Department of Physics and Astronomy
Heidelberg University

Bachelor Thesis in Physics
submitted by

Joran Valentin Köhler

born in Bad Salzungen (Germany)

1998

fINNishing vegas

Replacing the multidimensional integration module vegas by an Invertible Neural Network.

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

Abstract

This thesis is about working on a replacement of the vegas algorithm [1]. It gives estimates for the integral of an arbitrary multidimensional function. Even though it is well approved and a common choice also in high energy physics, it has some weaknesses. Neural Networks with their easily increased complexity now give the chance to outperform this simple algorithm for functions with high complexity. We combine two different types of training for an Invertible Neural Network (INN). The goal is to get a precise result for the integral and, at the same time, being able to generate samples, as the target distribution would do. Finally we challenge the state-of-the-art benchmark I-flow [2] with our enhancement BiG-flow.

Zusammenfassung

Diese Arbeit dreht sich um den Nachfolger vom vegas Algrithmus [1]. Dieser errechnet numerische Schätzungen für das Integral von beliebig vieldimensionalen Funktionen. Zwar ist der Algorithmus bewährt und auch in der Hochenergiephysik die übliche Wahl, doch hat er auch einige Schwächen. Neuronale Netzwerke bieten nun die Möglichkeit, die Ergebnisse dieses einfachen Algorithmus für komplexere Funktionen zu übertreffen. Für das Training unseres Invertierbaren Neuronalen Netzwerks (INN) kombinieren wir zwei verschiedene Trainingsarten. Das Ziel ist es einerseits, das Integral mit hoher Präzision abzuschätzen und andererseits genau so Daten zu generieren wie es die Zielverteilung tun würde. Schließlich messen wir uns an der aktuellen Benchmark I-flow [2] mit unserem erweiterten Model BiG-flow.

Contents

1 Introduction

At the European Organization for Nuclear Research (CERN) in Switzerland physicists are searching for new physics for example in form of dark matter [3] or Super Symmetry partners of particles [4]. The experiments in the huge colliders shall confirm or discount particles which would be part of the theories beyond the known theory of the *Standard Model of Particle Physics* (SM). Part of this approach is to make predictions upon the theories to test. This subject, which connects the theory with experiments is called phenomenology. To test a prediction, often not even very precise tests are needed, but rather a fast approximation is sufficient. Such a test often includes the calculation of a cross section of a given physical process. Practically this mostly means integrating a high-dimensional integrand with high complexity. An analytical solution of such an integral in requires a lot of resources, if at all possible. There is already a vast number of numerical methods which are more or less efficient in estimating the integral. We will focus on the method which is called *importance sampling*. Here the sampling distribution has to get modelled like the integrand. In current frameworks for event simulation, an algorithm called vegas is used for this task. It is an algorithm with a relatively simple principle, invented in 1978 by G. P. Lepage [1] and continuously upgraded since then.
At the same time we live in the age of Neural Networks (NNs). They are helping us to find - more or less reliably - a highly complex function which maps given input to the right output. The aim of this thesis shall be to tackle exactly this task of integration with Invertible Neural Networks [5][6][7][8]. These are special NNs, representing a trainable bijective map between a simple distribution on the one side and a highly complex distribution - for example from a real world problem - on the other side. The invertibility makes this kind of networks more explainable, which is a big gain for applications in science like physics or medicine.
A pioneer applying INNs for integration is the group of C. Krause, having introduced a network called I-flow [2]. This is also based on an INN, but only trained in one direction. We will try to expose the advantage of exhausting the full potential of the INNs: We combine two types of training, where one is in the same way as in I-flow and the other one processes the generated data in the opposite direction of the network.
Integrating with this method involves sampling as realistic as possible. I.e. the distribution of events we are simulating should be as close to real event data from a particle collider as possible. Though this could be a nice side effect, it allows us also to formulate this as the second goal. Thus we can assess our results under two different objectives.

