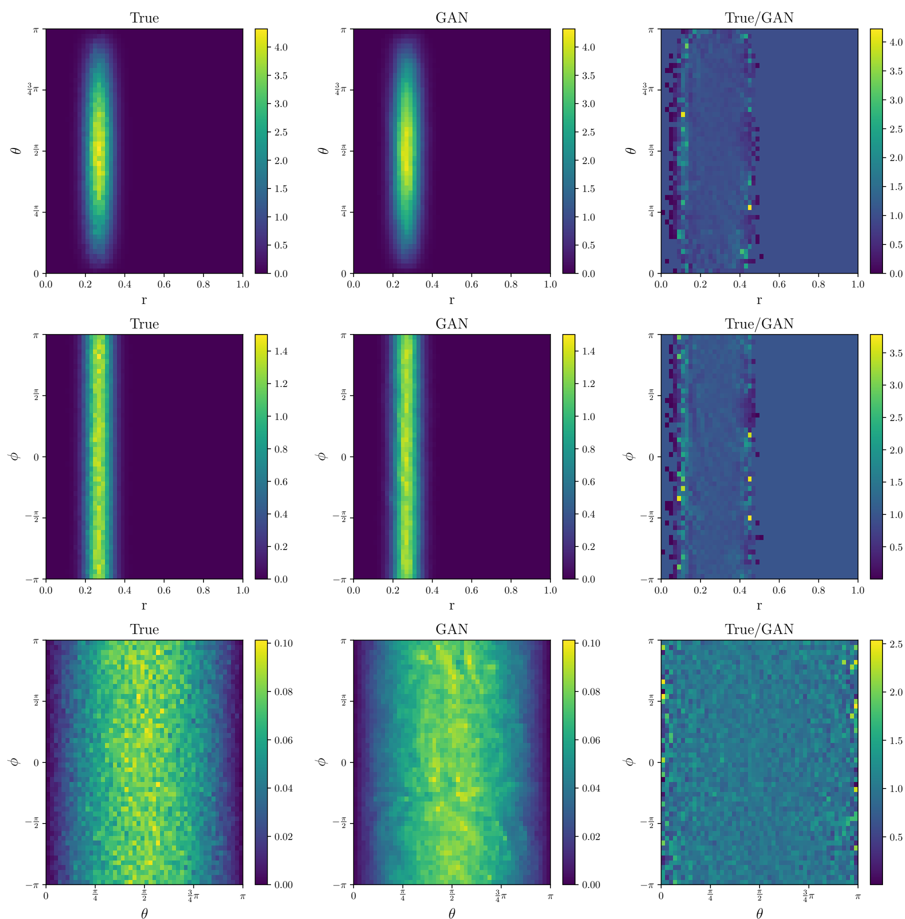

Figure A.2: True and generated distribution for a three dimensional toy model, illustrated as distributions of spherical coordinates (with the center shifted to $(0.5, 0.5, 0.5)$). Again the unweighting procedure is working well.

The three dimensional toy data is generated from the function

$$w_{3\mathrm{D}}(x, y, z) = \mathcal{N}_{3\mathrm{D}}\, e^{-\frac{1}{2\sigma^2}\left(\sqrt{(x-x_0)^2+(y-y_0)^2+(z-z_0)^2}-r_0\right)^2}, \tag{A.1}$$

with $(x_0, y_0, z_0) = (0.5, 0.5, 0.5)$, $r_0 = 0.25$, $\sigma = 0.05$ and the normalisation factor $\mathcal{N}_{3\mathrm{D}} \approx 9.768$.

