Department of Physics and Astronomy
Heidelberg University

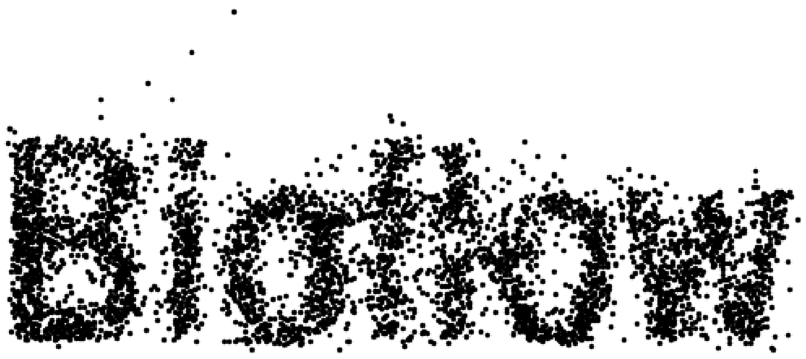
Bachelor Thesis in Physics
submitted by

Simon Pijahn

born in Erlangen (Germany)

2000

EndINNg Vegas
Event generation and integration with INNs



This Bachelor Thesis has been carried out by Simon Pijahn at the
Institute for theoretical Physics in Heidelberg
under the supervision of
Prof. Tilman Plehn

Abstract

In high Energy physics, integration of differential cross sections plays a major role, as well as sampling single unit-weighted events for a process. Algorithms like VEGAS provide adaptive Monte Carlo techniques to do this, however still face the issue of not being fully efficient in capturing non-factorizable integrands. With the raise of machine learning, projects like `i-flow` made it their mission to gain improved efficiency, with success. Nevertheless, training the implemented networks can take a long time, as evaluating the differential cross section is computationally expensive. We present a way to shorten this by generating less samples with invertible neural networks.

Zusammenfassung

In der Hochenergiephysik spielt die Integration von differentiellen Wirkungsquerschnitten eine wichtige Rolle, ebenso wie das Generieren einzelner einheitsgewichteter Ereignisse für einen Prozess. Algorithmen wie VEGAS bieten hierfür adaptive Monte Carlo Techniken, haben jedoch das Problem, dass sie bei der Erfassung nicht faktorisierbarer Integranden nicht vollständig effizient sind. Mit dem Aufkommen des maschinellen Lernens haben es sich Projekte wie `i-flow` zur Aufgabe gemacht, die Effizienz zu verbessern - mit Erfolg. Dennoch kann das Training der implementierten Netze viel Zeit in Anspruch nehmen, da die Auswertung des differentiellen Wirkungsquerschnitts sehr rechenintensiv ist. Wir stellen eine Möglichkeit vor, diese zu verkürzen, indem wir mit neuronalen Netzen weniger Ereignisse berechnen.

Contents

1 Introduction

Even though the Standart Model of particle physics describes very precisely the behavior of elementary particles, it still seems to be incomplete, as it can't yet describe some phenomena like dark matter or gravity. In the case of dark matter, many theoretical ideas exist to explain it, such as *Axions*, *WIMPS* or *supersymmetric particles* [1]. These beyond the Standart Model theories are constrained by experimental measurements, for example data taken by particle colliders like the LHC or astrophysical observations. The subject of connecting the theoretical prediction of the behavior of particles known to exist with experimental data is called *phenomenology*: Theoretical predictions of measurable quantities according to a new theory are calculated and compared with experiments. The calculation of the cross section or sampling unit weighted events for a certain process are widely used to check whether a theory is valid or to be rejected. The first calculation boils down to the integration of high dimensional, very complicated functions, the differential cross section, over the phase space of final state particles. Because these integrands are so complex, only rarely an analytical solution exists, raising the need of other means to obtain the integral. One common numerical and very stable way to do this, is to use Monte Carlo integration, as it makes very few assumptions about the integrand. It takes samples of the integrand and estimates the value of the integral with them. As the standard deviation scales to the inverse square root of number of samples taken, several variations of this approach exist to further minimize the error without the need of more samples (which can be computationally very expencive to calculate); in this thesis, it will be focused on importance sampling: By taking samples non uniformly in the integration space, but proportional to the value of the integrand, the variance can be greatly reduced. This has the side effect of greatly increasing the unweighting efficiency. There already exist some algorithms to automatically adapt the sample probability density, like VEGAS [2], however, it still struggles with high dimensional integrals, as it assumes the integrand to be separable in each dimension, which is most often not the case in interesting physical situations.
In recent years, the field of machine learning has made great progress, and neural networks can be viable alternatives to previously mentioned algorithms, as they can provide a highly adaptable mapping between two integration spaces. One existing program is `i-flow` [3], which also is trained by taking samples of the function, but does not suffer from the separability assumption of VEGAS any more. However, in this thesis, it is proposed that this project can be improved: As mentioned above, evaluating the function can take a lot of time, and `i-flow` only trains by taking samples, but with *invertible* neural networks, another training direction is possible by going over already sampled points again and still adapt the mapping, without the need of an expensive calculation.

In this thesis, first, the theoretical background of Monte Carlo techniques, physical processes, event generation and invertible neural networks are explained; after that, suitible program pipelines are sketched. For a baseline, VEGAS will be tested first on a toy function, then on the physical process $\mathbf{gg} \rightarrow \mathbf{ggg}$. To keep things more tangible, the proposed invertible neural network is also precisely examined on a toy function and compared to the benchmark produced by `i-flow`. Even though the benchmark could not be achieved, to show the feasibility of this project, the proposed innovation is shown to work.

2 Essentials

2.1 Monte Carlo techniques

Monte Carlo Integration

The integral can be illustrated by the area under the integrand. It equals the average value of the function multiplied with the integration volume. Adapting Monte Carlo techniques, random samples in the volume can be taken and the integrand evaluated at that point. Repeated many times, this can produce an approximation of the Integral: Consider

$$I = \int_\Omega f(u')du' \tag{1}$$

where $f : \Omega \subset \mathbb{R}^d \to [0, \infty)$, and Ω is typically an unit hypercube. The estimation of the Integral, given N uniform random samples $u_i \in \Omega$, can be calculated via

$$I \approx E_N = \frac{|\Omega|}{N} \sum_{i=1}^{N} f(u_i) = |\Omega| \langle f \rangle \overset{|\Omega|=1^d}{=} \langle f \rangle \tag{2}$$

$|\Omega|$ denotes the Volume of Ω, which is in the unit hypercube case equal to 1^d, hence it will not longer be mentioned if not necessary. The standard deviation of E_N is then given by

$$\sigma_N(f) = \sqrt{\frac{V_N(f)}{N}} = \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \tag{3}$$

with V_N the corresponding variance. [4]

Importance sampling

Importance sampling aims to reduce the variance of the Monte Carlo Integral estimation (taking the same number of Samples). Great variations in $f(u_i)$ lead to a great variance; in the optimum case, $f(u_i) = \text{const} \to V_N(f) = 0$. This can be achieved by a non-uniform sampling probability density, taking more samples where the value of the function is bigger. Mathematically, this corresponds to a change of variables. Instead of integrating directly uniformly over Ω, a positive definite mapping $G(u) : \Omega \to \Omega$, is introduced, and

$$I = \int_\Omega \frac{f(u')}{g(u')} g(u')du' = \int_\Omega \frac{f(u')}{g(u')} dG(u') = \langle \frac{f}{g} \rangle \tag{4}$$

such that g is as close as possible to f. Samples are taken according to $G(u)$ and weighted inversly by $g(u) = dG(u)/du$. The optimum case of a constant integrand corresponds to $g = f$, but this would require knowing the Integral beforehand. Finding the best choice of G now is the main quest to minimalize the standard deviation of the integral estimation, which is one measure of success for this project. [5] [6]

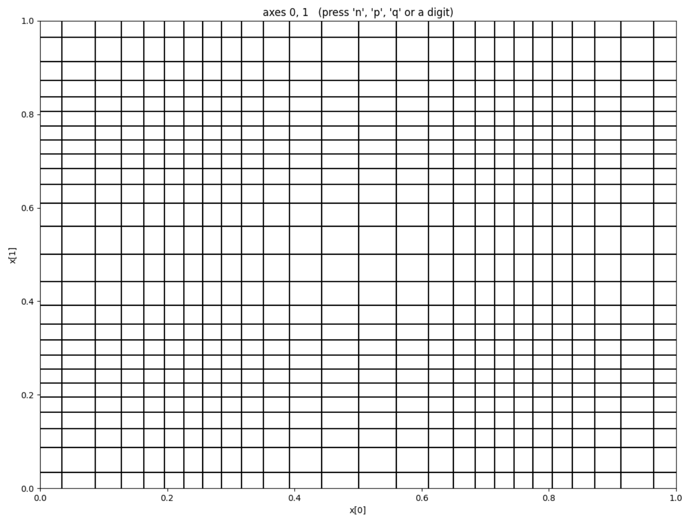


Figure 1: Vegas grid adapted to a double gaussian function: two artificial peaks visible

Vegas

A popular Monte Carlo integration alorithm utilizing importance sampling is Vegas. It was invented in 1978 by G. P. Lepage [2] to automatically adapt the sampling density to the integrand. To this end, it uses a grid in each dimesion, evaluates the mean value of the integrand in each cell and adapts the cell volume inversely to it. After several iterations (most times 5-10 iterations are sufficient and the grid is fully adapted), an adaptive monte carlo integration using the adapted grid map and its inverse jacbobian jac_{inv}^{vegas} is performed. As vegas assumes that the integrand is factorizable, it is a weakness of this algorithm, as shown in Figure 1 . The function the map adapted to is:

$$\begin{aligned} f(\mathbf{x}) =&\exp\left(\frac{-((x_1-0.25)^2+(x_2-0.25)^2)}{0.25^2}\right) \\ &+\exp\left(\frac{-((x_1-0.75)^2+((x_2-0.75)^2)}{0.25^2}\right) \end{aligned} \tag{5}$$

featuring only two peaks, however Vegas "sees" four peaks, and therefore its efficiency suffers.

2.2 Physical Background

Crosssection and Matrix element

This section follows closely the derivation of [7].

In high energy physics, one important observable in scattering experiments is the probability for a process to take place, for example in the case of two incoming particles:

$$\mathcal{P} = |\,\langle \underbrace{\phi_1\phi_2\ldots}_{future}\,|\,\underbrace{\phi_{\mathcal{A}}\phi_{\mathcal{B}}}_{past}\rangle\,|^2 \tag{6}$$

One can describe this likelihood in terms of the cross section σ. It is defined as follows: Consider two bunches $\mathcal{A}, \mathcal{B}$ of lenth $\ell_{\mathcal{A}}, \ell_{\mathcal{B}}$, density $\rho_{\mathcal{A}}, \rho_{\mathcal{B}}$ and identical cross-sectional Area A, that are colliding with some velocity v. Then,

$$\sigma \equiv \frac{\text{Number of scattering events}}{\rho_{\mathcal{A}}\rho_{\mathcal{B}}\ell_{\mathcal{A}}\ell_{\mathcal{B}}A} \tag{7}$$

To compute this quantity, one has to set up the incoming particles, evolve them in time with the correct time evolution operator conatining information about the implemented theory, and overlap overlap with the final-state particles.
A state is represented by its wavefunction:

$$\begin{aligned} |\phi\rangle &= \int \frac{d^3k}{(2\pi)^3}\frac{1}{\sqrt{2E_{\mathbf{k}}}}\phi(\mathbf{k})\,|\mathbf{k}\rangle \\ \langle\phi|\phi\rangle = 1 \qquad &\text{if} \qquad \int \frac{d^3k}{(2\pi)^3}|\phi(\mathbf{k})|^2 = 1 \end{aligned} \tag{8}$$

Where $|\mathbf{k}\rangle$ is a one-particle state of momentum $\mathbf{k}$ in the corresponding theory and $\phi(\mathbf{k})$ is the Fourier transform of the spatial wavefunction. Adopting the convention of collinear Beams (forced by setting the impact parameter $\mathbf{b} = 0$) and writing an explicit factor $\exp(-i\mathbf{b}\cdot\mathbf{k}_{\mathcal{B}})$ to take the spatial translation of the beams before the collision into account, setting up the initial state is straightforward:

$$|\phi_{\mathcal{A}}\phi_{\mathcal{B}}\rangle_{\text{in}} = \int \frac{d^3k_{\mathcal{A}}}{(2\pi)^3}\int \frac{d^3k_{\mathcal{B}}}{(2\pi)^3}\frac{\phi_{\mathcal{A}}(\mathbf{k}_{\mathcal{A}})\phi_{\mathcal{B}}(\mathbf{k}_{\mathcal{B}})e^{-i\mathbf{b}\cdot\mathbf{k}_{\mathcal{B}}}}{\sqrt{(2E_{\mathcal{A}})(2E_{\mathcal{B}})}}\,|\mathbf{k}_{\mathcal{A}}\mathbf{k}_{\mathcal{B}}\rangle \tag{9}$$

It is now convenient to set the initial and final state to a definite momentum, to be able to calculate the transition amplitude between them. To compute it, the initial state is evolved in time and the S-Matrix identified:

$$\begin{aligned} {}_{\text{out}}\langle\mathbf{p}_1\mathbf{p}_2\ldots|\mathbf{k}_{\mathcal{A}}\mathbf{k}_{\mathcal{B}}\rangle_{\text{in}} &= \lim_{T\to\infty}\langle\underbrace{\mathbf{p}_1\mathbf{p}_2\ldots}_{T}|\underbrace{\mathbf{k}_{\mathcal{A}}\mathbf{k}_{\mathcal{B}}}_{-T}\rangle \\ &= \lim_{T\to\infty}\langle\mathbf{p}_1\mathbf{p}_2\ldots|\,e^{-iH(2T)}\,|\mathbf{k}_{\mathcal{A}}\mathbf{k}_{\mathcal{B}}\rangle \\ &\equiv \langle\mathbf{p}_1\mathbf{p}_2\ldots|\,S\,|\mathbf{k}_{\mathcal{A}}\mathbf{k}_{\mathcal{B}}\rangle \end{aligned} \tag{10}$$

The S-Matrix contains a trivial non-interacting part, corresponding to no scattering taking place, and an interesting part containing the interactions:

$$S = \mathbb{1} + iT \tag{11}$$

Therefore we can extract the interaction and implement 4-momentum conservation:

$$\langle\mathbf{p}_1\mathbf{p}_2\ldots|\,iT\,|\mathbf{k}_{\mathcal{A}}\mathbf{k}_{\mathcal{B}}\rangle = (2\pi)^4\delta^{(4)}(\mathbf{k}_{\mathcal{A}} + \mathbf{k}_{\mathcal{B}} - \sum\mathbf{p}_f)\cdot i\mathcal{M}(k_{\mathcal{A}}k_{\mathcal{B}} \to p_f) \tag{12}$$

All 4-momenta correspond to real particles and are on-shell. $\mathcal{M}$ is the Matrix element specifying the process and calculated from Feynman Diagrams. In this thesis, this will not be treated for longer; in the realization of the project, it will be computed by MadGraph [8].

To connect the Matrix element to the cross section, the probability for an initial state $|\phi_{\mathcal{A}}\phi_{\mathcal{B}}\rangle$ to scatter into an infititesimal final state phase space volume $d^3p_1 \dots d^3p_n$ is considered:

$$\mathcal{P}(\mathcal{AB} \to 1 \dots n) = \left(\prod_f \frac{d^3p_f}{(2\pi)^3} \frac{1}{2E_f} \right) |_{\text{out}} \langle \mathbf{p}_1 \dots \mathbf{p}_n | \phi_{\mathcal{A}}\phi_{\mathcal{B}} \rangle_{\text{in}} |^2 \tag{13}$$

And for a single target $\mathcal{A}$ and many incident particles $\mathcal{B}$ with different impact parameters $\mathbf{b}$, the Number of events is

$$N = \sum_{\text{incident particles}\, i} \mathcal{P}_i = \int d^2 b n_{\mathcal{B}} \mathcal{P}(\mathbf{b}) \tag{14}$$

Where $n_{\mathcal{B}}$ is the area density of $\mathcal{B}$. It is assumed to be constant to arrive at the cross section

$$\sigma = \frac{N}{n_{\mathcal{B}} N_{\mathcal{A}}} = \frac{N}{n_{\mathcal{B}}} = \int d^2 \mathcal{P}(\mathbf{b}) \tag{15}$$

Now, puting the Equations 9, 13 and 15 together,

$$d\sigma = \left(\prod_f \frac{d^3p_f}{(2\pi)^3)} \frac{1}{2E_f} \right) \int d^2 b \left(\prod_{i=\mathcal{A},\mathcal{B}} \int \frac{d^3k_i}{(2\pi)^3} \frac{\phi_i(\mathbf{k}_i)}{\sqrt{2E_i}} \int \frac{d^3\tilde{k_i}}{(2\pi)^3} \frac{\phi_i^*(\tilde{\mathbf{k}_i})}{\sqrt{2\tilde{E_i}}} \right) \\ \times e^{i\mathbf{b}\cdot(\tilde{\mathbf{k}_i} - \mathbf{k}_i)} \left({}_{\text{out}} \langle \{\mathbf{p}_f\} | \{\mathbf{k}_i\} \rangle_{\text{in}} \right) \left({}_{\text{out}} \langle \{\mathbf{p}_f\} | \{\tilde{\mathbf{k}_i}\} \rangle_{\text{in}} \right)^* \tag{16}$$

is obtained,
where $\tilde{\mathbf{k}_i}$ are dummy integration variables. Investigating only interaction processes, the identity Matrix in Equation 11 can be dropped, and the 6 integrals can be performed. The one with $\tilde{k_i^z}$ gives a factor $\frac{1}{|v_{\mathcal{A}} - v_{\mathcal{B}}|}$, wich is the relative beam velocity as viewed from the laboratory frame. In high energy physics, the beams travel with approximately $c \equiv 1$, the factor approximates to one. The final form of the crosssection yields:

$$d\sigma = (2\pi)^4 \delta^4(\mathbf{p}_{\mathcal{A}} + \mathbf{p}_{\mathcal{B}} - \sum_f \mathbf{p}_f) \frac{|\mathcal{M}(p_{\mathcal{A}}, p_{\mathcal{B}} \to \{p_f\})|^2}{2E_{\mathcal{A}}E_{\mathcal{B}}} \left(\prod_f \frac{d^3p_f}{(2\pi)^3} \frac{1}{2E_f} \right) \tag{17}$$

Parton Distribution Function

As the name suggest, in hadron colliders like the LHC, hadrons are collided. They have internal structure, as they are made out of quarks and gluons, its so-called *Partons.* The probability that a hadron with momentum (P) contains a parton of type f with longitudal momentum Fraction $\xi \in [0, 1]$ is given by the *Parton Distribution Function* (short *PDF*, not to be confused with probability density function) [7]:

$$\text{PDF}_f(\xi) d\xi \tag{18}$$

gg → ggg

The process of two colliding gluons, producing an additional gluon, is chosen because of two reasons: Gluons dont have any rest mass, simplifying the phase space generation in the project, and because it is a rather simple QCD process. As it has for each final state particle three degrees of freedom, as they are on-shell, and also has to obey four momentum conservation, it has in total five degrees of freedom. In real collider experiments, there are two additional degrees of freedom for the input momenta. The Matrix element $\mathcal{M}(g_\mathcal{A} g_\mathcal{B} \to g_1 g_2 g_3)$ is provided by MadGraph. Paying attention to the PDFs, the differential cross section yields

$$d\sigma = \frac{|\mathcal{M}(g_\mathcal{A} g_\mathcal{B} \to g_1 g_2 g_3)|^2 PDF_\mathcal{A}(\xi_1) PDF_\mathcal{B}(\xi_2)}{2\xi_1\xi_2\sqrt{s}^2} dX$$
$$dX = (2\pi)^4 \delta^{(4)}(\mathbf{g}_\mathcal{A} + \mathbf{g}_\mathcal{B} - \sum_{f=1}^{3} \mathbf{g}_f) \left(\prod_{f=1}^{3} \frac{d^3 g_f}{(2\pi)^3} \frac{1}{2E_f} \right) d\xi_1 d\xi_2 \tag{19}$$

Where $\sqrt{s}$ is the collider energy, $\xi_{1,2} \in [0,1]$ are the momenta fractions from the hadron of $\mathbf{g}_{\mathcal{A},\mathcal{B}}$ and the integration variable dX handling conservation laws and all factors of 2π.
It has two infrared divergences, obviously for $\xi_{1,2} \to 0$, implicitly in the matrix element for the angular seperation $\Delta R \to 0$ of the final state particles. To take these into account, phase space cuts in the pahse space generator described in the next section are implemented.