At first in section 2 we will comprehend the physical background. Besides we will get to know possibilities to simulate the physics with numerical modules. After that we will take a look to fundamentals of Neural Networks and INNs in section 3 and 4. In section 5 we will introduce Monte Carlo Integration and the implementations like vegas or the INN methods. After that we are ready to draft the architectures for the integrator and event simulator in section 6. The results of vegas will be shown in section 7 and the results of our INN training in section 8. Finally in

section 9 we will let us lead by the implementation of I-flow and challenge it with our enhancements.
Because the project was done in collaboration, content and figures in this thesis can equal the ones in the Bachelor thesis of Simon Pijahn.

2 LHC Physics and numeric approximations for event generation

The Large Hadron Collider (LHC) at CERN is a particle collider, where massive particles like protons collide at high center of mass energy $\sqrt{s}$. The experiments are done to search for new physics beyond the SM. For proton-proton collisions it reaches around $\sqrt{s}$ =13TeV. The outgoing particles like quarks and gluons are characterised by their four momenta $p = (E, p_x, p_y, p_z)$. However these high energy particles are not measured directly by the detector. By successive branching, the particles create a collimated shower and below a certain energy, quarks and gluons hadronise again. This means at this energy level, the QCD coupling constant α_s dominates and forces every colour-charged particle to form a neutral (colourless) object. So quark pairs and gluons are bound to each other and for instance if two quarks are ripped apart, a new quark-anti-quark pair will be produced out of the interaction energy. Such an ensemble of particles is called a *jet*. The four-momenta of the jet particles is then calculated from various kinds of calorimeters and other detectors. Additional to the events in the so called pre-detector level, detector effects are included. Due to imperfections, the detectors always alter the actual signals at least a bit. Though in this thesis we do neither consider these detector effects, nor the hadronisation in the jet. We will concentrate on the physics in the parton level.

2.1 Matrix element and cross section

To simulate the generation of LHC events and to determine the cross section of a certain process, we need to calculate the probability amplitude for producing the given final state out of a given initial state. To comprehend the theoretical background we follow here Peskin / Schröder [9]. To represent the inital-state particles, we take wavepackets . We evolve these with the time-evolution operator and overlap the result with the given final state. In the case of two incoming particles, we can express this probability for example as

$$\mathcal{P} = |\langle \underbrace{\phi_1 \phi_2 ...}_{\text{future}} | \underbrace{\phi_A \phi_B}_{\text{past}} \rangle|^2 \tag{2.1}$$

where $|\phi_A \phi_B\rangle$ represents the initial state of two wavepackets, emerged in the past (time -t) and $\langle \phi_1 \phi_2 ...|$ the final state of several wavepackets emerged in the future (time t). We used here the Heisenberg picture, e. g. the states are time-independent. Though the names do indicate, that the states are defined at different times.
As our data are definite momenta in the end, we also want to calculate with states of definite momentum. Therefore we assume that the states are constructed independently at different locations. Thus we can leave the momentum integration of

the Fourier-transformed wave function out and can calculate with the transition amplitude of *in* and *out* states of definite momentum:

$$ {}_{\text{out}}\langle \mathbf{p}_1\mathbf{p}_2...|\mathbf{k}_A\mathbf{k}_B\rangle_{\text{in}} = \lim_{t\to\infty} \langle \underbrace{\mathbf{p}_1\mathbf{p}_2...}_{t} \,|\, \underbrace{\mathbf{k}_A\mathbf{k}_B}_{-t}\rangle \tag{2.2} $$

$$ = \lim_{t\to\infty} \langle \mathbf{p}_1\mathbf{p}_2...|e^{-iH(2t)}|\mathbf{k}_A\mathbf{k}_B\rangle = \langle \mathbf{p}_1\mathbf{p}_2...|S|\mathbf{k}_A\mathbf{k}_B\rangle \tag{2.3} $$

In equation 2.3 we then evolved an expression where the states are defined at the same reference time by adding the time-evolution operator with Hamiltonian H. This again can be written as an unitary operator and the limit in time is then called the *S-matrix*. To separate actual interaction from non-interacting transition we can write

$$ S = \mathbb{I} + iT \tag{2.4} $$

Lastly we also want to extract the four-momentum conservation from the matrix element and hence we arrive at

$$ \langle \mathbf{p}_1\mathbf{p}_2...|iT|\mathbf{k}_A\mathbf{k}_B\rangle = (2\pi)^4\delta^4(k_A + k_B - \sum p_f) \cdot i\mathcal{M}(k_A, k_B \to p_f) \tag{2.5} $$

Practically we will evaluate the matrix element for the desired process with the Madgraph framework [10].