A.3. Additional plots for the Drell-Yan process

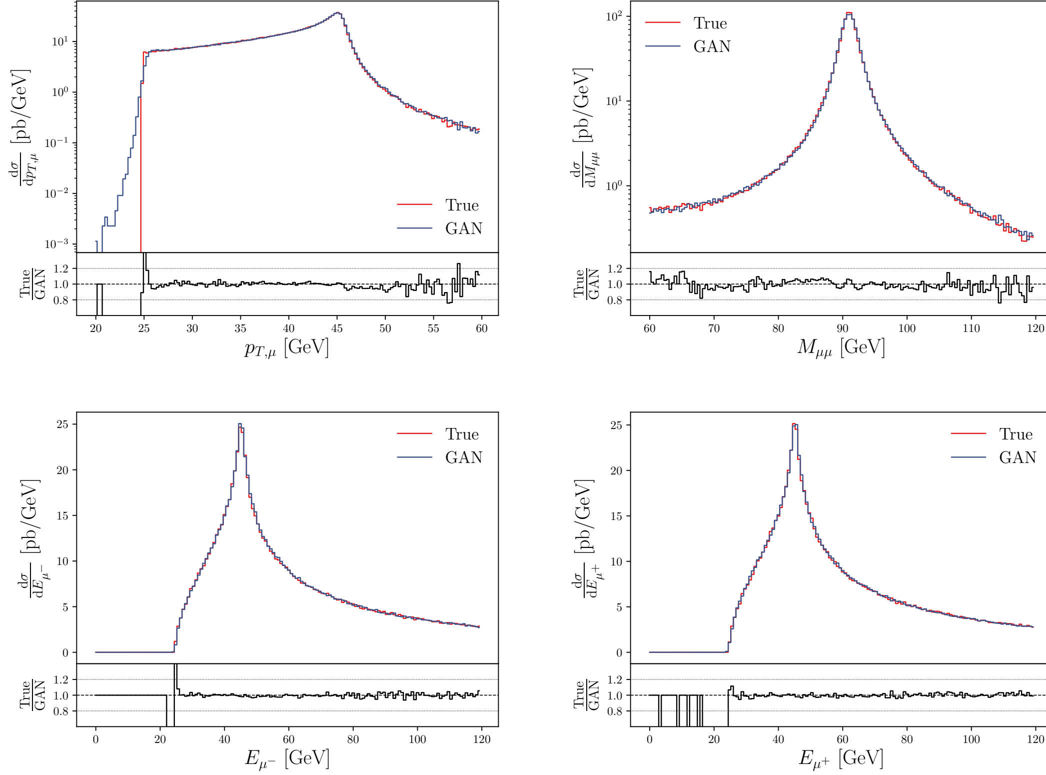

Figure A.3: Plots of observable distributions. On the upper left and right again the distributions of the transverse momentum $p_{T,\mu}$ and the invariant mass $M_{\mu\mu}$ are shown, but with a logarithmic scale. On the lower left there is the energy E_{μ^-} of the outgoing muons, on the lower right the energy E_{μ^-} of the outgoing antimuons.

References

[1] Plehn, T. (2015). *Lectures on LHC Physics.*
Springer. 2nd edition.

[2] Goodfellow, I.J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014) *Generative Adversarial Networks.*
Retrieved from https://arxiv.org/abs/1406.2661

[3] Butter, A., Plehn, T., & Winterhalder, R. (2019). *How to GAN LHC Events.*
SciPost Physics, 7(6). https://doi.org/10.21468/SciPostPhys.7.6.075

[4] Butter, A., Plehn, T., & Winterhalder, R. (2019). *How to GAN Event Subtraction.*
Retrieved from https://arxiv.org/abs/1912.08824

[5] Bellagente, M., Butter, A., Kasieczka, G., Plehn, T., & Winterhalder, R. (2019). *How to GAN away Detector Effects.*
Retrieved from http://arxiv.org/abs/1912.00477

[6] Peskin, M.E., Schroeder, D.V. (1995). *An Introduction to Quantum Field Theory.*
Avalon Publishing

[7] Drell, S., Yan, T.-M. (2014). *The Parton Model and its Applications.*
Retrieved from https://arxiv.org/abs/1409.0051

[8] Wu-Ki, T. (2009). *Bjorken scaling.*
Scholarpedia, 4(3):7412

[9] Dokshitzer, Y.L. (1977). *Calculation of the Structure Functions for Deep Inelastic Scattering and e+ e- Annihilation by Perturbation Theory in Quantum Chromodynamics.*
Sov.Phys.JETP, 46:641–653

[10] Placakyte, R. (2011). *Parton Distribution Functions.*
Retrieved from https://arxiv.org/abs/1111.5452

[11] Harland-Lang, L.A., Martin, A.D., Motylinski, P., Thorne, R.S. (2011). *Parton distributions in the LHC era: MMHT 2014 PDFs.*
Retrieved from https://arxiv.org/abs/1412.3989

[12] Sahoo, H. (2016). *Relativistic Kinematics.*
Retrieved from https://arxiv.org/abs/1604.02651

[13] Drell, S., Yan, T.-M. (1970). *Massive Lepton Pair Production in Hadron-Hadron Collisions at High-Energies.*
Retrieved from https://inspirehep.net/literature/60911

[14] Ledwig, O. (2017). *Next-to-leading order QCD corrections to the Drell-Yan process.*
Master thesis, WWU Münster

[15] UA1 Collaboration, CERN, Arnison, G. et al (1983). *Experimental observation of isolated large transverse energy electrons with associated missing energy at* $\sqrt{s} = 540$ *GeV.*
Phys. Lett. B122, 103

[16] UA2 collaboration, Banner, G. et al (1983). *Observation of single isolated electrons of high transverse momentum in events with missing transverse energy at the CERN* $\bar{p}p$ *collider.*
Phys. Lett. B122 476

[17] CDF Collaboration, Abe, F. et al. (1995). *Observation of Top Quark Production in* $\bar{p}p$ *Collisions with the Collider Detector at Fermilab*
Phys. Rev. Lett. 74, 2626

[18] Purkait, N. (2019). *Hands-On Neural Networks with Keras.*
Packt Publishing.