Phase Space Generators

Integration Volumes, such as the phase space for $gg \to ggg$, rarely take the form of an unit hypercube, yet most implementations work within this case to maintain flexibility over a wide range of tasks. This introduces the need for a phase space generator. The mapping must be surjective to guarantee full phase space coverage.
Calculations should be as easy as possible, so choosing the center of mass frame and boosting the momenta to laboratory frame afterwards is instructive. Beginning with the 4-vector (E, p_x, p_y, p_z), first of all, the process has a rotational symmetry around the beamaxis (chosen to be the z axis), so its convenient to parametrise the momentum components (p_x, p_y) of final state particles to momentum in transversal direction, p_T, and an azimuthal and longitudinal angle (ϕ, θ). Now, the four components E, p_T, θ, ϕ are obtained:

$$E = \sqrt{1 + \sin(\theta)}\ \ p_T \qquad p_T = \sqrt{p_x^2 + p_y^2}$$
$$\theta = \sin^{-1}(\frac{p_T}{\sqrt{p_T^2 + p_z^2}}) \qquad \phi = \sin^{-1}(\frac{p_x}{\sqrt{p_x^2 + p_y^2}}) \tag{20}$$

The longitudinal angle can be further parametrised to the commonly used pseudorapitity $\eta \equiv -\log(\tan(\frac{\theta}{2}))$, where the difference of two values is Lorentz invariant under a boost along the longitudinal axis. The angular seperation can then be defined as well:

$$\Delta R_{i,j} = \sqrt{\Delta\eta_{i,j}^2 + \Delta\phi_{i,j}^2} \tag{21}$$

For the boost to the laboratory frame in z-direction, knowledge of the input momenta fractions (as viewed from the lab frame) $\xi_{1,2}$ is required:

$$\frac{v}{c} = \beta = \frac{p_{z,lab}}{E_{lab}} = \frac{\xi_1 - \xi_2}{\xi_1 + \xi_2} \frac{(\xi_1 + \xi_2)\sqrt{2}/2}{(\xi_1 + \xi_2)\sqrt{2}/2} = \frac{\xi_1 - \xi_2}{\xi_1 + \xi_2} \tag{22}$$

The "RAMBO on diet" [9] algorithm taking the code from [10] is adapted . It is a bijective and (for massless particles, as in this project) flat. In this project, it takes seven random numbers $r_i \in (0, 1)$ as input (edges are avoided, as they are a problem for most phase space generators), five for the degrees if freedom of the final state particles, and two for the PDFs of the incoming particles and produces valid sets of 4-momenta, associated with a weight containing all factors of Equation 19, as well as the phase space volume for each event,

$$\begin{aligned} V_n &= \left(\frac{\pi}{2}\right)^{n-1} \frac{(Q^2)^{n-2}}{(n-1)!(n-2)!} \\ Q &= (\xi_1 + \xi_2)\frac{\sqrt{s}}{2} \\ w_{\text{RAMBO}} &= \frac{(\pi/2)^{n-1} Q^{n-2}}{(2\pi)^(3n-4)(n-1)!(n-2)!} \end{aligned} \tag{23}$$

to yield flat sampling. Phase space cuts, such as minimum colliding momenta fractions, angular seperations to avoid infrared divergences or minimal transversal momentum to only examine events with more interesting physics are implemented by multiplying the weight with zero.
The Code for the creation of a set of 4-momenta at defined collision energy works (for two final state particles) as follows:

Algorithm 1 Rambo for n=2

```
Require: $r_1, r_2, E_{cm}$
  $M_1 \leftarrow E_{cm}$, $M_2 \leftarrow 0$
  $\cos\,\theta \leftarrow 2r_1 - 1$
  $\sin\,\theta \leftarrow \sqrt{1 - \cos^2\theta}$
  $\phi \leftarrow 2\pi r_2$
  $q_2 \leftarrow \frac{M_1}{2} = \frac{E_{cm}}{2}$
  $\boldsymbol{p}_1 \leftarrow q_2 \begin{pmatrix} \cos\,\phi \sin\,\theta \\ \sin\,\phi \sin\,\theta \\ \cos\,\theta \end{pmatrix} = \frac{E_{cm}}{2} \begin{pmatrix} \cos\,\phi \sin\,\theta \\ \sin\,\phi \sin\,\theta \\ \cos\,\theta \end{pmatrix}$
  $p_1 \leftarrow (q_2, \boldsymbol{p}_1)$, $Q_2 \leftarrow (q_2, -\boldsymbol{p}_1)$
  (boost by (1,0,0,0))
  $p_2 \leftarrow Q_2$
```

The Boost to the laboratory frame requires two additional random numbers to define the input momenta fractions:

Algorithm 2 Boost to lab frame

```
Require: $p_{in1}, p_{in2}, p_{out}, r_3, r_4$
  reflab $\leftarrow (p_{in1} \cdot r_3 + p_{in2} \cdot r_4)$
  if reflab[0] $< 0$ or (reflab)$^2 < 0$ then
      print(invalid boost)
  $\boldsymbol{\beta} \leftarrow$ reflab[1 :]/reflab[0]
  $\gamma \leftarrow \frac{1}{\sqrt{1-\boldsymbol{\beta}^2}}$
  for $p$ in $p_{out}$ do
      $\beta_p \leftarrow \boldsymbol{\beta} \cdot p[1:]$
      $p[0] \leftarrow \gamma(p[0] + \beta_p)$
      $\gamma' \leftarrow (\gamma - 1)/\boldsymbol{\beta}^2$
      $f \leftarrow \beta_p \gamma' + \gamma p[0]$
      $p[1:] \leftarrow p[1:] + f\boldsymbol{\beta}$
  return $p_{out}$
```

2.3 Unweighting

As shown above, each phase space point has multiple weights associated: For the Integration its weighted with the inverse jacobian of the variable transformantion j_{inv} and the value of the function to be integrated, here $d\sigma(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$ as well as the weights from the phase space generator w_{Rambo}. For the overall weight of an event, these are multiplied:

$$w_{event} = j_{inv} \cdot d\sigma(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3) \cdot w_{\text{Rambo}} \tag{24}$$

However, in real experiments, each event has unit weight, whereby the necessity of an unweighting procedure is presented. One algorithm is the hit-and-miss procedure: the weights are scaled to $[0, 1]$ and random number pairs are sampled: one to set the phase space point, another one to compare the weight with. Only samples where the random number is lower than the sample weight are saved. The unweighting efficiency is given by [4]

$$\epsilon_{unwgt} = \frac{\langle w_{event} \rangle}{\max(w_{event})} \tag{25}$$

The procedure is illustrated in Figure 2. Out of 300 samples, 143 were accepted, resulting in an unweighting efficiency of ≈ 48 %, with a mean weight of 0.49. Clearly, isolated events with exceptionally high weight punish the unweighting procedure; in the optimum case each event has already unit weight before the unweighting prcedure, resulting in a δ-distribution of event weights. Approaching this form is another measure of success of this project.

2.4 INNs

In this project, invertible neural network's (INNs) utilizing coupling layers are implemented. First of all, the setup of a neural network is explained.

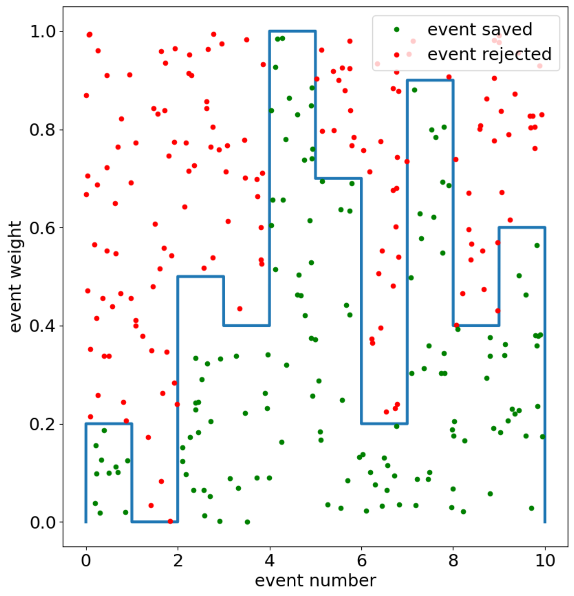


Figure 2: Hit-and-miss procedure

Neural Networks

A neural network is a function that can transform a vector. It consits of an input layer $\mathbf{i}$, an output layer $\mathbf{o}$ and at least one hidden layer. Each layer consists of nodes, that represent the networks neurons. Nodes are connected between the layers, with an associated weight w_j and bias b_j. Each node h_j follows an activation function a,

$$h_j = a\left(\sum_k w_{j,k} h_k + b_j\right) \tag{26}$$

where k is the summation index for all input nodes for the neuron j. The hyperparameter $w_{j,k}$ for one layer can be expressed as matrix $\mathbf{W}$, b_j as vector $\mathbf{b}$. One layer can now be expressed as:

$$\mathbf{h}_{\text{layer j}} = a\left(\mathbf{W}\ \mathbf{h}_{\text{layer k}} + \mathbf{b}\right) \tag{27}$$

Often, like in this project, the ReLU function is used for a. It vanishes for input smaller than 0 and grows linear for input bigger than 0. For the output layer the activation function is not used, and the values are in the range $(-\infty, \infty)$. To transform them to a unit hypercube, an output function is used, e.g. a scaled tanh or the errorfunction.

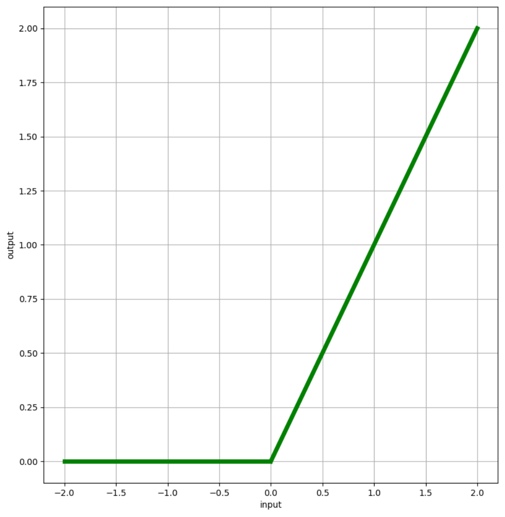


Figure 3: ReLU activation function

Coupling layers

Figure 4 shows how an INN with an easy to compute jacobian can be achieved: The input vector $\mathbf{x}$ is split into two parts. One part is used as input for a neural network. Its output vector is channeled through a softmax function to be normalized to $[0, 1]$ and used as arguments for a coupling transform C applied to $\mathbf{x}_B$. At the end, the vector is permuted randomly to ensure that over the span over a few blocks, the whole vector $\mathbf{x}$ is transformed. The jacobian for one such block G then evaluates to:

$$\left|\frac{\partial G(\mathbf{x})}{\partial \mathbf{x}}\right|^{-1} = \left|\begin{pmatrix} \mathbb{1} & 0 \\ \frac{\partial C}{\partial m}\frac{\partial m}{\partial \mathbf{x}_A} & \frac{\partial C}{\partial \mathbf{x}_B} \end{pmatrix}\right|^{-1} = \left|\frac{\partial C(\mathbf{x}_B; m(\mathbf{x}_A))}{\partial \mathbf{x}_B}\right|^{-1} \tag{28}$$

Evaluating Equation 28 does not require the computation of the gradient of $(m(\mathbf{x}_A))$, which

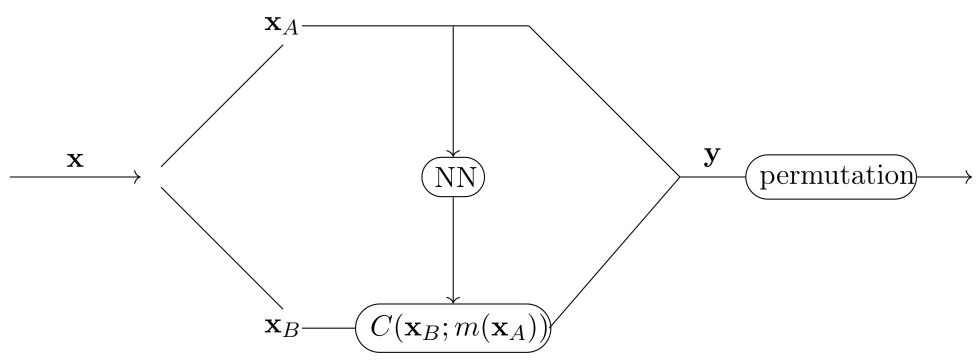


Figure 4: Schema of a coulping block

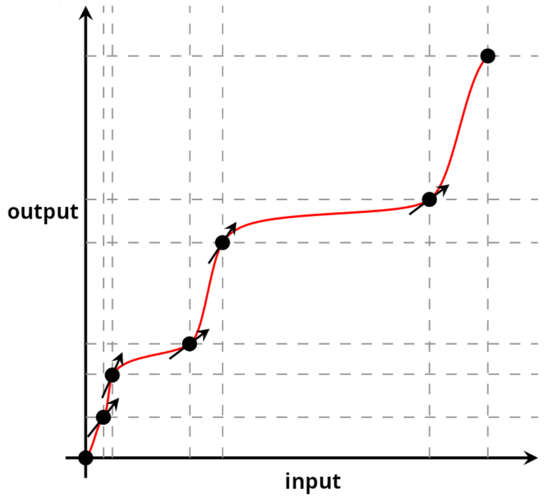


Figure 5: Rational Quadratic Spline Fit, [12]

would scale as $\mathcal{O}(D)^3$, with D the number of dimensions. This normalizing flow method only scales lineary and is handy even for higher dimensions [11].
In practice, a Rational Quadratic Spline Coupling is used for C. Given the number of grid points (bins) n it fits to, it needs $3n - 2$ parameters, so this is the output dimensionality of the neural network. $2n$ Dimensions are required to determine the location of the grid points, and $n - 2$ Dimensions for the slope at the gridpoints, that are not on the edge (there it is set to 1). In Figure 5 a sketch is drawn, arrows indicate the slope at grid points.

Loss and optimizer

The network structure however is not enough on its own, weights and biases are initalised randomly resulting in a random output of the network. It has to be trained, the network hyperparameters have to be adapted to the desired output. This is achieved by the loss and the optimizer. The loss is a function of the measure of success, and can take many forms. Indispensable is, that the loss is minimal, if the network is optimally tuned.

The loss function is minimalized with the optimizer, which tunes the hyperparameters. One commonly used algorithm for this is the ADAM-algorithm also implemented in this project. It utilizes first-order gradient descent and adaptive momenta (hence the acronym) to minimize the loss, requiring little memory. Typical and well-tested Values for the parameters of the algorithm itself are $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$ [13]:

Algorithm 3 ADAM

```
Require: $\alpha$: Stepsize
Require: $\beta_1 \beta_2 \in [0, 1)$: Exponential decay rates for the momentum estimates
Require: $f(\theta)$: Differentiable loss function with hyperparameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialise first momentum vector)
  $v_0 \leftarrow 0$ (Initialise second momentum vector)
  $t \leftarrow 0$ (Initialise timestep)
  while $\theta_t$not converged do :
      $t \leftarrow t + 1$
      $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients at timesstep $t$)
      $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first momentum estimate)
      $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw momentum estimate)
      $\hat{m}_t \leftarrow m_t/(1 - \beta_1^t)$ (compute bias-corrected first momentum estimate)
      $\hat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ (compute bias-corrected second raw moment estimate)
      $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)
  return $\theta_t$: (resulting parameters)
```

Training

As the invertible network can run in both directions, training can also take place in both directions.

Generative Training: During the generative training, samples in the phase space are taken according to the current probability distribution of the network: Points are sampled randomly according to a Prior distribution in the latent space (unphysical space) and sent through the network to determine the corresponding phase space point. The loss can be directly one measure of success, like the standard deviation of the integral or some form of the unweighting efficiency, or a more abstract divergence of the test probability distribution of the network (Q) and the true probability distribution of the function (P), such as the Kullback-Leibler-Divergence

$$D_{KL}(P||Q) = \sum_{x \in X} P(x) \log \left(\frac{P(x)}{Q(x)} \right) \tag{29}$$

where X is the probability space [14].

Recycling Training: In the other direction, the network can be trained by iterating over the during generative training sampled phase space points: It sends them in the other direction through the network, calculating the corresponding latent space points (which are not necessarily the same point they got sampled at, because the network is dynamic during training), and uses the divergence of this latent space distribution and the Prior for the loss. This is a big novelty in this project: The Program can benifit from the expensive calculation it has already done before by evaluating the matrix element $\mathcal{M}$ with a relatively cheap and fast computation. In theory, this could reduce training time for neural networks greatly.