The cross section σ is now defined as following. We consider a beam of particles targeting another beam of particles. Now we expect some number of events and this should be proportional to the cross-sectional area, the density distribution of the beams and the length of the beams. Now the cross section is the number of events we actually observe divided by all these quantities (so called *luminosity*), it should be proportional to.
With our matrix element, defined in Equation 2.5, the cross section for two initial particles can developed as

$$ d\sigma = \frac{|\mathcal{M}(k_A, k_B \to \{p_f\})|^2}{2E_A 2E_B |v_A - v_B|}(2\pi)^4\delta^4(k_A + k_B - \sum p_f)\prod_f \frac{d^3p_f}{(2\pi)^3}\frac{1}{2E_f}. \tag{2.6} $$

Since only the highly relativistic case is relevant we can simplify this by

$$ |v_A - v_B| = c = 1 \;\text{ and }\; 4E_AE_B = 2(k_A + k_B)^2 = 2s \tag{2.7} $$

and the definition

$$ dX := (2\pi)^4\delta^4(k_A + k_B - \sum p_f)\prod_f \frac{d^3p_f}{(2\pi)^3}\frac{1}{2E_f} \tag{2.8} $$

to

$$ d\sigma = \frac{|\mathcal{M}(k_A, k_B \to \{p_f\})|^2}{2s} dX. \tag{2.9} $$

2.2 Parton Distribution Function and LHAPDF module

In this thesis we will focus on proton proton collisions. Here always collide only constituents (*partons*) of the protons, which have different fractions of the proton momentum. Because these fractional momenta are mainly collinear with the fraction of the proton, these fractions are also called *longitudinal fractions*. To take this additional selection of constituents into account, we have to include the so called parton distribution function (PDF) in our calculation. This function is the probability of finding the specific parton f with longitudinal fraction ξ [9].

LHAPDF is a general purpose C++ interpolator, used for evaluating PDFs from discretised data files [11]. These data files are provided as so called PDF sets and besides meta data they contain PDF values for each flavour on a rectangular grid of "knots" in the plane (ξ, Q^2). Here Q^2 is the factorization scale, which is defined as the negative squared momentum of the hadron, that collides. Thus at each point there are values for all flavours given. The values come from the evaluation of fits by different groups. In the range of the fits LHAPDF interpolates in $\log(Q^2)$–$\log(\xi)$ space and outside of the fit range extrapolates in a similar programatic way.

2.3 Phase space parametrisation and RAMBO algorithm

As described above we consider colliding partons which do not have the same momentum. But to simplify the calculations we would like to be in the centre of mass frame and therefore we have to boost the momenta from the lab frame. We can find better parameters than the four-momentum, to characterise the particle, to make it easier to compare particles in different frames: (E, p_x, p_y, p_z) we can describe by E, the transverse momentum p_T, the azimuthal angle ϕ and the rapidity y.

$$p_T = \sqrt{p_x^2 + p_y^2} \quad y = \frac{1}{2}\log\left(\frac{E + p_z}{E - p_z}\right) \tag{2.10}$$

Comparing two particles rapidity, the difference will be invariant under boosts along the z-direction. In our highly relativistic cases, we can write this also as the pseudo rapidity η:

$$\eta = \frac{1}{2}\log\left(\frac{|\mathbf{p}| + p_z}{|\mathbf{p}| - p_z}\right) = \log\left(\tan\left(\frac{\theta}{2}\right)\right) \tag{2.11}$$

with the polar angle θ. η and ϕ now parametrise a cylindrical surface, also referred to as the η-ϕ-plane. Here we can define the separation of two particles i and j with

$$\Delta R_{ij} = \sqrt{\Delta\eta_{ij}^2 + \Delta\phi_{ij}^2}. \tag{2.12}$$

The minimum of needed parameters is equal to the degrees of freedom of the considered process. For the example of two colliding high energy partons producing again two outgoing partons we can reduce the needed number of parameters by 4. This is because the overall p_T is zero and because we consider both to be light-like particles. For n outgoing particles this makes in general 3n-2 degrees of freedom,