[19] Loy, J. (2019). *Neural Network Projects with Python.*
Packt Publishing.

[20] Nwankpa, C., Ijomah, W., Gachagan, A., & Marshall, S. (2018). *Activation Functions: Comparison of trends in Practice and Research for Deep Learning.*
Retrieved from http://arxiv.org/abs/1811.03378

[21] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning.*
MIT Press

[22] Choi, D., Shallue, C.J., Nado, Z., Lee, J., Maddison, C.J., & Dahl, G.E. (2019). *On Empirical Comparisons of Optimizers for Deep Learning.*
Retrieved from http://arxiv.org/abs/1910.05446

[23] Qian, N. (1999). *On the momentum term in gradient descent learning algorithms.*
Retrieved from https://doi.org/10.1016/S0893-6080(98)00116-6

[24] Kingma, D.P., & Ba, J.L. (2014). *Adam: A method for stochastic optimization.*
Retrieved from https://arxiv.org/abs/1412.6980

[25] Lawrence, S., & Giles, C.L. (2000). *Overfitting and Neural Networks: Conjugate Gradient and Backpropagation.*
Retrieved from https://ieeexplore.ieee.org/abstract/document/857823

[26] Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R.R. (2012). *Improving neural networks by preventing co-adaptation of feature detectors.*
Retrieved from http://arxiv.org/abs/1207.0580

[27] Song, H., Kim, M., Park, D., & Lee, J.-G. (2019). *How Does Early Stopping Help Generalization against Label Noise?*
Retrieved from http://arxiv.org/abs/1911.08059

[28] Roth, K., Lucchi, A., Nowozin, S., Hofmann, T. (2017). *Stabilizing Training of Generative Adversarial Networks through Regularization.*
Retrieved from https://arxiv.org/abs/1705.09367

[29] Arjovsky, M., Chintala, S., Bottou, L. (2017). *Wasserstein GAN.*
Retrieved from https://arxiv.org/abs/1701.07875

[30] Lepage, G.P. (1980). *VEGAS-An adaptive multi-dimensional integration program.*
Retrieved from https://cds.cern.ch/record/123074/files/clns-447.pdf

[31] Lepage, G.P. (2020). *vegas Documentation for Python.*
Retrieved from https://vegas.readthedocs.io/en/latest/index.html

[32] Freeman, P., Doe, S., Siemiginowska, A. (2001). *Sherpa: a mission-independent data analysis application.*
Proc. SPIE 4477, Astronomical Data Analysis.

[33] Gretton, A., Borgwardt, K.M., Rasch, M.J., Schölkopf, B., & Smola, A.J. (2008). *A Kernel Method for the Two-Sample Problem.*
Retrieved from https://arxiv.org/abs/0805.2368

[34] Tanabashi, M. et al. (2018). *Review of Particle Physics.*
Physical Review D. 98 (3): 030001.

Danksagung

Abschließend möchte ich mich bei allen bedanken, die mich während dieser Bachelorarbeit unterstützt haben.
Zunächst gilt mein Dank Tilman Plehn und Anja Butter, für die Möglichkeit in dieser Gruppe meine Bachelorarbeit zu schreiben, obwohl ich wegen der Coronapandemie lediglich online betreut werden konnte. Danke auch für das sehr interessante Thema und die vielen hilfreichen Gespräche, wenn es darum ging, was der nächste sinnvolle Schritt ist.
Ebenso bedanken möchte ich mich bei Ramon Winterhalder, der immer für alle meine Fragen und Probleme ein offenes Ohr hatte und mich wirklich tatkräftig unterstützt hat. Es hat großen Spaß gemacht an dem Projekt zusammen zu arbeiten!
Des Weiteren gilt mein Dank Timo Janssen, der uns bezüglich VEGAS und SHERPA geholfen hat.
Schlussendlich will ich mich noch bei all meinen Freunden bedanken, die Korrektur gelesen haben, sowie bei meiner Familie für die stetige Unterstützung während meines gesamten Studiums.

Erklärung

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

Heidelberg, den 21.06.2020,

Mathias Backes