3 Program Pipelines

3.1 Vegas

Now, everything can be put together. First of all, to obtain a widely used and well tested baseline, a pipeline for the integration of the differential cross section is set up. For this project, it is sufficient to produce weighted events, the unweighting will in practice be done by implementation of other programs.
The pipeline for the Vegas implementation is shown in Figure 6. Vegas needs the dimensionality d and the integrand with the center-of-mass energy E_{cm} as input. For each iteration j, it produces a set number of samples in the unit hypercube according to its grid distribution (r-space), which are then translated to the physical phasespace with Rambo. The differential cross section for each sample is evaluated; the matrix element $\mathcal{M}$ for the chosen process ($\mathbf{gg} \rightarrow \mathbf{ggg}$) is provided by MadGraph, the parton density functions should be provided by LHAPDF, however later was found, that using the "Tilman gluon distribution function" $\text{TDF}(r_i) = \frac{1}{r_i^2}$ produced more stable outputs. Then, an adaptive Monte Carlo integration is performed and the evaluations of $\mathcal{M}$ together with the Rambo weights are returned, so the grid cell volumes can be adapted, and the next iteration starts. After a set number of iterations, the algorithm is stopped, and samples can be taken to be unweighted later. For the final result of the integral, every integral evaluation is considered and weighted by its inverse variance [15] [16]:

$$\begin{aligned} \sigma &= \frac{1}{\sum\limits_j \frac{1}{\text{Var}(\sigma_j)}} \cdot \sum_j \frac{\sigma_j}{\text{Var}(\sigma_j)} \\ \text{Var}(\sigma) &= \frac{1}{\sum\limits_j \frac{1}{\text{Var}(\sigma_j)}} \end{aligned} \tag{30}$$

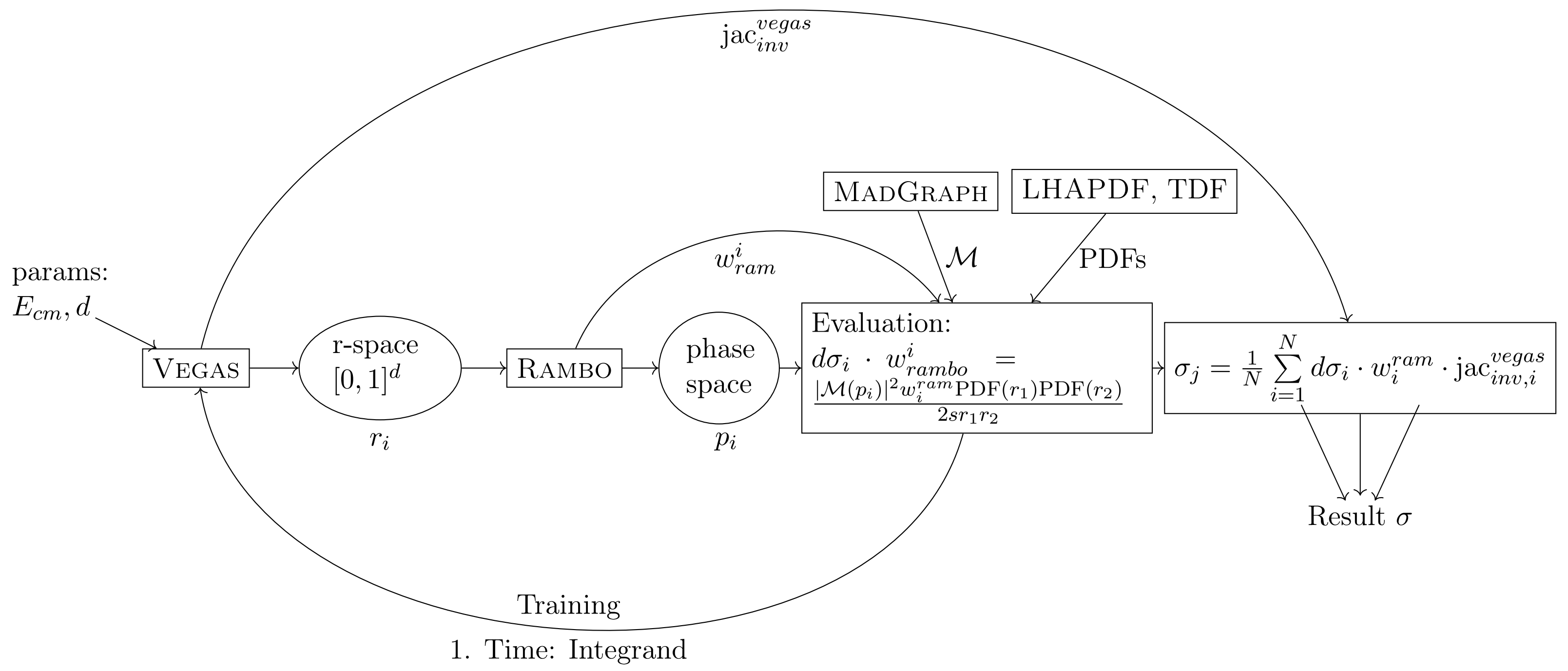

jac$^{vegas}_{inv}$
params: E_{cm}, d
VEGAS
r-space $[0,1]^d$
r_i
RAMBO
w^i_{ram}
phase space
p_i
MADGRAPH
LHAPDF, TDF
$\mathcal{M}$
PDFs
Evaluation: $d\sigma_i \cdot w^i_{rambo} = \frac{|\mathcal{M}(p_i)|^2 w_i^{ram} \mathrm{PDF}(r_1)\mathrm{PDF}(r_2)}{2sr_1r_2}$
$\sigma_j = \frac{1}{N}\sum_{i=1}^{N} d\sigma_i \cdot w_i^{ram} \cdot \mathrm{jac}^{vegas}_{inv,i}$
Result σ
Training
1. Time: Integrand

Figure 6: Program pipeline for VEGAS-Program

3.2 INN

The INN pipeline (Figure 7) is quite similar, as the network takes the place of Vegas.
During generative training, the INN samples batches of random points according to the Prior in the latent space and transforms them; after the output function, they are in the unit hypercube and Rambo will take them as input. The loss is evaluated and the hyperparameters are adjusted after each batch; if a divergence is used for the loss, it is the inverse of the inverse network jacobian (this is a detail: The inverse network jacobian is automatically calculated when a point is transformed from the latent into physical space, therefore it is used. However, the network jacobian could be taken as well (if the hyperparameters weren't adjusted in between), but that would require an extra computation) and the target function.
When the training direction is switched, the points from the phasespace associated with the value of the differential cross section are traced back to the latent space. As loss, a divergence of the network jacobian and the differential cross section is taken, and the hyperparamters are adjusted. The network tries to adjust itself in such a way, that, if the phasespace points are transformed back into the latentspace, they distribute like the Prior.
After every generation period, the Integral is evaluated with all function values that were sampled during that period. After the training has ended, the final evaluation of the integral is computed according to Equation 30

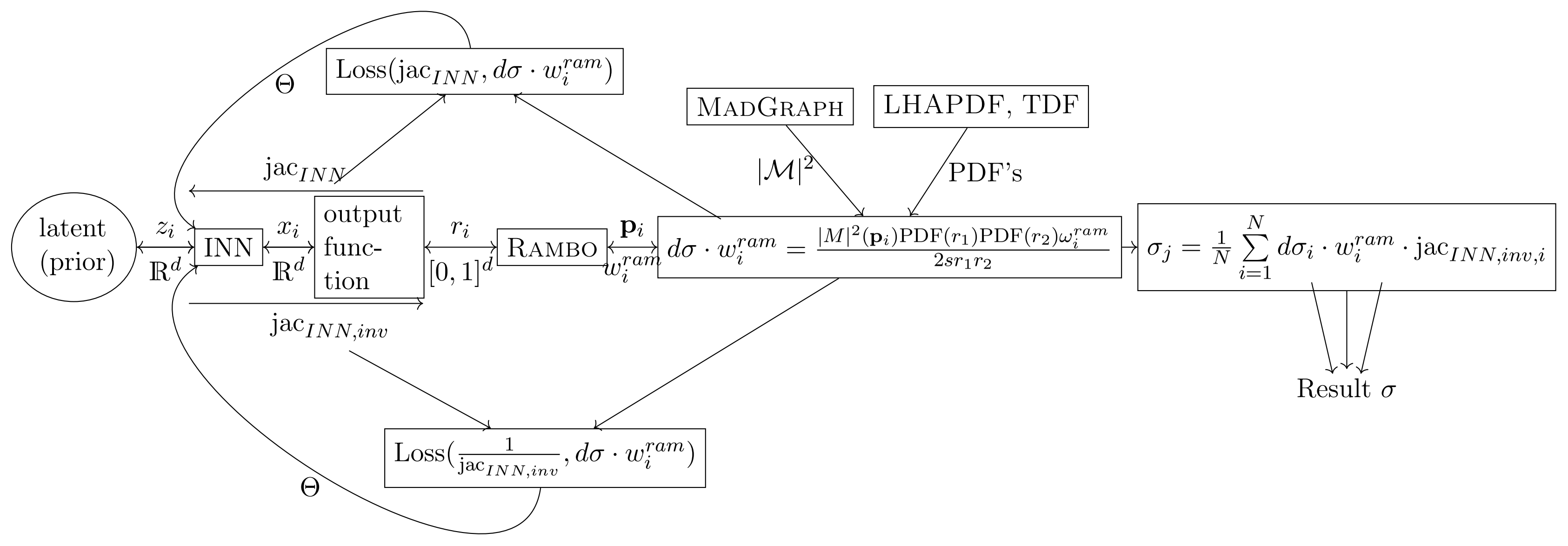

latent (prior)
z_i
$\mathbb{R}^d$
INN
x_i
$\mathbb{R}^d$
output function
r_i
$[0,1]^d$
RAMBO
$\mathbf{p}_i$
w_i^{ram}
$d\sigma \cdot w_i^{ram} = \frac{|M|^2(\mathbf{p}_i)\text{PDF}(r_1)\text{PDF}(r_2)\omega_i^{ram}}{2sr_1r_2}$
$\sigma_j = \frac{1}{N}\sum_{i=1}^{N} d\sigma_i \cdot w_i^{ram} \cdot \text{jac}_{INN,inv,i}$
Result σ
MADGRAPH
$|\mathcal{M}|^2$
LHAPDF, TDF
PDF's
$\text{Loss}(\text{jac}_{INN}, d\sigma \cdot w_i^{ram})$
jac_{INN}
Θ
$\text{jac}_{INN,inv}$
$\text{Loss}(\frac{1}{\text{jac}_{INN,inv}}, d\sigma \cdot w_i^{ram})$
Θ

Figure 7: Pipeline for INN-Program

4 VEGAS

To have a solid working ground to begin with, the VEGAS integration pipeline was set up. However, it turned out, it was more complicated than previously thought and some issues were to be tracked down. Therfore it was decided, to go to an easier toy function to have a better understanding what was going on.

4.1 Digression: Plots

In this thesis, often many lines of a model are shown in one figure. The lines most times correspond to the same samples associated with differnt weights obtained by a model, most importantly the inverse jacobian of the variable transformateion and the value of the integrand at the sampeled point. Plots labeled by "samples", "phasespace samples" or "pure samples" are not weighted at all and represent the raw output of the model. Labels like "inv vegas weight" or "inverse INN Jac" denote, that the samples obtained are weighted by the inverse of the variable transformation. The corresponding plot should be flat, as this reverses the effect of the transformation; this serves as sanity check. "All weights" and "integrand weights" show, that the samples produced are weighted by all available weights, namely the inverse transformation jacobian, the value of the integrand ant the point, and, in the physical examples, the phase space volume calculated by RAMBO. This can be seen as the true target distribution. In the optimal case, the plot without any weights is identical to the plot with all weights, as this indicates that the model has adapted perfectly to the target distribution. Heat maps are shown to qualitatively analyze the output distributions, as all details are shown directly and are not averaged out by a projection to a lower number of dimensions; the samples here are not associated with any weights. Additionally, the distribution of weights are plotted. This is the value of the inverse jacobian multiplied by the value of the integrand (and, if possible, the phase space volume). They are the values that the raw samples are weighted with for the "all weights" distributions. In the optimal case, it resemples a delta peak, because then, the jacobian of the model is identical to the integrand and therefore contains all information.

4.2 Toy functions

During this phase, several parts of the pipeline (Figure 6) were omitted: RAMBO, MADGRAPH and LHAPDF. To keep things simple and easy to understand, the toy function was chosen to exist in a one-dimensional hypercube and to be linear, $f(x) = x$, with a corresponding target probability density of $p(x) = 2x$. Plots show the target probability density (red), the raw number of samples (green), the number of samples weighted by the inverse transformation jacobian (blue) and the samples weighted with the function value as well (orange). If the grid is adjusted perfectly, it is expected that the green and the orange plot follow the target density in red, and that the blue histogram is uniformly distributed. Note, that for the histograms, their 'integrals' are normalized to one, such that they are easier to compare in one plot, as they can then also be interpreted as probability density.
After some difficulties with getting to know how this implementation works, the toy-function was modeled correctly (Figure 8).

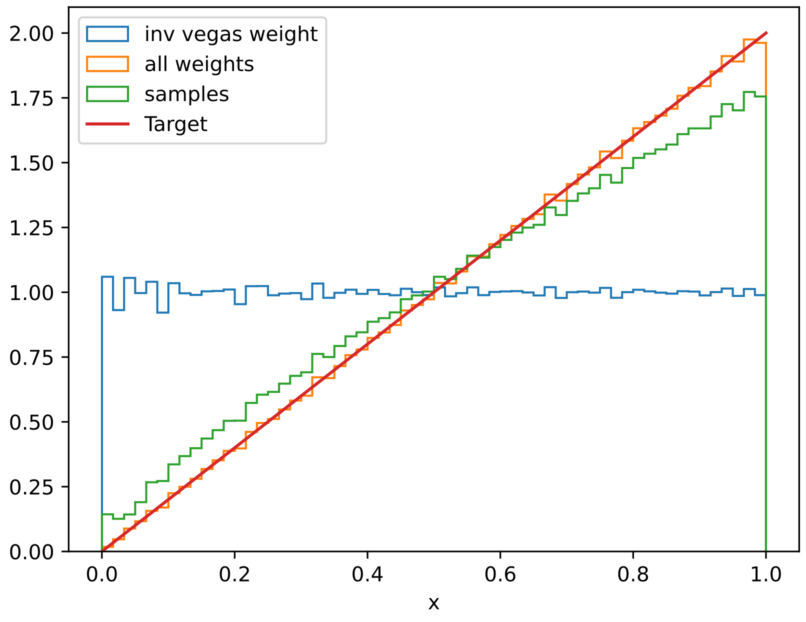


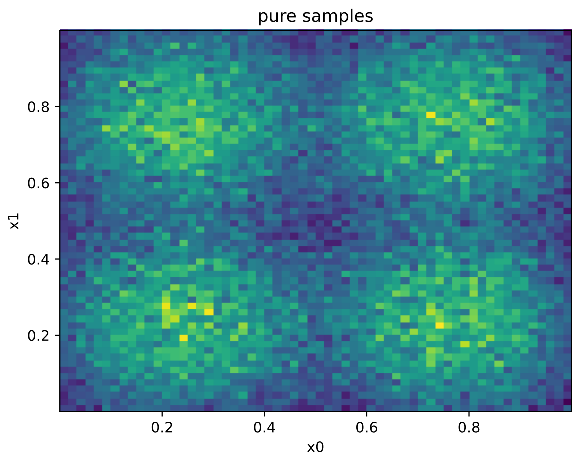
Figure 8: VEGAS adapted to a linear toy function

Later in this thesis, the INN will be mostly trained on the double gaussian function of equation 5. To be able to compare it to VEGAS directly and to demonstrate its seperability assumption, it is showcased in Figure 9. While in the 1D-projection the unweighted and weighted distributions are reasonably close, clearly, in the heatmap without any weights, four peaks are visible, resulting in a weight distribution ranging more than three orders of magnitude.

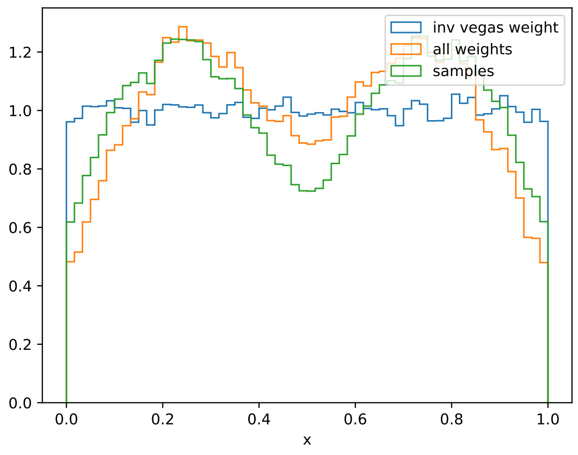
4.3 Cross sections

With a correctly working VEGAS, the physical function is again implemented. As with any physical prediction, an error is associated to the value. Several histograms are plotted: In green the distribution obtained by MADGRAPH, which is the reference to control the output and considered to be true; blue are the raw phase space samples produced, without the consideration of any weights. The closer it is to the true (green) distribution, the more efficient will the unweighting be and the faster the variance of the integral converges. In red, the phase space samples are weighted with all weights, (inverse transformation jacobian, phase space weights/cuts and function value) are shown as sanity check. It should be within the errorbars to the true distribution, ensuring that the system produces correct results. For the output of VEGAS and MADGRAPH, the fraction between then and the relative deviation from 1 is shown as well. For each graph without any weights, $\frac{1}{\sqrt{N}}$ errors for any bin are used. For weighted events, the error for each bin calculated according to:

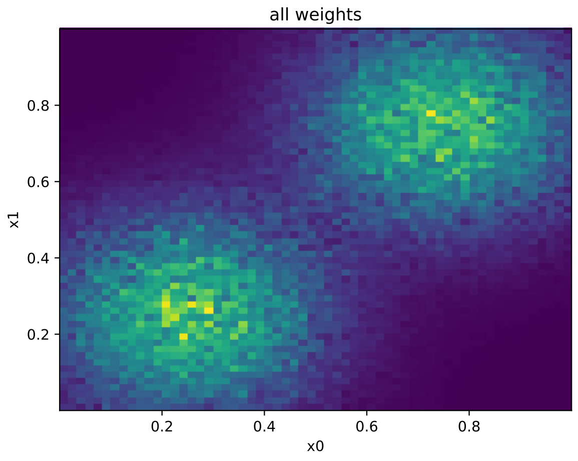
$$\frac{N}{\sqrt{\frac{\sum \text{weights}}{\max(\text{weights})}}} \tag{31}$$



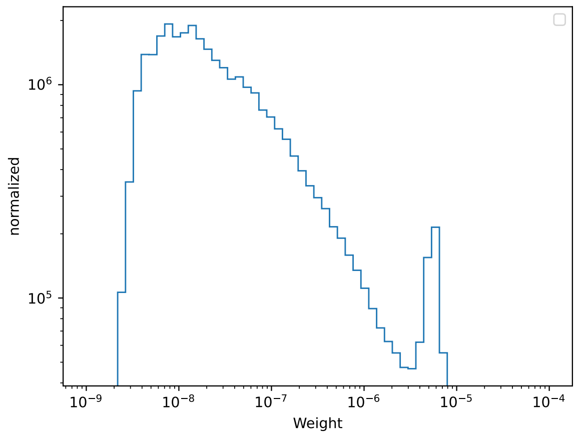
(a) Heatmap of samples without any weights



(b) 1D Projection



(c) Heatmap of samples with all weights



(d) Weight distribution

Figure 9: VEGAS trained on the double gaussian function

Observable Name	Number of occurances
$2r_1r_2\sqrt{s}$	1
p_z	3
p_T	3
η	3
ϕ	3
ΔR	3
r (Momenta fraction)	1

Table 1: plotted Observables

Parameter Name	Value
$\sqrt{s}$	13 TeV
p_T-mincut	20GeV
ΔR-mincut	0.4 GeV
η-maxcut	None
PDF-set	MSTW2009lo68cl_nf3