including the momentum fractions.
We call the reduction of phase space parameters here phase space parametrisation and the transformation back to four-momenta phase space generation. Of course there are arbitrarily many ways to parametrise a phase space. One automation is done by the so called RAMBO algorithm. RAMBO is a phase space generator, first implemented by R. Kleiss and W.J. Stirling in 1985 [12]. Then in 2013 S. Plätzer could refine the algorithm to *RAMBO on diet*, such that it indeed only needs the number of degrees of freedom as input size [13]. This input gets transformed into the four-momenta of the final state by successive branching the tree of decays. Therefore it takes the energy of the intermediate particle as new center of mass energy and boosts into its centre of mass frame to find the new decay products. In the following section we will comprehend the algorithm for two different examples.

The algorithm for 2→2 scattering

In the following we comprehend the algorithm, taken from [13], for the case of two particles in the initial and the final state. Additionally we assume here that the two initial particles collide completely and their masses are neglectable. The final particles masses are set to zero.
As input parameters RAMBO takes variables $\{r_i \in \mathbb{R} \,|\, 0 < r_i < 1\}$ and the centre of mass energy $\sqrt{s}$ of the colliding hadrons. The RAMBO algorithm than looks as following:

Algorithm 1 RAMBO for n=2

Require: $r_1, r_2, \sqrt{s}$

1: $M_1 \leftarrow \sqrt{s}$, $M_2 \leftarrow 0$
2: $\cos\theta \leftarrow 2r_1 - 1$
3: $\sin\theta \leftarrow \sqrt{1 - \cos^2\theta}$
4: $\phi \leftarrow 2\pi r_2$
5: $q_2 \leftarrow \frac{M_1}{2} = \frac{\sqrt{s}}{2}$
6: $\boldsymbol{p}_1 \leftarrow q_2 \begin{pmatrix} \cos\phi\sin\theta \\ \sin\phi\sin\theta \\ \cos\theta \end{pmatrix} = \frac{\sqrt{s}}{2} \begin{pmatrix} \cos\phi\sin\theta \\ \sin\phi\sin\theta \\ \cos\theta \end{pmatrix}$
7: $p_1 \leftarrow (q_2, \boldsymbol{p}_1)$, $Q_2 \leftarrow (q_2, -\boldsymbol{p}_1)$
8: (boost by (1,0,0,0))
9: $p_2 \leftarrow Q_2$

For our test run we take $\sqrt{s} = 1000$, $r_1 = 0.1$ and $r_2 = 0.5$. That means we have $\cos\theta = -0.8 \Rightarrow \sin\theta = 0.6$ and $\phi = \pi$.
Going further through the algorithm gives us the four-momenta:

$$p_1 = \begin{pmatrix} 500 \\ -300 \\ 0 \\ -400 \end{pmatrix} \text{ and } p_2 = \begin{pmatrix} 500 \\ 300 \\ 0 \\ 400 \end{pmatrix} \tag{2.13}$$

The algorithm for proton-proton collision

Now let us consider two incoming protons, where only a fraction of each collides. This means we have now two additional input variables $r_3, r_4 \in (0,1)$ providing the fractions. The only difference here is that we have to adjust s to $\hat{s}$ and then boost our result from above in z-direction. $\hat{s}$ is determined by

$$\hat{s} = (r_3 p_{p1} + r_4 p_{p2})^2 \approx 2 r_3 r_4 p_{p1} p_{p2}, \tag{2.14}$$

where $p_{p1} = (\sqrt{s}/2, 0, 0, \sqrt{s}/2)$ and $p_{p2} = (\sqrt{s}/2, 0, 0, -\sqrt{s}/2)$ are the four-momenta of the incoming protons. This means:

$$\hat{s} = 2 r_3 r_4 2s/4 = r_3 r_4 s \tag{2.15}$$

The boost can be executed with Boost matrix in z direction, where β can be determined as following:

$$\beta = \frac{\gamma m' \beta}{\gamma m'} = \frac{p_z}{E_{labframe}} = \frac{(r_4 - r_3)\sqrt{s}/2}{(r_3 + r_4)\sqrt{s}/2} = \frac{r_4 - r_3}{r_3 + r_4}, \tag{2.16}$$

where $p'_z = (r_4 - r_3) p_z$ with $p_z = \sqrt{s}/2$ and the energy of the colliding fraction in the lab frame $E_{labframe} = (r_3 + r_4)\sqrt{s}/2$.

Let us take for example $r_3 = 0.1$ and $r_4 = 0.4$, than with the same parameters r_1 and r_2 from above we get in the center of mass frame:

$$p'_1 = \begin{pmatrix} 100 \\ -60 \\ 0 \\ -80 \end{pmatrix} \text{ and } p'_2 = \begin{pmatrix} 100 \\ 60 \\ 0 \\ 80 \end{pmatrix} \tag{2.17}$$

Boosting this to the lab frame by β gives the following four-momenta:

$$p_1 = \begin{pmatrix} 65 \\ -60 \\ 0 \\ -25 \end{pmatrix} \text{ and } p_2 = \begin{pmatrix} 185 \\ 60 \\ 0 \\ 175 \end{pmatrix} \tag{2.18}$$

The implementation we use executes the boost in the following way:

Algorithm 2 boost to lab frame

Require: $p_{in1}, p_{in2}, p_{out}, r_3, r_4$
1: reflab $\leftarrow (p_{in1} \cdot r_3 + p_{in2} \cdot r_4)$
2: **if** reflab[0] < 0 or $(\text{reflab})^2 < 0$ **then**
3: print(invalid boost)
4: $\boldsymbol{\beta} \leftarrow$ reflab[1 :]/reflab[0]
5: $\gamma \leftarrow \frac{1}{\sqrt{1-\boldsymbol{\beta}^2}}$
6: **for** p in p_{out} **do**
7: $\beta_p \leftarrow \boldsymbol{\beta} \cdot p[1:]$
8: $p[0] \leftarrow \gamma(p[0] + \beta_p)$
9: $\gamma' \leftarrow (\gamma - 1)/\boldsymbol{\beta}^2$
10: $f \leftarrow \beta_p \gamma' + \gamma p[0]$
11: $p[1:] \leftarrow p[1:] + f\boldsymbol{\beta}$
return p_{out}

Here the line 4 is exactly the calculation for eq. (2.16).

RAMBO weights

RAMBO returns also a weight for each four-momentum in the output. Partly it can be understood as the Jacobian of the phase space generation, though in our massless case the Jacobian will always be constant. Instead the weight is mainly carrying the phase space volume corresponding to the given fractions or the resulting $\hat{s}$. Thus the weight w is calculated by

$$w = \frac{(\pi/2)^{n-1} \cdot \hat{s}^{n-2}}{(2\pi)^{3n-4} \cdot (n-1)! \cdot (n-2)!} \tag{2.19}$$

with n massless final states and thus $3n-4$ degrees of freedom [13].
The implementation of RAMBO we use, has also an option to specify several cut-offs in the phase space. This is useful to avoid regions, where the matrix element diverges a priori. These cuts are implemented by setting the weights of the corresponding event to zero.

2.4 gg → ggg

We decided to work with the process of two colliding gluons, producing three gluons. This is a rather simple QCD-process with light-like four-momenta. The initial state is well defined by the PDFs for the gluon. The seven degrees of freedom make the problem for INN seven dimensional, which is challenging, but still doable.
With Equation 2.9 and 2.15 we can write down the cross section for this process as

$$d\sigma = \frac{\text{PDF}_g(r_1)\text{PDF}_g(r_2)\left|\mathcal{M}(k_{g1}, k_{g2} \to p_{g3}, p_{g4}, p_{g5})\right|^2}{2r_1 r_2 s} dX', \tag{2.20}$$

where dX' is dX adapted to our parametrisation with $3 \cdot 3 - 4$ parameters.
For this process we have to consider the *infrared divergence* for $\hat{s} \to 0$ as well as the *collinear divergence* for $\theta \to 0$. For the first one we can implement in RAMBO the ΔR-cut and for the second one the p_T-cut.

3 Fundamentals of Neural Networks

Let us go roughly through some fundamental concepts of Deep Learning. In this section we will mainly follow Goodfellow, Bengio and Courville [14].