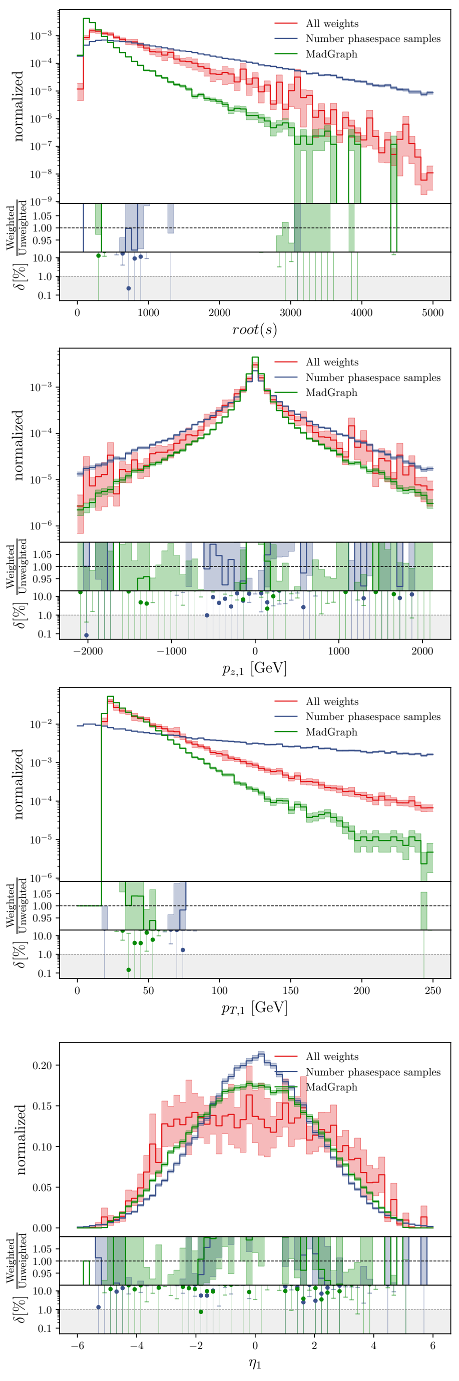
Table 2: parameters for the Process

With this calculation, if one bin is dominated by a single high-weighted event, the error goes to N, if all weights are equal, it reduces to the case without weights.
In general, a total of 17 observables is plotted (Table 1); some observables appear multiple time for each particle or particle pair respectively. For these, only one case is shown, as they distribute very similary ($d\sigma$ is invariant under final state particle exchange of the same kind: Here, exchange of bosons will only give a factor of one, but even for fermions with a factor of minus one in the matrix element, the squared absolute Value is still invariant). Each time, the distribution of the center-of-mass energy (root(s)), the momentum in beam- and transversal direction (p_z, p_T), the pseudorapidity (η), azimuthal angle (ϕ), angular separation (ΔR) and the momenta fractions are plotted. The paramters for VEGAS are the same as in the toy example; phase space cuts and th PDF set are listed in Table 2.
Even with the fixes to VEGAS, the result of the observable distributions are not in accordance with MADGRAPH (Figure 10a).
To enclose the error, individual parts of the pipeline are deactivated, by setting their corresponding values in the calculation to one. To still be able to compare to MADGRAPH, the corresponding factors in its code are also set to one.
As sanity check for RAMBO, the PDFs and the matrix element are disabled, leaving only

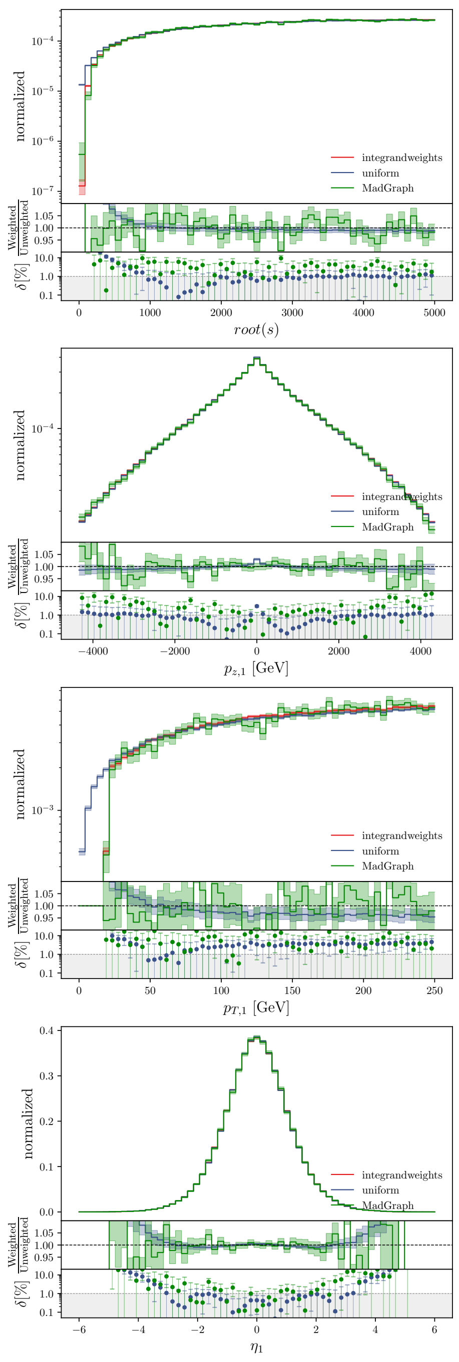
$$\frac{w_{\text{RAMBO}}}{2r_1r_2\sqrt{s}^2} \tag{32}$$

for the integrand. The distributions in Figure 10b are in perfect accordance. Note that the samples were taken uniformly in the phase space, as VEGAS was already ensured to work correctly.

Next, the Matrix element $\mathcal{M}$ is activated again. The distributions of VEGAS' samples with



(a) Cross section with VEGAS fixes



(b) Cross section without Matrix element and PDFs

Figure 10: Progress of correctly setting up the cross section for VEGAS

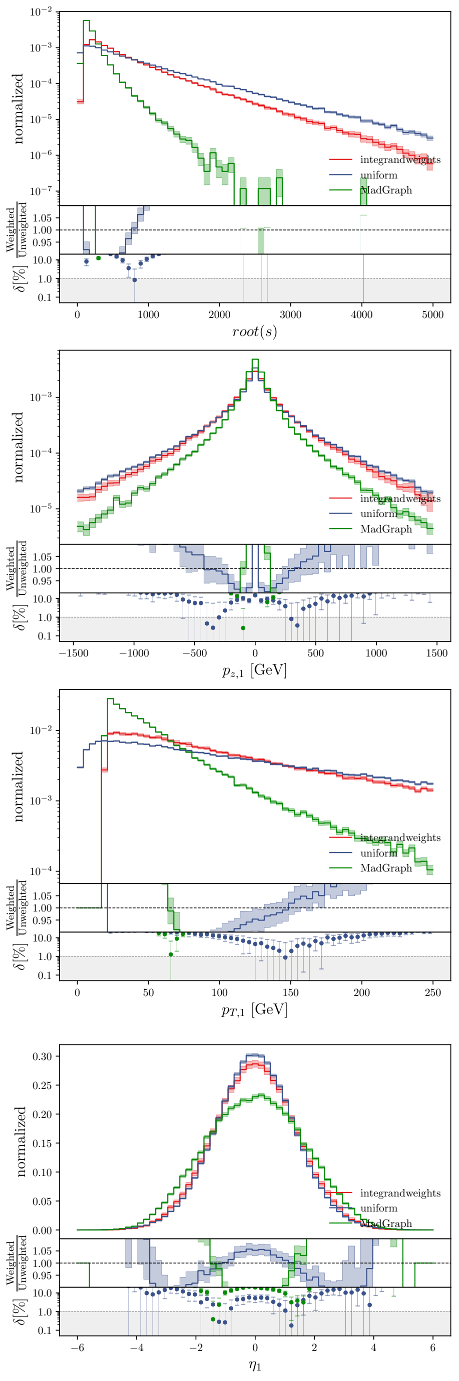
all associated weights is within the errorbars of MadGraph. The raw number of phase space samples only roughly represents the true distribution, but this can be expected in such complex high-dimensional functions. So, with Figure 11b can be shown that madGraph works correctly.

Having checked the Matrix element, it is deactivated again, and the parton distribution functions provided by LHAPDF are next on the list. The source of evil is now found, as the all-weighted and the true distribution in Figure 11a differ a lot. Trying do identify the source of error, it was found, that in this project a slightly different parton distribution function was used than in MadGraph, but changing it to the identical didnt yield much difference. Also hardcoding the Energy scale of the PDFs to the mass of the Z-Boson didnt produce the desired results. After consulting the supervisor of this project, it was decided to use a quite simple function for the parton distribution function ('Tilman gluon distribution function' TDF):

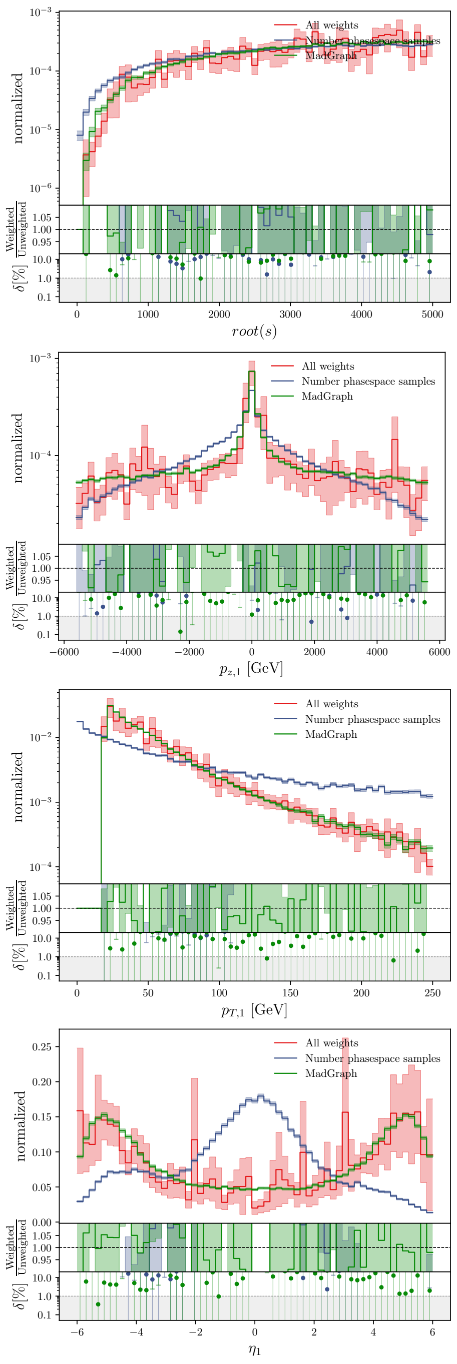
$$\mathrm{TDF}(x) = \frac{1}{x^2} \tag{33}$$

With this change, the final results are sufficiently satisfactory (Figure 12). The weight distribution is still quite broad, spanning almost 12 orders of magnitude. The predicted Values of the cross sections are shown in Equation 34. While they differ by ≈ 15.7 standard deviations, they are at least in the same order of magnitude.

$$\sigma_{gg\to ggg}^{\textsc{MadGraph}} = 15.296(15)\mathrm{GeV}^{-2} \qquad \sigma_{gg\to ggg}^{\textsc{Vegas}} = 12.78(16)\mathrm{GeV}^{-2} \tag{34}$$



(a) Cross section without Matrix Element



(b) Uniform phase space sampling without LHAPDF

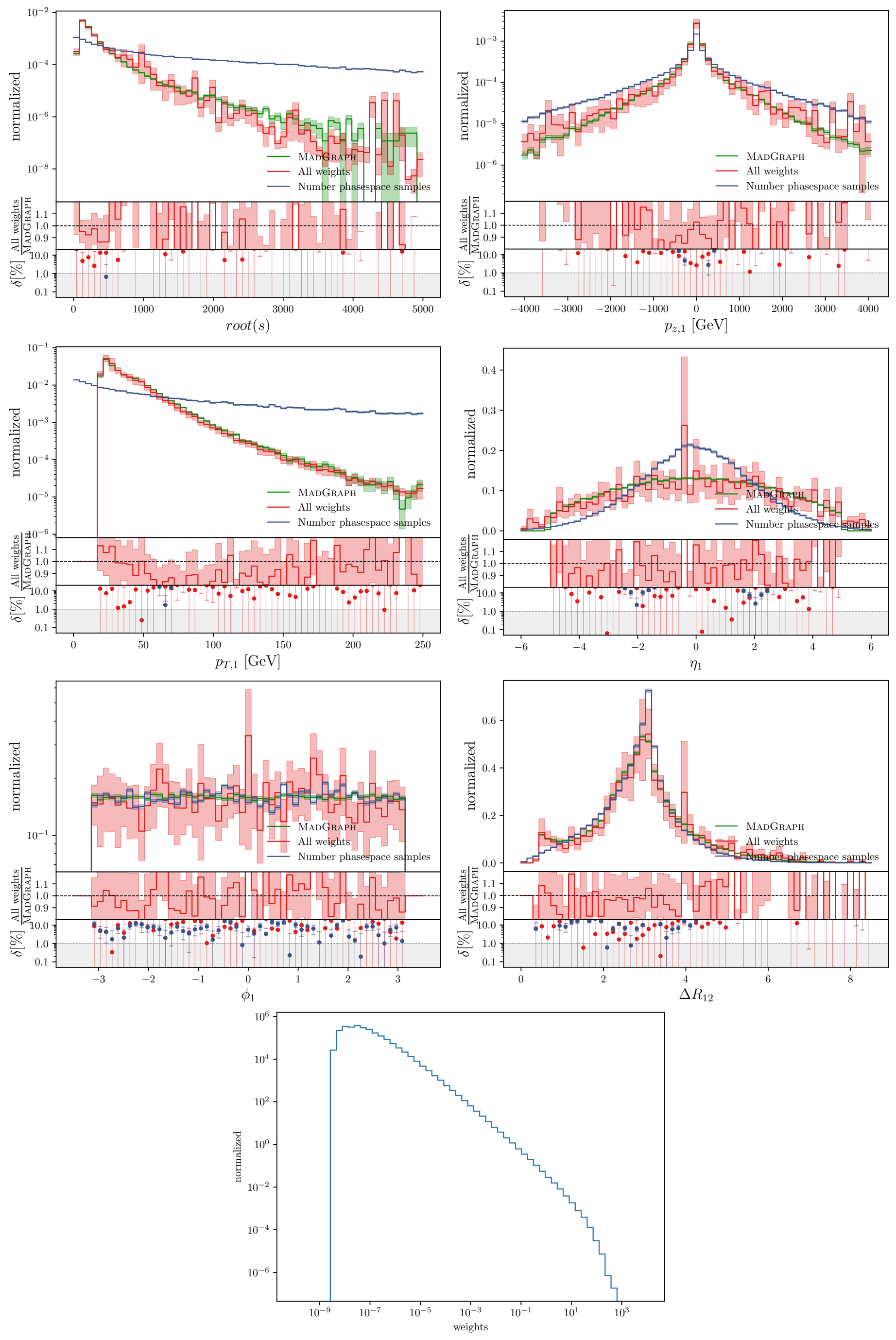


Figure 12: Vegas adapted to the cross section with TDFs

5 INN

Having set up the VEGAS Baseline, it was time to actually get to work with the INNs. As described earlier, training can take place in two directions, with the recycling training being the proposed innovation of this project. Therefore, the `i-flow` package [3] can be used as benchmark. It utilizes only the generative training and an attempt was made to copy this part here.
To first get a feeling for both training directions, they were examined on their own respectively. To have more control of what was going on and having learned from the difficulties with the VEGAS pipeline, the INN was trained on a toy function. It again lives in an unit hypercube and was chosen to be two dimensional, to allow for some complexity, yet retaining the possibility of viewing the details directly by using heat maps.

5.1 Toyfunctions

Two different toy functions were used. One of them is the doublegauss first described in Equation 5, illustrated with a heatmap and a projection to 1D in Figure 13.

Another function used is the gaussian ring (Figure 14):

$$f_{\text{ring}}(x_1, x_2) = \exp\left(\left(-\frac{\sqrt{(x_1-0.5)^2+(x_2-0.5)^2}-0.25}{\sqrt{0.03}}\right)^2\right) + 0.01 \tag{35}$$

5.2 Learning rate Schedulers

The learning rate descibes, how much the hyperparameters are adapted within each step of the optimizer. It can change during the training to help the network to quickly find a strong minimum in the loss landscape and settle on a low Value. In this project, two different approaches were tried **One Cycle Learning rate** and **Step Learning rate**

One Cycle Learning rate

With this Scheduler, the learning rate starts on a low value, increases in the first third of the training to a maximum and smoothly decreases down again. It aims to stabilize the network with a low beginning learning rate, then speeds up the traing to minimize the training time needed. The low ending learning rate helps to settle in a minimum [17]. The progress is illustrated in Figure 15. The beginning and maximum learning rate can be adjusted.

Step Learning rate

The step learning rate can be either hardcoded or automatically adjusted. The key how it works is, that it multiplies the learning rate with some factor once a condition is fulfilled, either after a set number of epochs, of if learning stagnates. Often, the automatic version is also called "reduce learning rate on plateau". So, the learning rate follows an exponential

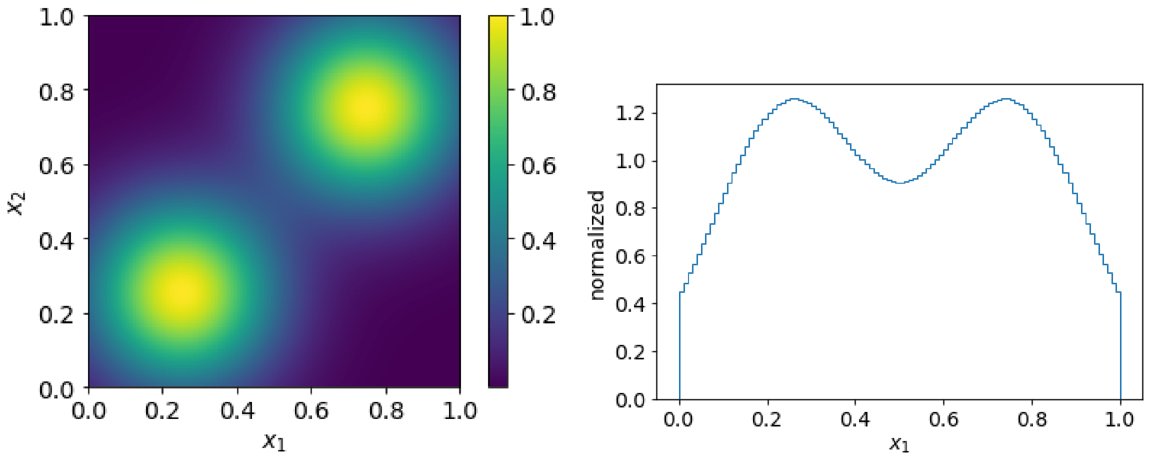


Figure 13: Double gaussian toy-function

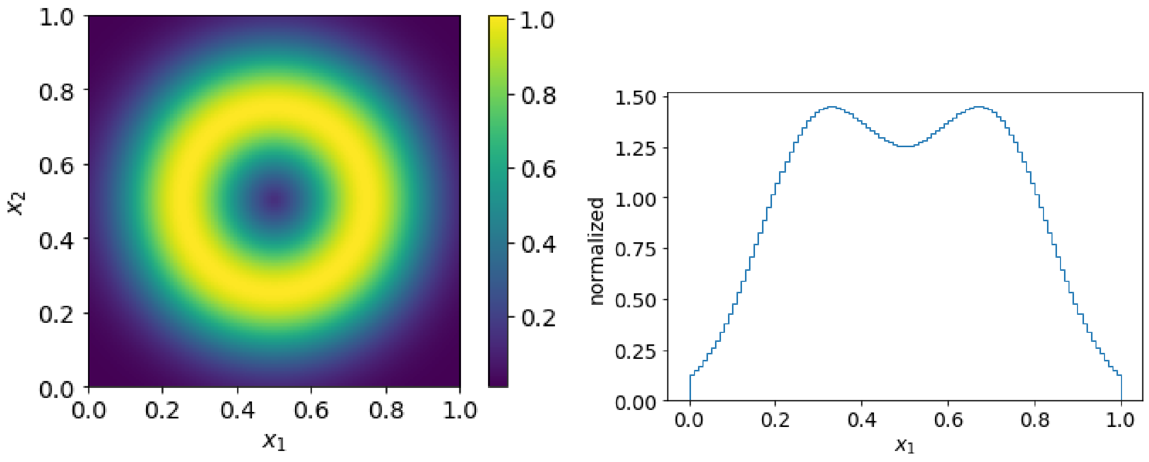


Figure 14: Gaussian ring toy-function

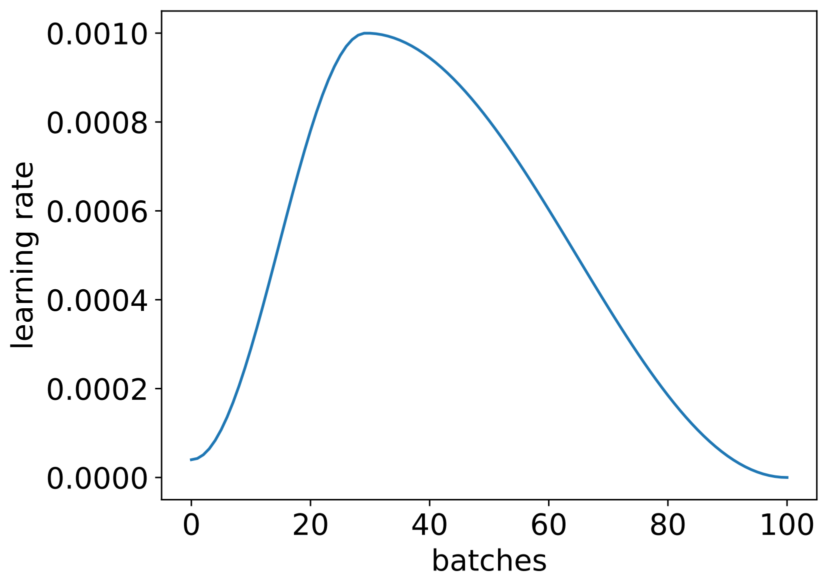


Figure 15: Progress of the One Cycle learning rate

(decay). If hardcoded, the beginning learning rate, the number of epochs after which it is reduced, and the factor with which it is multiplied can be set. In the automatic version, typical parameters are `patience` (The number of epochs the scheduler waits on the plateau before it reduces the learning rate), `threshold` (Value for measuring the new optimum to only focus on significant changes) or `cooldown`(Number of epochs the learning rate is frozen after being reduced) [18].