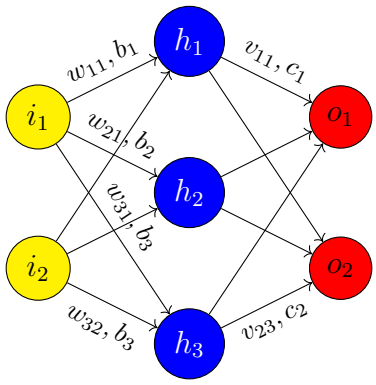


Fig. 1: Scheme of a simple neural network with one input layer, one hidden layer and one output layer

A neural network consists of one input layer, at least one hidden layer and one output layer. These layers contain so called nodes, which represent the neurons of the network. On each edge (arrow in Figure 1) there is a weight w_{ij}. In each node h_i, except in the input nodes, a non-linear activation function f acts on the input x_j with a specific bias b_i. We can write:

$$h_i = f\left(\sum_j w_{ij}x_j + b_i\right) \tag{3.1}$$

or with the whole layer:

$$\mathbf{h} = f\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right) \tag{3.2}$$

Here we want to consider only the very simplest standard activation function, since we are using it also in our implementation. It is called Rectified Linear Unit (ReLU) and defined as:

$$\text{ReLU}(\text{input}) = \max(0, \text{input}) \tag{3.3}$$

3.1 Universal Approximation Theorem

The success of Neural Networks is grounded on the **Universal Approximation Theorem** [15]:
Let $\phi(x)$ be a non-constant, bounded and non-decreasing continuous function. For any $\epsilon > 0$ and any continuous function f on $[0,1]^D$ there exists an $N \in \mathbb{N}, \mathbf{v} \in \mathbb{R}^N, \mathbf{b} \in \mathbb{R}^N$ and $\mathbf{W} \in \mathbb{R}^{N \times D}$, such that if we denote

$$F(\mathbf{x}) = \mathbf{v}^T \phi(\mathbf{W}\mathbf{x} + \mathbf{b}), \tag{3.4}$$

then for all $\mathbf{x} \in [0,1]^D$

$$|F(\mathbf{x} - f(\mathbf{x})| < \epsilon. \tag{3.5}$$

Later the theorem was proofed also for ϕ =ReLU [16]. We can sketch the proof here in Figure 2:
If a function is continuous on a closed interval, it can be approximated by a sequence of lines to arbitrary precision.

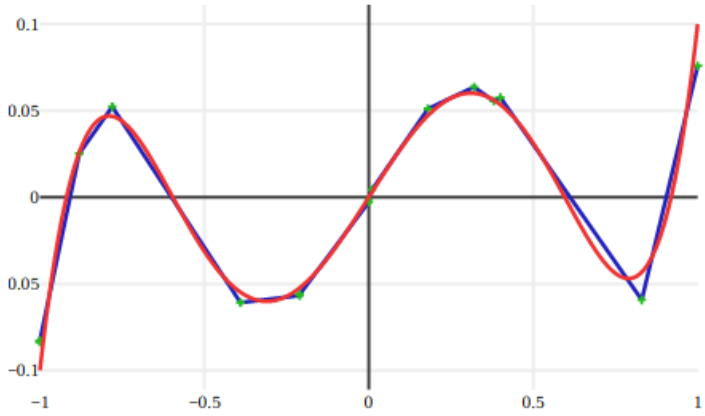


Fig. 2: Approximation by ReLU taken from [17]

By setting the biases b_i we can create exactly this sequence of linear segments as a sum of ReLU units.

3.2 Loss and Optimiser

In neural networks the training process is implemented by calculating the gradient of a *loss function* and updating the parameters. This loss function is a metric how big the error of the network is. It has to be differentiable.
The most basic implementation of updating parameters is the Stochastic Gradient Descent Algorithm (SGD): Given a network as computing function $f()$, it takes a *minibatch* of m training samples, e.g. a fraction of the training batch. This consists for example of training data $\mathbf{x}$ and the corresponding target values y. Now it calculates their mean gradient $\mathbf{g}$ of the loss function L and then updates the parameters Θ by subtracting $\mathbf{g}$, scaled by the learning rate α smaller than one.

Algorithm 3 SGD Optimiser

Require: $f()$, Θ, α

1: **while** stopping criterion is not met **do**
2: Sample minibatch of m training samples $(\mathbf{x}^i, y^i)$
3: $\mathbf{g} \leftarrow \frac{1}{m} \frac{\partial}{\partial \Theta} \sum_i L(f(\mathbf{x}^i; \Theta), y^i)$
4: $\Theta \leftarrow \Theta - \alpha \mathbf{g}$

Over the years this algorithm got improved further and further. The main gain of current versions is that the direction of the previous gradients are included by the *momentum*. The most common optimiser nowadays is called Adam [18] and we will use it always in our implementation. The algorithm additionally takes the constants ϵ, β_1 and β_2. Their default values are listed in Table 1.

The algorithm uses three additional variables: $\mathbf{s}$ is the momentum, which contains

Adam parameter	default value
α	0.001
ϵ	10^{-8}
β_1	0.9
β_2	0.999

Tab. 1: Adam parameters

Algorithm 4 Adam Optimiser

Require: $f()$, Θ, α, ϵ, β_1, β_2

1: $\mathbf{s} = 0, \mathbf{r} = 0, t = 0$
2: **while** stopping criterion is not met **do**
3: Sample minibatch of m training samples $(\mathbf{x}^i, y^i)$
4: $\mathbf{g} \leftarrow \frac{1}{m} \frac{\partial}{\partial \Theta} \sum_i L(f(\mathbf{x}^i; \Theta), y^i)$
5: $t = t + 1$
6: $\mathbf{s} \leftarrow \beta_1 \mathbf{s} + (1 - \beta_q)\mathbf{g}$
7: $\mathbf{r} \leftarrow \beta_2 \mathbf{r} + (1 - \beta_2)\mathbf{g}^2$
8: $\alpha_t \leftarrow \alpha \sqrt{1 - \beta_2^t}/(1 - \beta_1^t)$
9: $\Theta \leftarrow \Theta - \frac{\alpha_t}{\sqrt{\mathbf{r}} + \epsilon} \mathbf{s}$

the previous gradients. $\mathbf{r}$ is the second moment estimate. It is used to normalise the momentum with respect to its variance. Both are biased in the sense, that if we go through the algorithm for t steps, we have:

$$\mathbf{r}_t = (1 - \beta_2) \sum_{i=1}^{t} \beta_2^{t-i} \mathbf{g}_i^2. \tag{3.6}$$

where

$$(1 - \beta_2) \sum_{i=1}^{t} \beta_2^{t-i} = (1 - \beta_2) \frac{1 - \beta_2^t}{1 - \beta_2} = 1 - \beta_2^t \tag{3.7}$$

This means our expectation of $\mathbf{r}_t$ is biased:

$$\mathbb{E}[\mathbf{r}_t] = \mathbb{E}[\mathbf{g}^2](1 - \beta_2^t) \tag{3.8}$$

Thus we see the reason why α_t gets a factor of $\sqrt{1 - \beta_2^t}/(1 - \beta_1^t)$ in line eight.

3.3 Learning Rate Schedules

In our experiments we will work a lot with different types of learning rate scheduling. This means over the training procedure we are adapting the learning rate α. The goal is to give the optimiser the optimal dynamic to find the global minimum. In general this means a higher learning rate at the beginning a decreasing learning rate in the end, not to make too big steps out of the loss minimum.
There are many different ways to schedule learning rate, here we will get to know two different schedules, we implemented for the results.

Exponential Decay

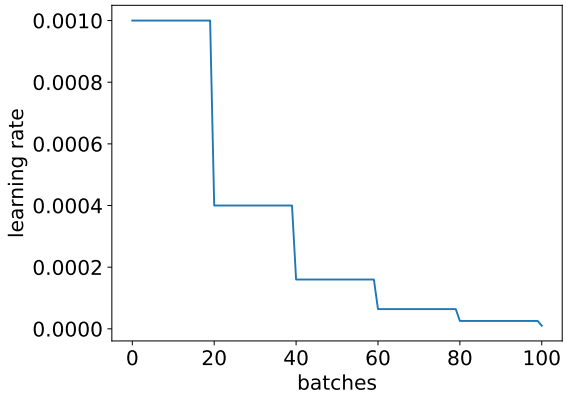


Fig. 3: Example of STEPLR for stepsize=20 and decay=0.4

In Figure 3 we see an example of the learning rate scheduler with exponential decay (STEPLR) applied on a training of 100 batches. The initial learning rate is set to 0.001 and every 20 batches it gets decreased by 0.4. Usually this is a very robust schedule, though it might be, that the periods with higher learning rate are too short.