5.3 Generative Training

First, the network is trained only by generating events. The size of the network was chosen to be rather big at first (Table 3). This made training longer but easier to control. The network is evaluated regulary during training to visualize the dynamics of the adaption to the target function. Both toy functions and learning rate schedulers are shown. Furthermore, the value

Parameter	Value	Comment
batch size	5000	number of samples per evaluation of the loss
batches	1000	longer training than needed for illustration
number of coupling block	15	
internal neural network size	256	see Figure 4
layers per block	3	see Figure 4
number of bins	60	see Figure 5
gradient clip	0.1	
latent space	$\mathbb{R}^2$	effectively cut at $\pm$ 10
prior	gauss	

Table 3: Network parameters for the 2-dimensional toy function

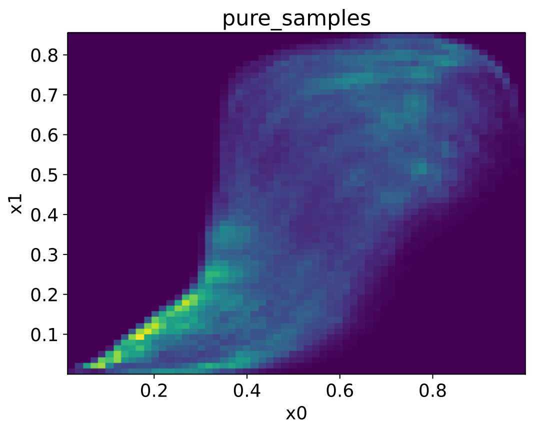
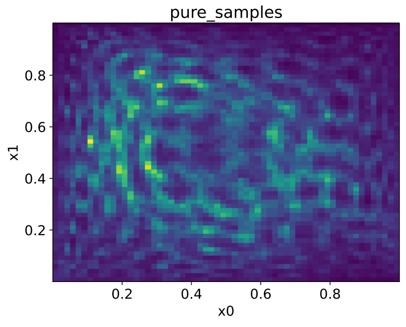
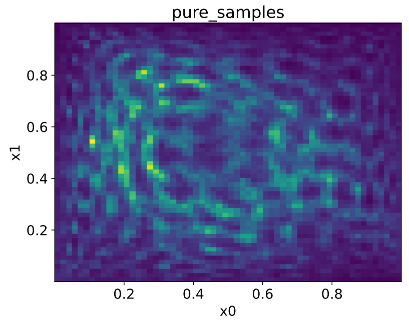


Figure 16: Negative example: Network falls apart with too high generating learning rate

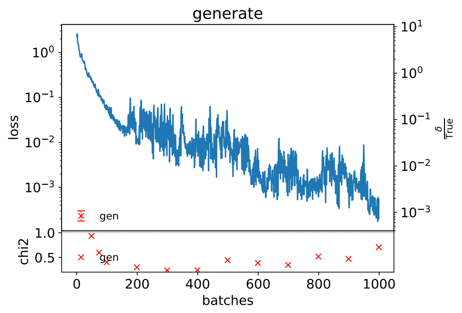
of the loss, together with the absolute deviation of the integral evaluation to the true value are plotted on a log-scale (If the integral is very close to the true value, it doesnt show up anymore because of a mincut in the Integraldeviation); an evaluation if the reduced χ^2 sum is drawn as well as the distribution of weights.
In Figure 17, the network adapts in the beginning very fast to the target function. After circa 170 batches, the loss starts to become noisy, but in general still decreases. At this time, the network has capured most of the structure, and only needs to model minor details in itself. This took some time, but this could be because of the automatic reduce-on-plateau scheduler, that already lowered the learning rate from $1 \cdot 10^{-4}$ in the beginning to $4 \cdot 10^{-5}$ after $\approx$500 and lastly $1.6 \cdot 10^{-5}$ for the last 200 batches. The Weight distributions are shown at each evaluation of the network; a nearly untrained network spans about 3 orders of magnitude, decreasing to less than one order of magnitude in the end. The projection to one dimension in the end shows, that the network (blue, "Pure samples") adapts the target function (red, "All Weights") really good. The samples weighted with the inverse network jacobian produce a flat distribution, as expected. Only at the integration edges, there is some noise, but there, the function value is generally lower, resulting in less samples produced in that region, and lead to larger statistical errors, so this can be expected.
The doublegaussian toy function was combined with the One Cycle learning rate in Figure 18. The beginning learning rate was set to $1 \cdot 10^{-5}$, with a maximum of $2 \cdot 10^{-4}$. Although the loss is smoother for a longer amount of time, it takes long for the network to even capture the rough for of the target, and also quite long to smoothen out the details. This is not the problem of the scheduler, it is also true for combining this function with other schedulers. In the end, the function is yet still well approximated.
Generally, it was found out by combining different learning rate schedulers with the toy functions, and experimanting with the learning rates, that the double gaussian toy function seems to be harder for the network to adapt to. With the One Cycle learning rate, higher maximum learning rates could be achieved: If the learning rate is too high, the network makes a wrong turn in the loss landscape and begins to not sample some regions in the physical space and doesn't adapt the target where it samples at all (e.g. Figure 16 : One Cycle learning rate with a maximum of $5 \cdot 10^{-4}$).



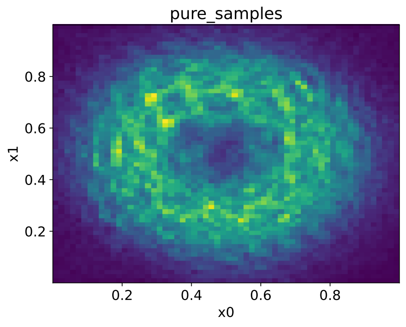
(a) 25 Batches



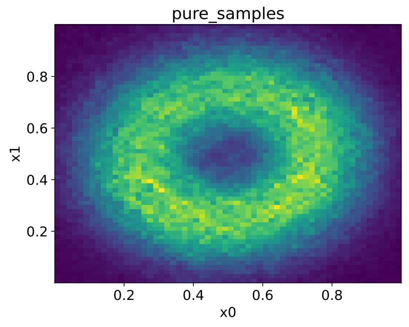
(b) 50 Batches



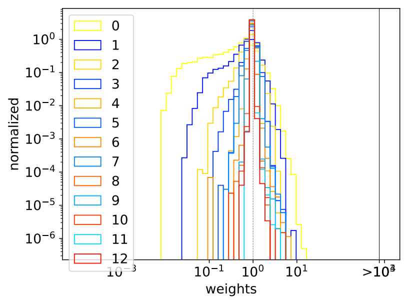
(c) Generating Loss



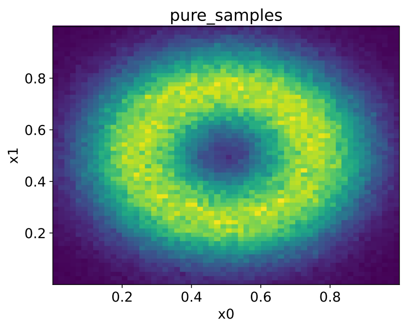
(d) 100 Batches



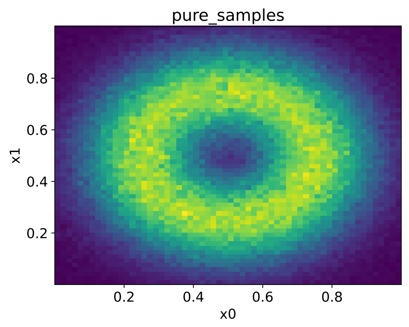
(e) 200 Batches



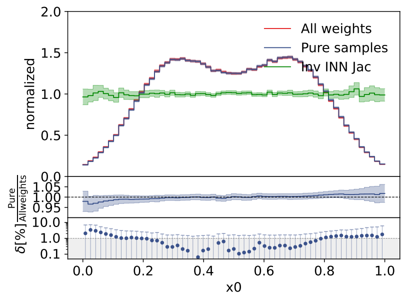
(f) Progress of the weight distributions narrowing down



(g) 600 Batches

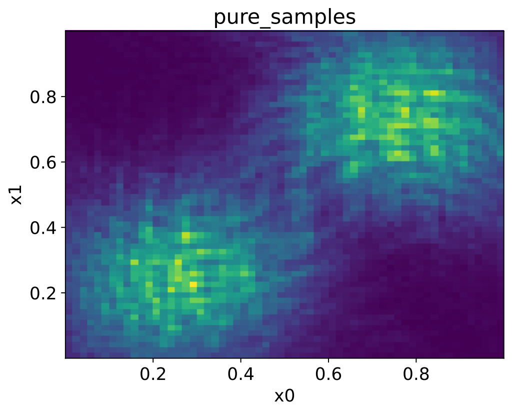


(h) 1000 Batches

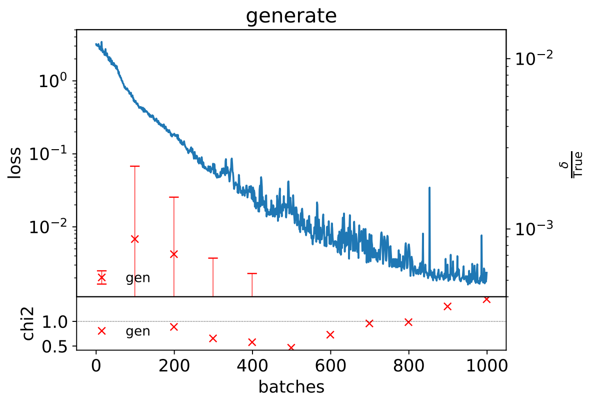


(i) 1D-Projection

Figure 17: Gaussian ring adapted with reduce on plateau scheduler

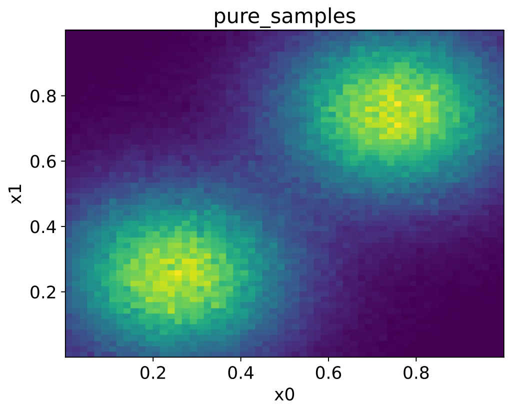


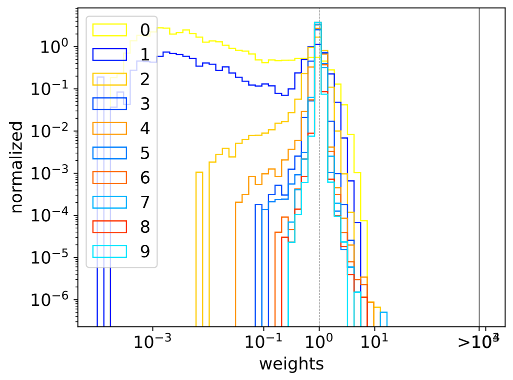
(a) 300 Batches



(b) Generating Loss

(c) 1000 Batches





(d) Weight Distributions

Figure 18: Double gauss adapted with the One Cycle lr scheduler

5.4 Recycling Training

The recycling training needs data to even begin with. To this end, some data is generated in the beginning with an untrained Network (essentially flat in the integration space), and the loss isn't evaluated during this phase; the optimizer doesn't change any hyperparameters. Then, the network can iterate over those data points. One iteration over all provided Data points is called an epoch.
For Figure 19, 125000 function evaluations were provided and the network iterated over them using the step scheduler with a reducing factor of 1, resulting in a constant learning rate of $2 \cdot 10^{-4}$. While the adaption is okay and in the beginning quite fast, it quickly reaches a plateau in the loss landscape. The structure is less pronounced than with generative training and not as accurate to the target distribution; this is not a problem of a too high learning rate, the One Cycle scheduler produces similar results. To the loss, 0.1 is added so it remains positive and plottable on a log-scale. The weight distributions narrow down similary to generative training.

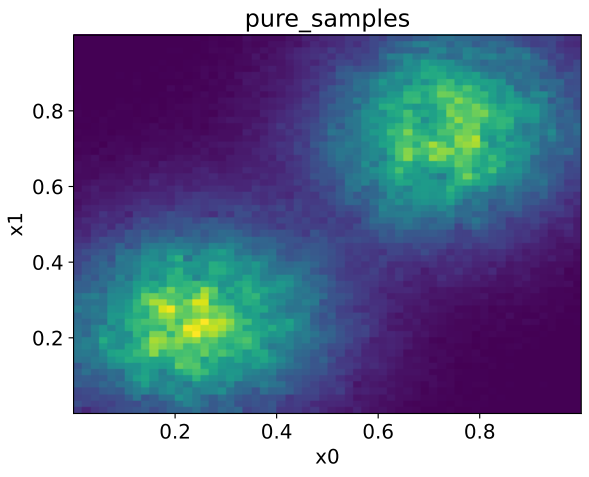
One idea is to provide more samples for the network to learn from, Figure 20. Here, the network was trainined with 1.25 Million samples and for 50 epochs. Indeed, this improves the distribution; however the amount of more samples needed is quite big compared to the improvement in the result, and also scales the time needed to recycle for an epoch accordingly.

If the learning rate is set to a relatively high Value ($2 \cdot 10^{-3}$ in Figure 21), the result still looks okay. However, this can lead to peaks in the loss early in the training, indicating the optimizer has changed the hyperparameters too much, so that the minimum is missed and overshot. Additionally, a relatively high learning rate in the end makes it very hard to settle low in a minimum. If the value is set too high, it behaves similar to generative training, but it immediately only samples a very small region in the center of the integration space (Figure 22), and does not manage to recover. Generally, the learning rate during recycling can be set a bit bigger than during generating.
In the end, an example of overfitting is shown in Figure 23. This happens, if very few function evaluations are provided (here 1500) and the network recycles them for a very long time (160 epochs). Instead of adapting to the function, it begins to fit the individual points and memorizes them.

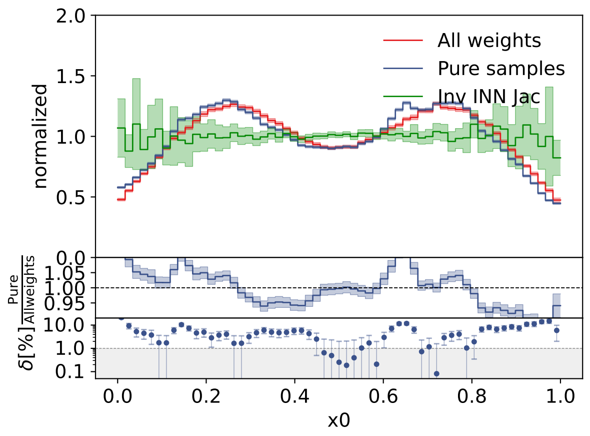
5.5 Combined Training

When combining the training directions, the author of this thesis proposes the name BIOFLOW (BIjective nOrmal distributed flow), although no official name was chosen within the research group.
The training schedule, i.e. when the network is trainied by generating and when it recycles the data produced, is experimented with, as it turns out to be quite influential. The generating loss is evaluated during the recycling epochs and visible when the curve is piecewise flat (one flat region represents one evalutaion of the loss for better visibility). For better comparability (although not perfect), it was chosen to generate 250 batches and recycle 4 epochs.

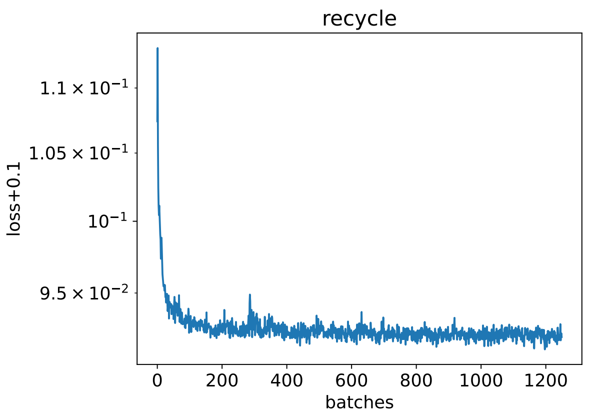
First, for Figure 24, the training directrions were interchanged repeatedly with 50 generating batches and 1 recycling epochs. The generating loss is evaluated after every recycling epoch.



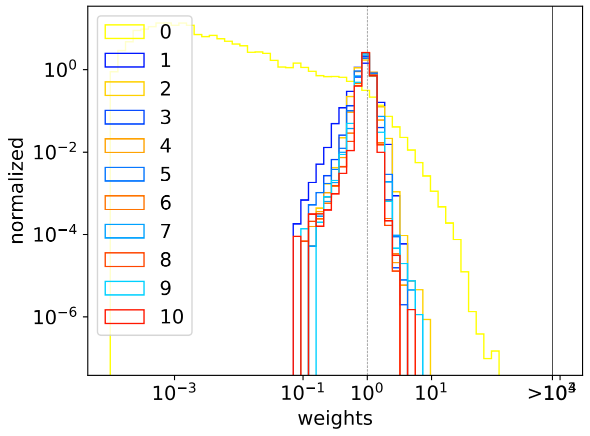
(a) Pure sample Distribution



(b) 1D Projection



(c) Recycling Loss



(d) Weight Distributions

Figure 19: Double gauss trained by 50 epochs of recycling

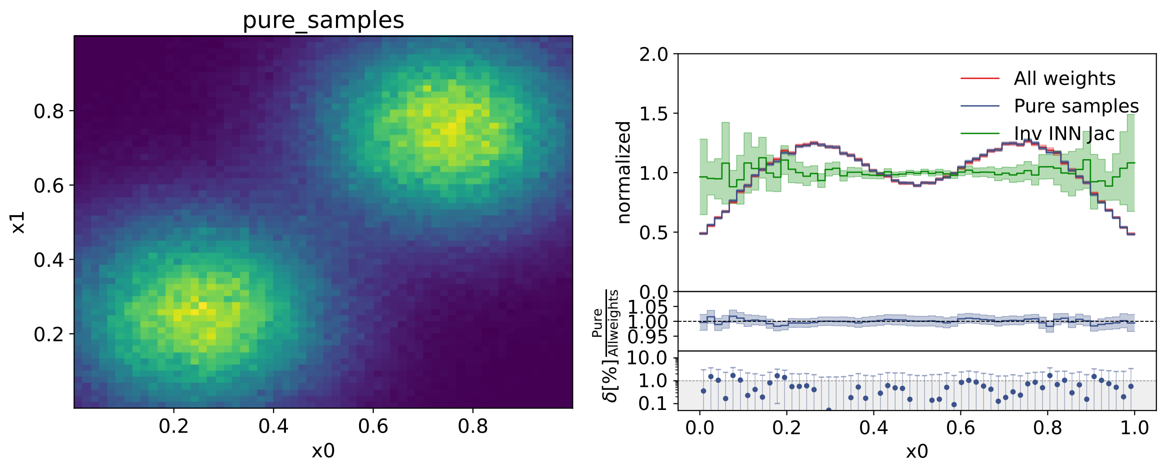


Figure 20: Recycling training with 1.25 Million Data points

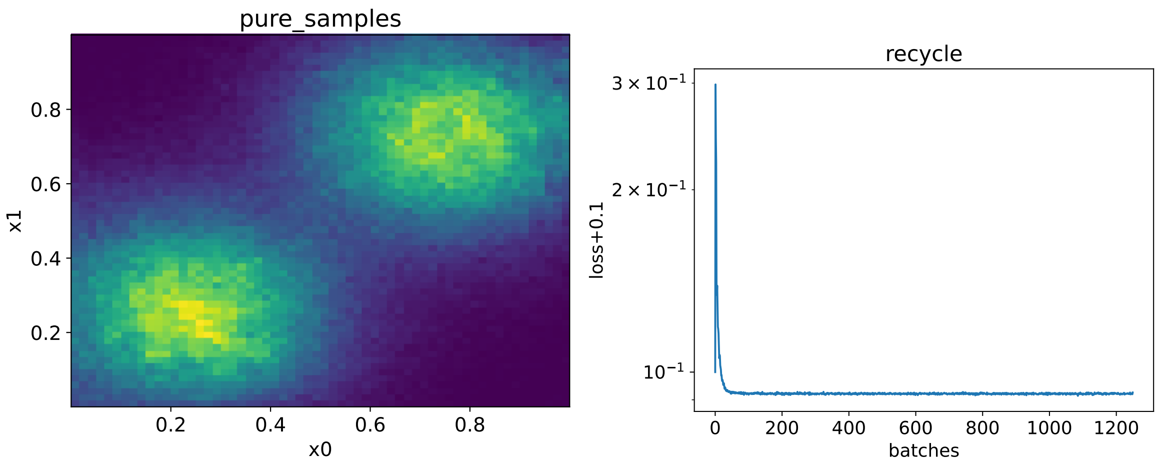


Figure 21: Relatively high recycling learning rate for 30 epochs

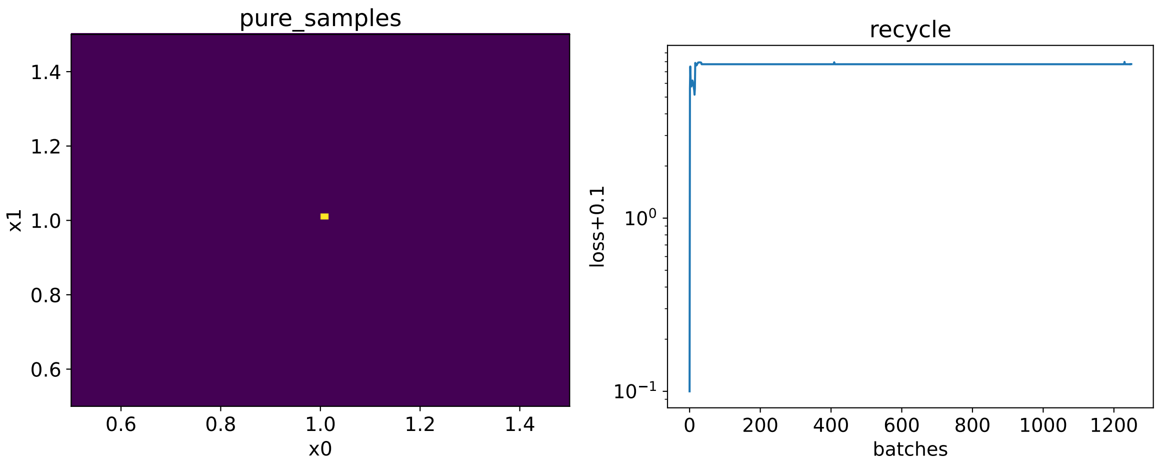


Figure 22: Way too high learning rate during recycling $(2 \cdot 10^{-4})$ after 5 epochs

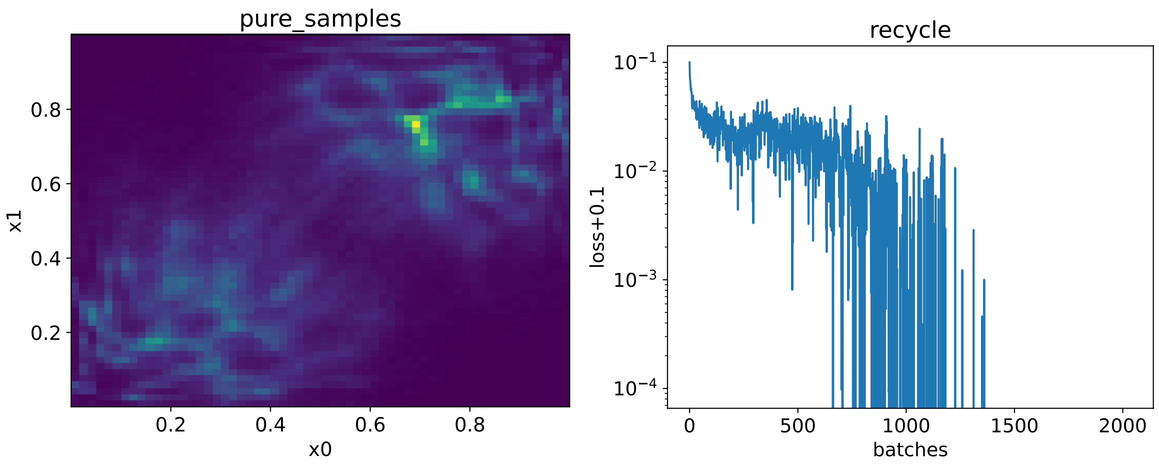
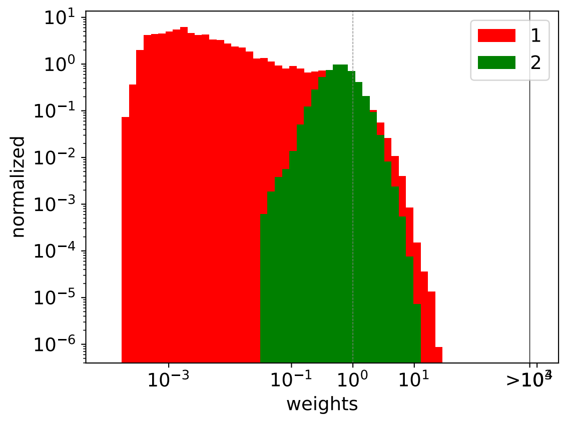
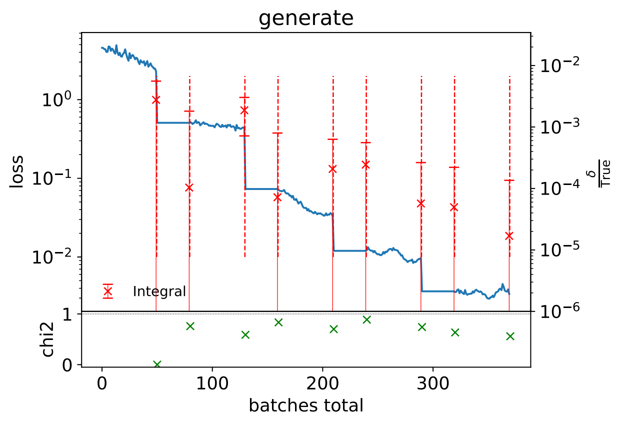


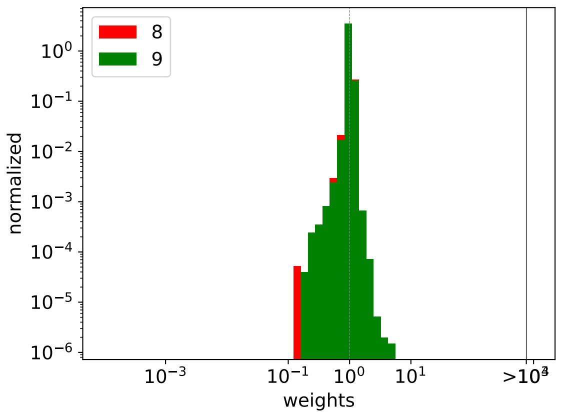
Figure 23: Overfitting



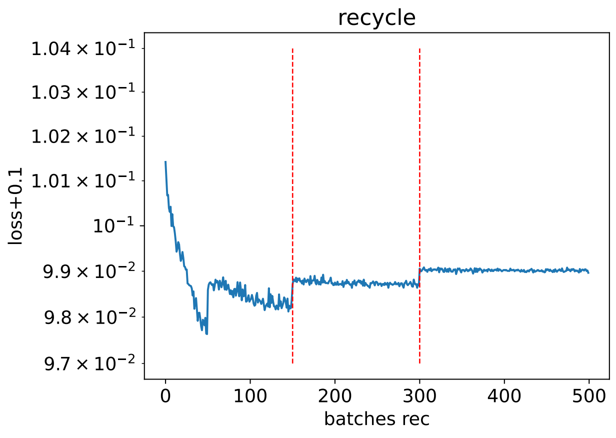
(a) After recycling



(b) Generating loss also evaluated during recycling



(c) After generating

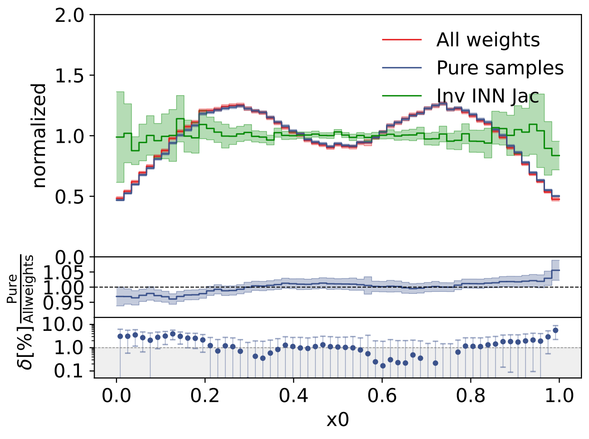


(d) Recycling loss

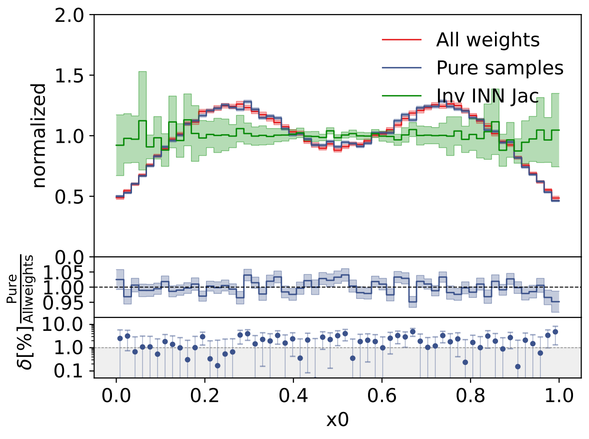
Figure 24: Repeatedly interchanged Training directions

Two things can be figured out: First, the weight distribution get primarily narrowed down by the recycling training, while generating points might even broaden it again (Figure 24a: Weights before (red) and after (green): First recycling step: great improvement; Figure 24c: After before/after last generation: slight detoriation) Second, the value of the recycling loss does not neccessarily correlate with the quality of the adaption, as it increases with training, though the recycling loss increases after each training direction change; furthermore, while the recycling loss is constant during training in the end, and the net work doesn't seam to learn anymore, the generating loss still decreases after each recycling period. The sample distribution is very close to the target (Figure 25a)

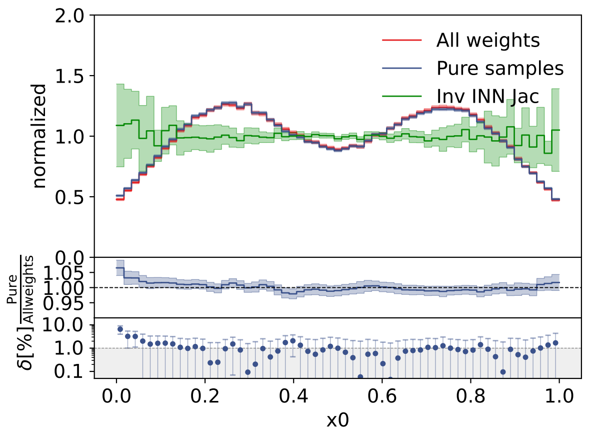
Another option is to first only generate a few batches, recycle over them, and then generate the rest. This method is quite similar to pure recycling training: During the first generating phase, the net hardly learns anything due to the short duration. In the next period, the network gets quiet well and fast trained given a good number of function evaluations. The final generating phase then resembles training on and already reasonably trained network. Note that this is the fastest setup in this section, but it also produces the least well output



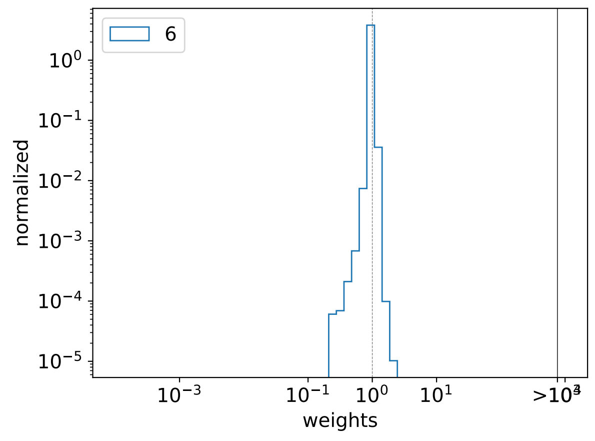
(a) Interchanged Training



(b) Recycle in the beginning



(c) Recycle in the end



(d) Weight distribution for recycling in the end

Figure 25: Output distributions for different training schedules

(Figure 25b).

Then, first all batches were generated and then recycled. Note that this is more training than the runs before, since during recycling, it sees all batches during recycling, instead of first only 50, then 100 etc. This is the longest training method in this chapter, but also produces the best results (Figure 25c); especially the final weight distribution is very narrow (Figure 25d).

In Summary, interchanged training with ending on recycling training is a good midway between moderate training time and good output distributions, though each setup has its advantages and disadvantages.

Parameter	Value
learning rate	$1 \cdot 10^{-3}$
batch size	5000
number of coupling blocks	2
internal neural network size	32
layers per block	4
number of bins	16
gradient clip	10
latent space	$[0, 1]^d$
prior	uniform

Table 4: `i-flow` Architecture

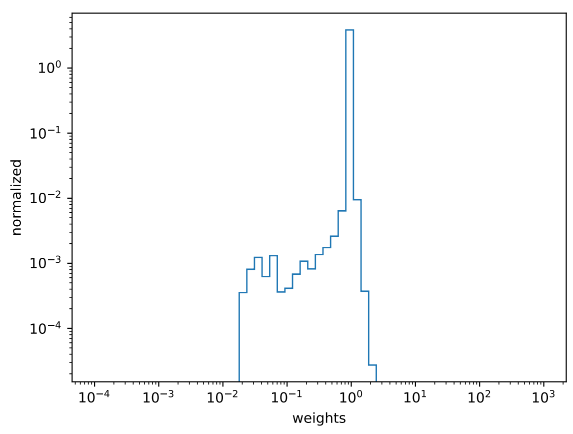
5.6 Loss

With the now working combined training, many different Loss functions were tested on the double gaussian function. Most of them were divergences inspired by `i_flow` [3] [19]. About half of them produced satisfactory results similar to the exponential loss, but not noticeably better, so it was kept for the future investigation, also to keep compareability. The others weren't as true to the target distribution and seemed to require some more work to function nicely, so they were omitted. Furthermore, as they are the measures of success for this work, directly the variance of the integral was used for the loss, and some implementations of the weight distribution. The variance-loss worked okay, but suprsingly, didn't reduce the variance more than the exponential loss, but narrowed down the weight distribution a little bit more, which seems counterintuitive. Unfortunately, no way with meaningful output to use the weight distribution as loss could be found. The idea was to punish weights assymetrically, with an exponential increasing part at high values and a not so extreme scaling function on low values (linear and one-over-square were tried).

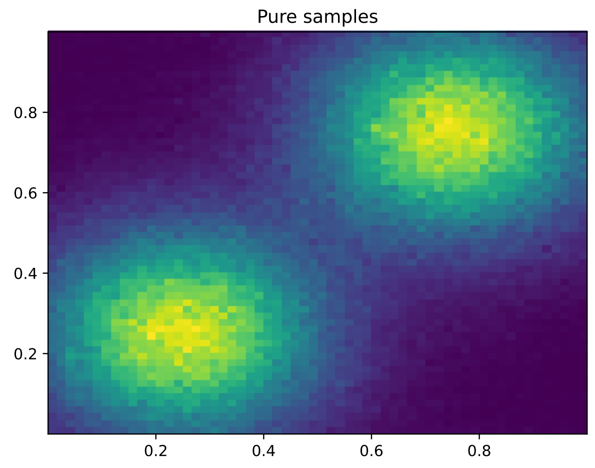
5.7 Benchmark

Now, that everything was working properly, it was decided to test `i-flow` on the double gaussian toy function to get a benchmark for the generation training in Bioflow, that should be roughly achieved and then beaten with the combination of generative and recycling training. This was done before implementing the cross section again, because the ultimate goal of this project is no proof-of-concept, but to actually improve an existing program and do meaningful work.
It was soon found, that this would be no easy task. The network used by `i-flow` was significantly smaller than the networks built until now, allowing for a high learning rate (see Table 4), yet produced impressive accurate output (Figure 26: 1000 Batches with a constant learning rate of $1 \cdot 10^{-3}$). Note the very little tail of the weight distribution at high values in Figure 26a indicating high efficiency for unweighting. The evaluation of the integral is accurate to 0.0042 % of the true value (≈ 0.333349), with a deviation of 0.53 σ.

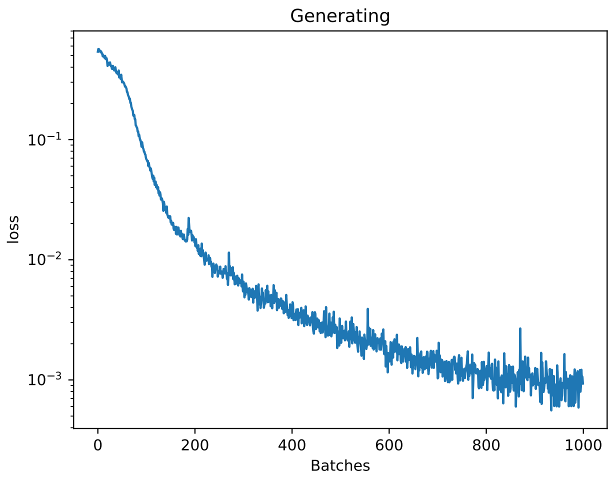
But the learning rate could be set even an order of magnitude higher, requiring about one tenth of the batches used before to obtain a similar result of the final distribution (Figure 27).



(a) `i-flow` weights

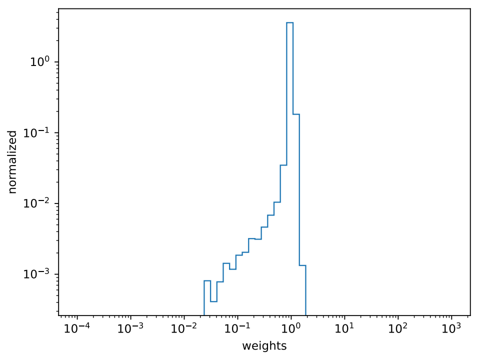


(b) `i-flow` samples

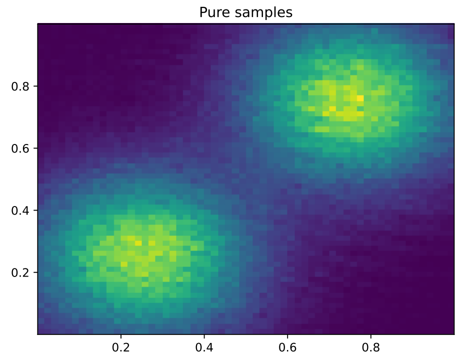


(c) `i-flow` loss

Figure 26: `i-flow` out of the box



(a) `i-flow` weights



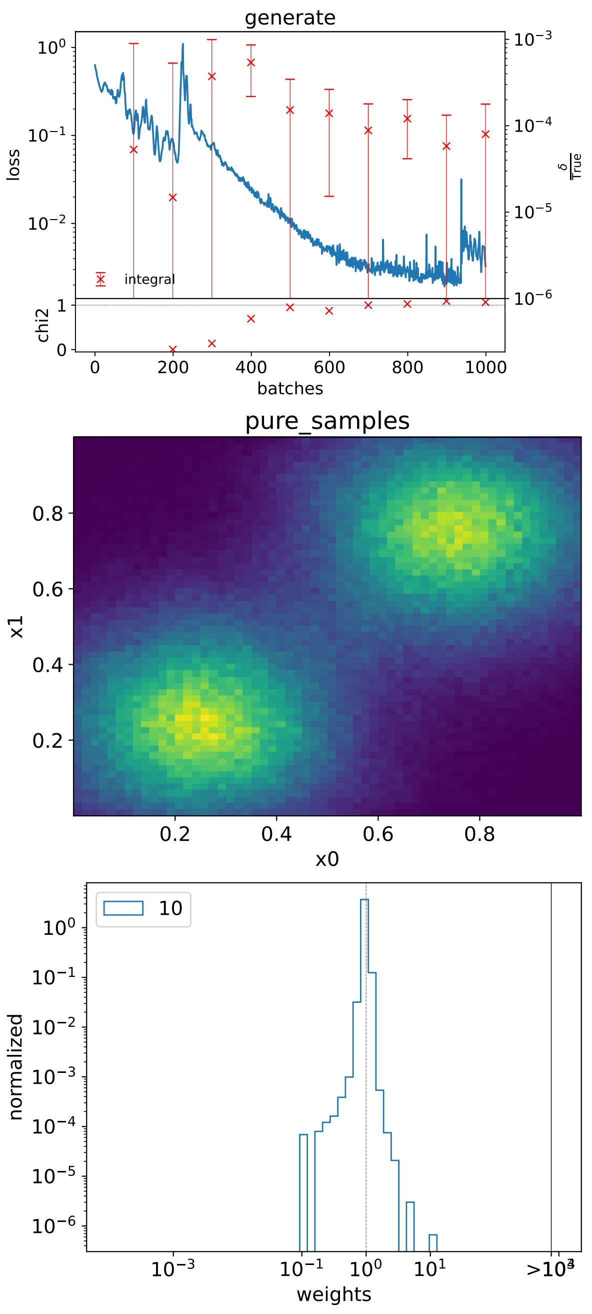
(b) `i-flow` samples

Figure 27: `i-flow` with high lr

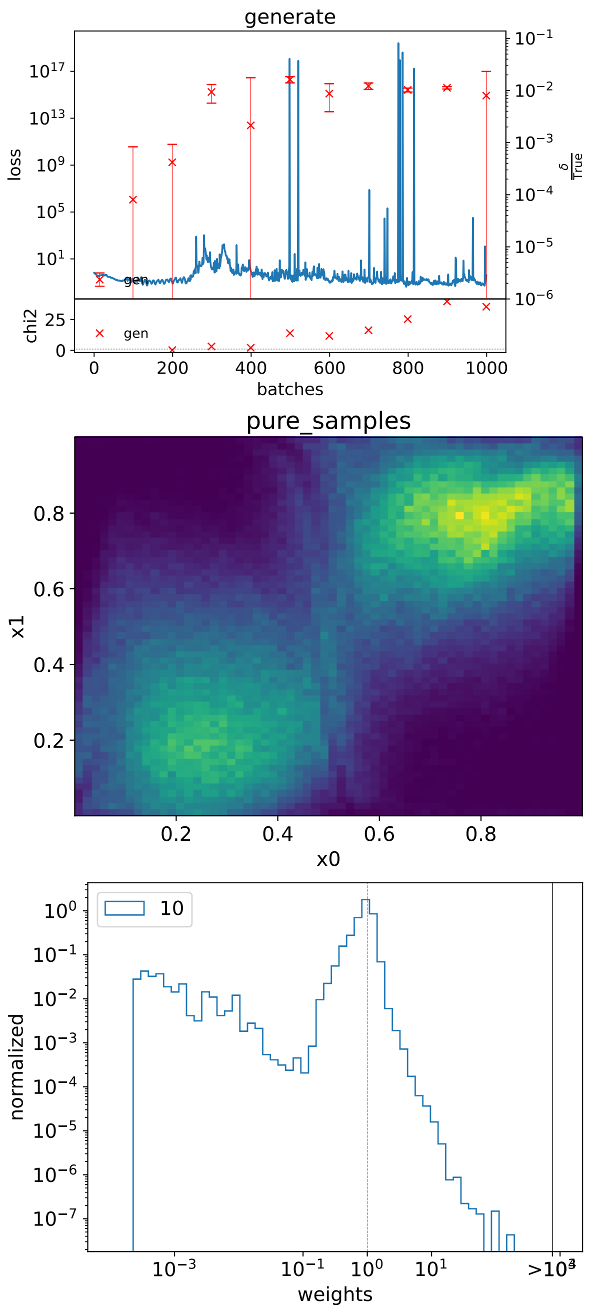
This has the side effect, that the training is *very* fast, taking only a few seconds, while the integral evaluation gets clearly worse, now achiving 0.034% to the true value, yet comfortably within 1 σ. The distribution of weights gets even narrower.
With a network setup of the same size and learning rate, the setup of BIOFLOW is not so stable and behaves differently from run to run with the same setup (Figure 28). Yet even in lucky runs, it is not quite as efficient as `i-flow`, the loss is more noisy and the weight distribution is not so good. For the high learning rate of $1 \cdot 10^{-2}$, the network seems overwhelmed (Figure 29). Because `i-flow` is clearly better in generating training, the next search for the error began.

Soon it was found, that the initialisation of the splines was not correct: Instead of forming the identity transformation, the derivatives at the bin edges were set to ≈ 0.6941 instead to 1, resulting in a "wavy" function. This behavior in 2 Dimensions (For the original network architecture) in the physical space can be seen in Figure 30a. After fixing this, the samples in the physical space in the beginning are uniformly distributed (Figure 30b). Furthermore, `i-flow` uses an uniform sampling from an unit hypercube in the latent space. The permutation applied after the spline coupling is different: `i-flow` works with hardcoded masks that flip the input vector, in BIOFLOW, the rotation matrix was taken from $SO(N)$. This explains the "lucky" and "unlucky" runs in Figure 28: Since, with two coupling blocks, only one rotation is effective and until now continously and random, theres a chance, that one part of the vector is only slightly affected by the transformation of the network, making it very hard to adapt to the target. Furthermore, in this project, the variation ADAMW of ADAM was used. ADAMW should handle the weight decay slightly better, but this is a minor detail.

Implementing these changes yield results with mixed results. In the beginning, both programs behave very similary, showing that these fixes indeed were reasons for the long training time needed. However, while the loss of `i-flow` monotonically decreases with training time (Figure 26c), producing very good weight distributions (Figure 26a), in BIOFLOW, the loss increases again (Figure 31a), which damages the network output distribution and unweighting efficiency (Figures 31b, 31c, 31d). With the high learning rate of $1 \cdot 10^{-2}$ the setup doesnt produce any

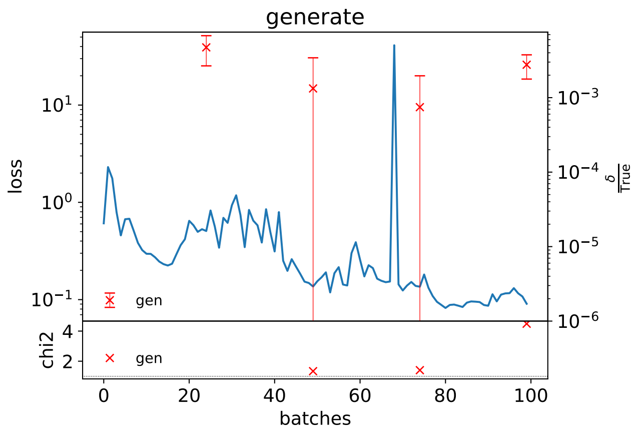


(a) Lucky run with small net



(b) Bad run with small net

Figure 28: BIOFLOW generation with small network setup



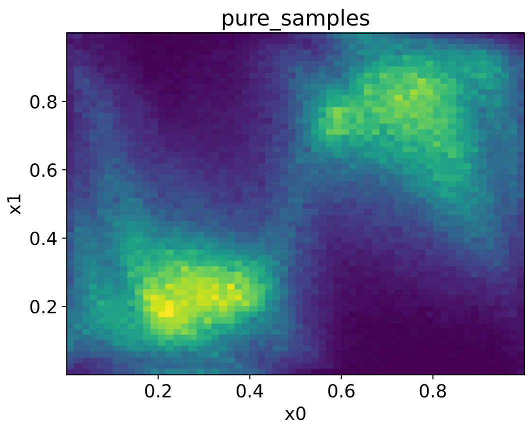
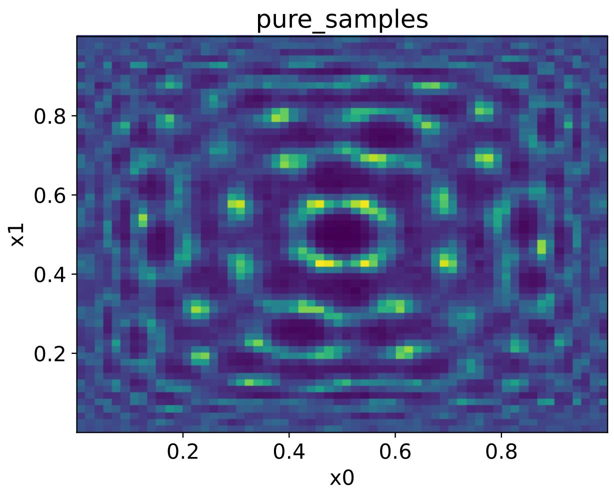
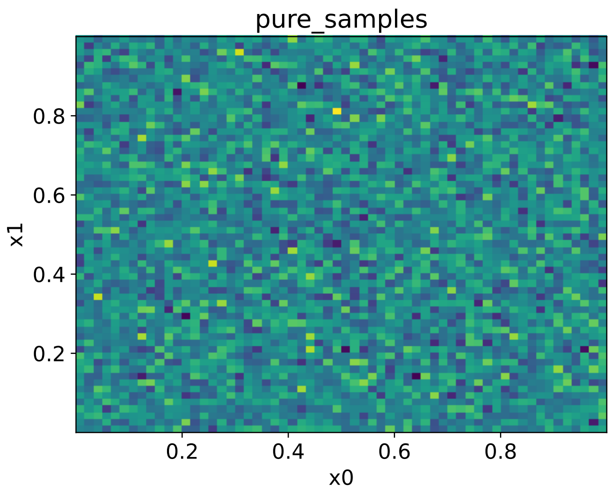


Figure 29: Small network setup with high learning rate

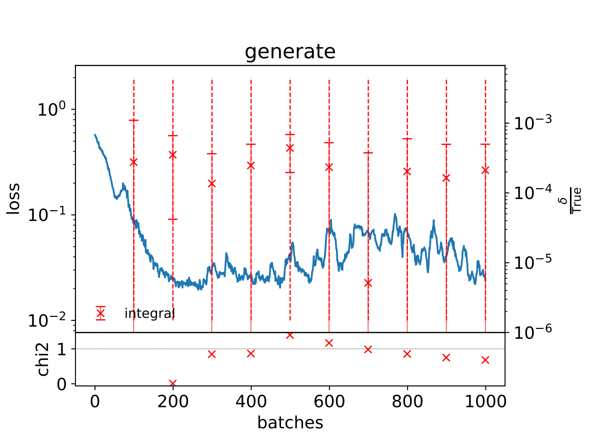


(a) Non-uniform initialisation

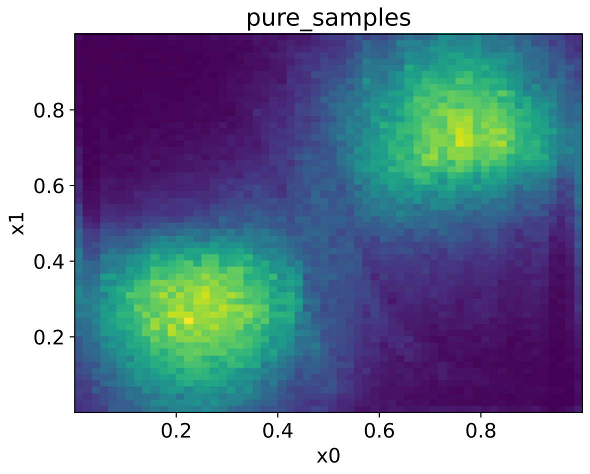


(b) Uniform initialisation

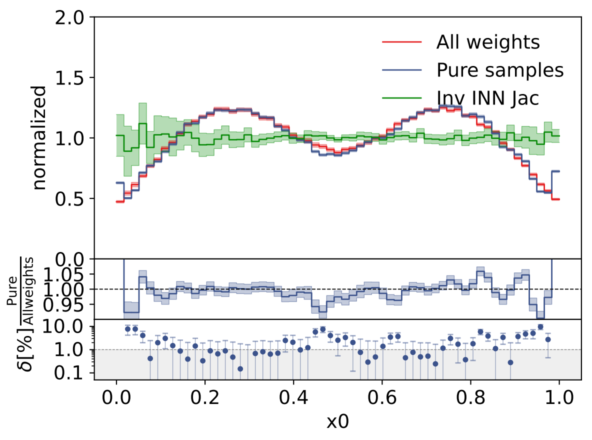
Figure 30: Sample Distributions without training



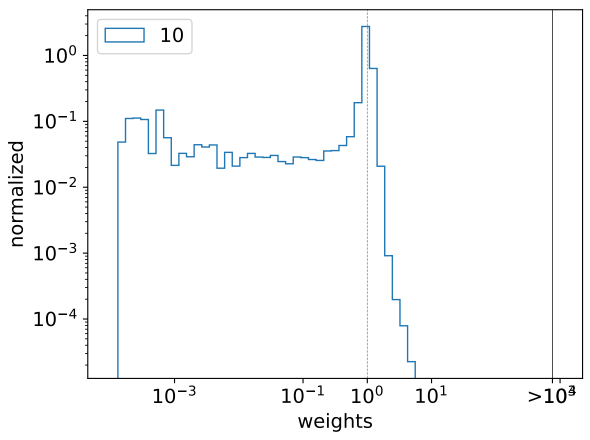
(a) loss increases



(b) Artifacts visible



(c) Artifacts in 1-D



(d) Distorted Weight distribution

Figure 31: Small Network after `i-flow` changes

meaningful results.

This is the current state of BIOFLOW: Because it suffers from increasing and more noisy loss, and falls apart for high learing rates, the result obtained cannot achieve the benchmark.

5.8 Adding recycling

Anyways, to show that the idea of adding recycling training indeed optimizes the procedure, it is implemented again with the small network setup. For only two coupling blocks, taking the rotation matrix form $SO(N)$ doesnt seem to make much sense. For the remaining network setup, gaussian latent space with uniform initialisation (to not be too close to the target function in the beginning, for two coupling blocks the non-uniform initialisation resembles the double gauss already quite a bit) was chosen. For the integration schedule, three setups were tested: One setup is designed to train for a compareable time as I-FLOW with the lower learning rate, one for the fast adaption, and another one aims for a very narrow weight

distribution without paying much attention to runtime.
Note that comparing single runs of BIOFLOW and `i-flow` is difficult, since they were done on changing devices, with also other groups running their code on them, so the programs dont run continoously. For this reason, it was tested once, that recycling one data point takes roughly $\frac{2}{3}$ of the time generataing one, and this is the measure of time.
For the "standard" BIOFLOW setup, the output sample distribution does not look quite perfect, but is reasonably close to the target. Especially the weight distribution is significantly narrower than `i-flow`. The issue with increasing loss during generating training once the network is reasonably adapted still exists, but by recycling, this gets more than counteracted.

For the Fast BIOFLOW setup (Figure 33), the output distribution is, as expected with less training, a bit worse, but the sampled weights still outperform `i-flow`.
Running BIOFLOW for quiet a long time results in a nearly perfect output distribution as well as a very narrow weight distribution (Figure 34).

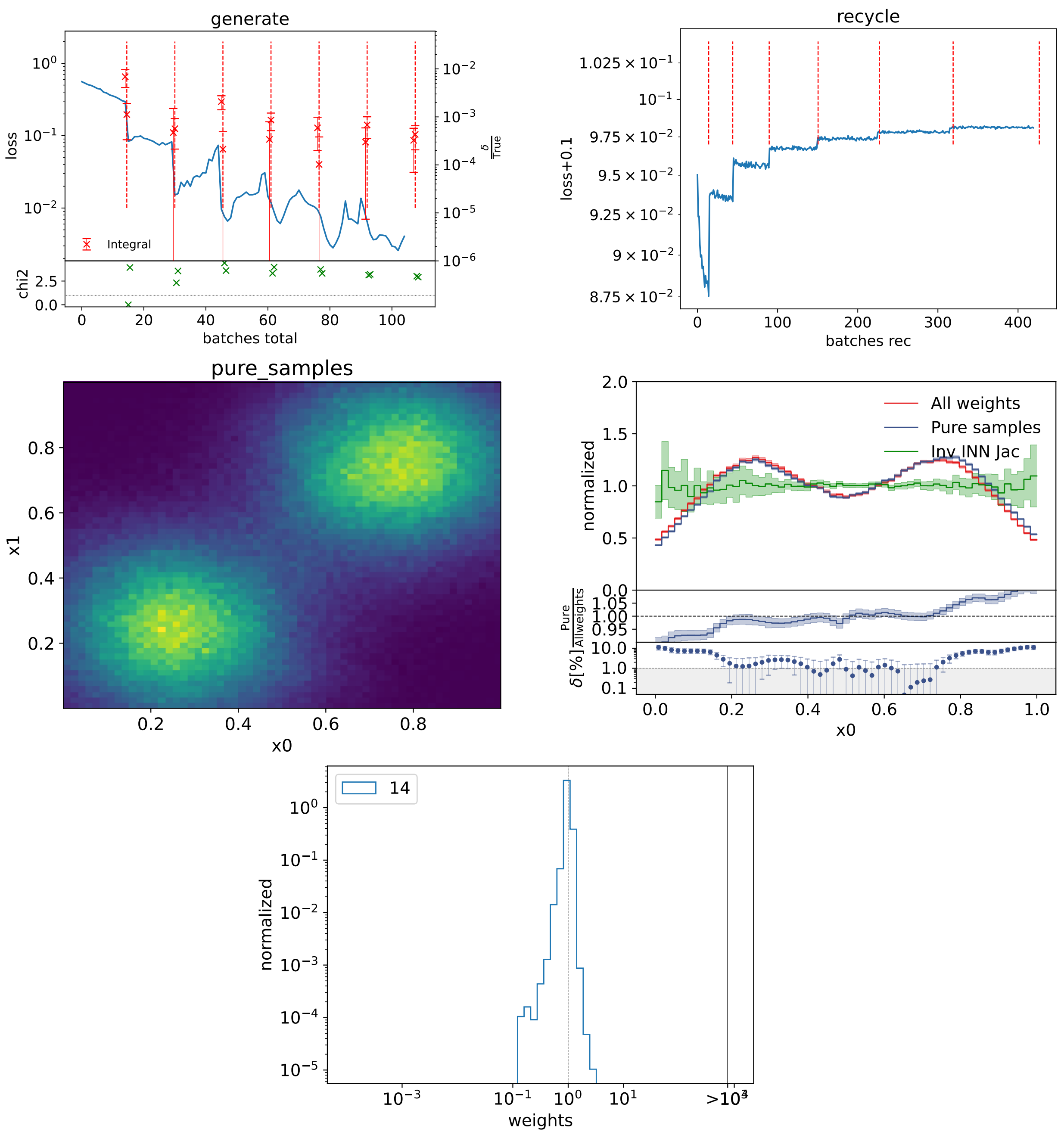


Figure 32: 'Standard' BIOFLOW training

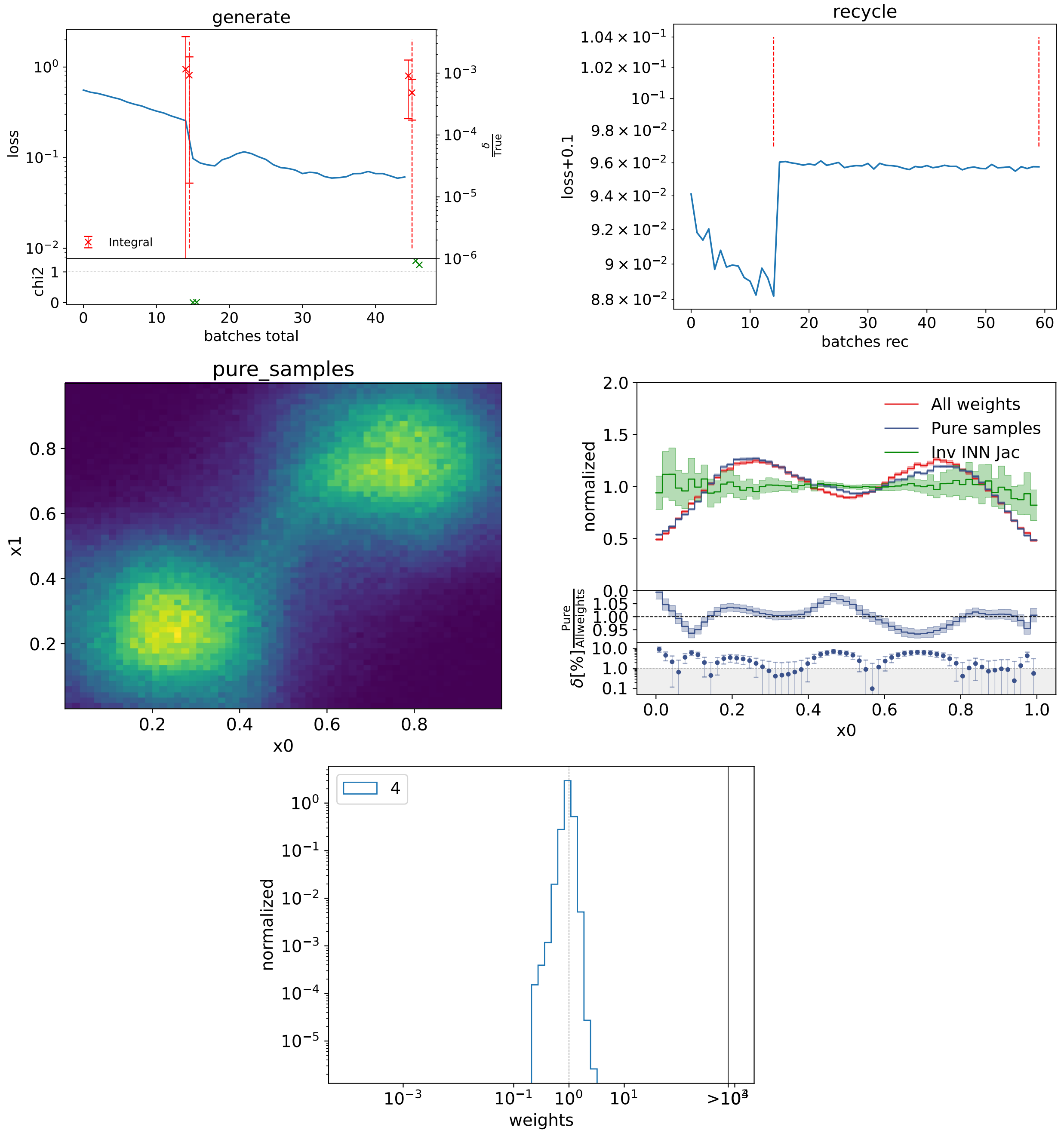


Figure 33: Fast BIOFLOW training

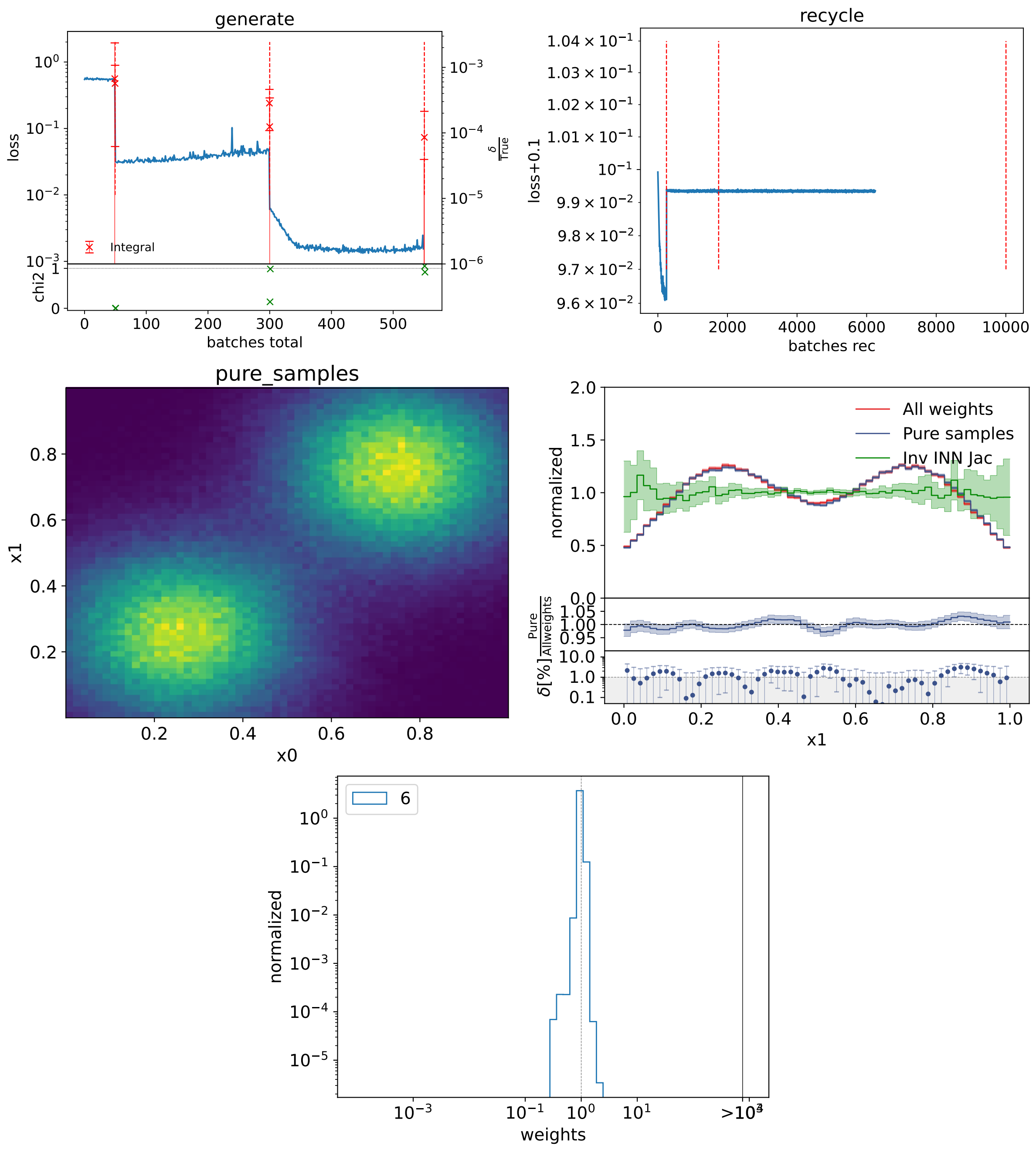


Figure 34: Long BIOFLOW training

6 Conclusion

6.1 Summary

After pointing out all essential theoretical background, program pipelines for integrating differential crosssections and generating weighted events with adaptive monte carlo techniques using VEGAS and INNs were sketched. They resemble each other much, as the INN takes essentially the place of VEGAS, while the rest of the pipeline, most importantly the phase space generator RAMBO, the source for the matrix element $\mathcal{M}$, MADGRAPH, the provider of PDFs (first LHAPDF, later the TDFs) and the final calculation of the integral.
VEGAS was ensured to work as intended by using it on simple toy functions. One important result for later comparison is the distribution of weights for the double gaussian function; it spans about 3.5 orders of magnitude. Then, the whole pipeline was applied to the $\mathbf{gg} \to \mathbf{ggg}$ process. There it turned out, that LHAPDF was an unexpected source of error, that wasnt able to be fixed by simple means. So, the simple "Tilman gluon distribution function" was used to approximate the PDFs. After this fix, everything was working acceptable. With the chosen phase space cuts, the cross section was calculated to $12.78(16)\text{GeV}^{-2}$. Unfortunately, this differs multiple standard deviations to the result of the MADGRAPH framework ($15.296(15)\text{GeV}^{-2}$). After some investigation for this reason, it was suspicious that the final result of VEGAS was fluctuating noticeably more than the estimated error, so maybe the estimation for the standard deviation of VEGAS is incorrect.
Next, having learned from the difficulties working with the whole VEGAS pipeline, the INN was trained on toy functions first. To keep things comprehensible, but allow for some complexity, they were chosen to be two dimensional, but generalizable to higher dimensions, namely the gaussian ring and the double gaussian function. It was soon found out, that the double gaussian was more difficult for the network to adapt to, and that's why it was focused from that time on. In the beginning, a relatively big network architecture was chosen and both training directions were examined on their own, ensuring that both have the ability to improve the network distribution properly. Setting the learning rates too high, the network fails to learn the function, resulting in regions in the integration space, where so samples are taken. Combining both training directions, very good adaptions were achieved and the final weight distributions outperform VEGAS by a long shot, spanning less than two orders of magnitude; however it should be mentioned, that the integrand is chosen, such that the drawbacks of VEGAS are heavily punished. The best results are obtained by a training scheduler that recycles data in the end; this is expectable, as in this way, the network iterates over the most data points and the training takes the longest. It was found out that the value of the recycling loss is not so expressive for the quality of the trained distribution.
Next, some different loss functions for generative training were tried out. Many of them were divergences motivated by `i-flow`, and about half of them worked right out of the box, but the until now used exponential loss had a slight edge over them all. The others either produced distorted or entirely unrecognizeable distributions, indicating that some tuning of the network or sometimes of tunable parameters in the losses themselfe would be needed. However, as one very well working loss was already identified, this was not pursuit further. Lastly, for less abstract loss functions, directly the variance of the integral and a asymetric punishment of extreme weight values were used. While the variance-loss worked great, competing with the exponential loss, no good working implementation of the weight punishment was found.

A very similar project to Bioflow is `i-flow`. It only implements generative training and is the perfect choice to create a benchmark for this direction, to have a gauge, how fast good adaption is possible. Its standard architecture is quite different from Bioflow, especially being way smaller allowing for stable training with a high training rate. With this setup, training can take a very short amount of time, yet still produces impressive output: The weight distribution only spans about two orders of magnitude and having very little tail on high weight values. Impressively, the weight distribution slightly improves with higher learning rate and correspondingly less data. Changing Bioflow accordingly, it suffers from increasing loss once the network is reasonably adapted, preventing it from reaching that benchmark. The reason for that is still searched for. There might be an error in the implementation of the spline coupling, as for very high learning rates, there occurs an error if a discriminant is smaller than zero. It is suspected, that the network tries to not be invertible any more.
Nevertheless, even though the benchmark was not achieved, recycling training was added again. In contrast to the generative training, it behaves very similary with the small network architecture, always improving the distribution and lowering the generative loss again. This counteracted the problem of increasing generative loss during generative training so much, that Bioflow is able to produce even narrower weight distributions than `i-flow`, spanning significantly less than two orders of magnitude with compareable training time, although the resulting sampling distribution is not in perfect accordance with the target in the case of fast training with high learning rate. Very long training improves the network output even more.

6.2 Outlook

It was shown, that the task Vegas performs can be done by neural networks better, as they make less assumptions about the integrand. Unfortunately, as mentioned, the `i-flow` benchmark could not be achieved with Bioflow, however, it was shown, that the alternative training direction indeed in principle improves the network performance. Reaching the benchmark will be an important milestone for Bioflow; the reason for staying behind is still searched for. If it won't be found for some time, perhaps some help will be contributed by one of the co-workers on `i-flow` joining the ressearch group anticipated later this year. Next to that, going from a two dimensional toy function to the physical process will be an essential task that will take some time: Slowly increasing the dimesnionality of thye integrand, the optimal parameters for the network will be seaerched for and tried to find a rule how they develope, to gain maximum efficiency. Furthermore, the training schedule was hardcoded all the way. A primitive auto scheduler, switching the training directions as deemed appropriate was developed, but until now not further pursuited. Improving this scheduler might yield optimal results for a wide range of functions without much necessity for tuning by hand. It also might be worth a try, if pretraining the network will make generalisation to other processes more efficient and effective. Lastly, Bioflow might be a candidate for a surrogate for rejection sampling proposed by [4] for computationally very expensive processes, as it needs due to recycling training very few samples to capture the rough shape of the integrand.

References

[1]Bergström, *Dark matter candidates*, 2009.

[2]G. Peter Lepage, “A new algorithm for adaptive multidimensional integration”, Journal of Computational Physics **27**, 192–203 (1978).

[3]Gao, Isaacson, and Krause, *I-flow: high-dimensional integration and sampling with noramlizing flows*, 2020.

[4]Danyiger, Janßen, Schumann, and Siegert, *Accelerating monte carlo event generation - rejection sampling using neural network event-weight estimates*, 2021.

[5]Bothmann, Janßen, Knobbe, Schmale, and Schumann, *Exploring phase space with neural importance sampling*, 2020.

[6]F. James, *Monte carlo theory and practice*, 1980.

[7]Peskin and Schroeder, *An introduction to quantum field theory* (CRC Press, 1995).

[8]Alwall, Frederix, Frixione, Hirschi, Maltoni, Mattelaer, Shao, Stelter, Torielli, and Zaro, *The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations*, 2014.

[9]Plätzer, *Rambo on diet*, 2018.

[10]NGoetz, *Nf*, (2020) `https://github.com/NGoetz/NF/tree/master/nisrep/PhaseSpace` (visited on 02/16/2022).

[11]Gao, Höche, Isaacson, Krause, and Schulz, *Event generation with normalizing flows*, 2020.

[12]Stienen and Verheyen, *Phase space sampling and inference from weighted events with autoregressive flows*, 2020.

[13]Kingma and L. Ba, *Adam: a method for stochastic opimization*, 2015.

[14]D. J. C. MacKay, *Information theory, inference, and learning algorithms* (Cambridge University Press, 2003).

[15]G. P. Lepage, *Vegas 5.1.1 documentation*, (2021) `https://vegas.readthedocs.io/en/latest/tutorial.html` (visited on 02/21/2022).

[16]Hartung, Knapp, and Sinah, *Statistical meta-analysis with applications* (Wiley, 2008).

[17]Smith and Topin, *Super-convergence: very fast training of neural networks using large learning rates*, 2018.

[18]T. Contributors, *Pytorch documentation*, (2019) `https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html` (visited on 02/24/2022).

[19]Gao, Isaacson, and Krause, *I-flow*, (2020) `https://gitlab.com/i-flow/i-flow/-/tree/master` (visited on 03/04/2022).

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.
Erlangen, den 17.03.2022,

[illegible]