1cycle

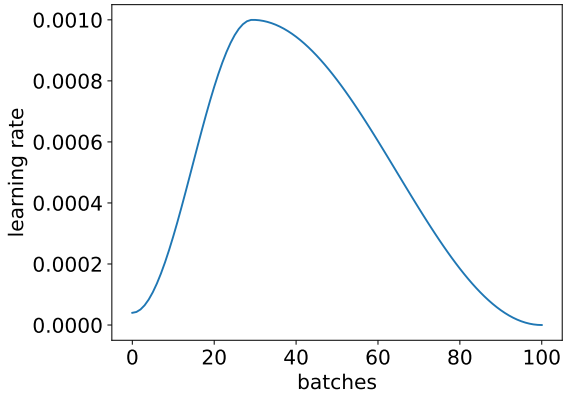


Fig. 4: Example of One cycle learning rate scheduler for maximum learning rate = 0.001

The idea of ensuring a stable beginning in the training lead to the *1cycle learning rate policy* [19]. As visible in Figure 4 the scheduler increases the learning rate in the first third and then switches smoothly to the decrease until a minimum is reached. This can boost the training strongly.

3.4 Generative Models

In this thesis we will perform *unsupervised learning*. In contrast to *supervised learning* we do not give the network a label y_i to the corresponding data point x_i. A special type of unsupervised learning is *reinforcement learning*, where the network still gets a reward r_i for every decision it makes. In our case the network should learn structures and correlations in the data itself instead.
For generative models the goal is to approximate the true probability density $P(x)$ by a model with parameters Θ. Usually in the end the model shall be able to sample from $P(x)$, but sometimes even an explicit calculation of $P(x)$ is possible [14].
As in most of the cases in supervised learning, the principle of designing a loss for generative models is usually the maximum likelihood principle, introduced by R. A. Fisher. It says, if we have a data-generating distribution p_{data} and training data $\mathbb{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, then our model parameters Θ should maximise the likelihood of this data:

$$\Theta_{model} = \arg\max_{\Theta} p_{model}(\mathbb{X}; \Theta) \tag{3.9}$$

4 Invertible Neural Networks

One approach for maximising the likelihood in Equation 3.9 is to assume that the data x depends on a latent variable z which is the random variable of a much more simple distribution like uniform or Gaussian. The idea to make the mapping m_Θ between any variables x and z invertible, lead to the Invertible Neural Network (INN) [20], also called Normalising Flow [5][21]. Since we are going to use it as a model to generate events, we will call the space of z the latent space $\mathcal{Z}$ and the other side the *pre-event space* $\mathcal{X}$.

$$m_\Theta : \mathcal{Z} \to \mathcal{X}, z \to x \tag{4.1}$$
$$m_\Theta^{-1} : \mathcal{X} \to \mathcal{Z}, x \to z \tag{4.2}$$

From the invertibility follow some restrictions for the network. The mapping has to be constructed as homeomorphism. This firstly means, the dimension of $\mathcal{Z}$ and $\mathcal{X}$ have to be the same. Secondly this means the transformation has to be bijective. Though we can build our bijective map out of many bijective transformations. For INNs we call these single transformations *coupling blocks*. One coupling block can be viewed in Figure 5. Unfortunately the ReLU activation function is not invertible and in general inverting a Dense layer is - if at all possible - computationally expensive. At the same time we do not want to forgo the power of dense layers described in section 3. The solution is to switch to an easily invertible transformation, where the Jacobian can be calculated very fast, but at the same time using classical Dense networks for determining the transformations parameters.

Therefore we have to split the input variable into two halfs, e.g. the resulting variables have only half of the dimension of z. Then we use the upper one as input for the dense layer, to transform the lower half by h_Θ with the calculated parameters Θ.

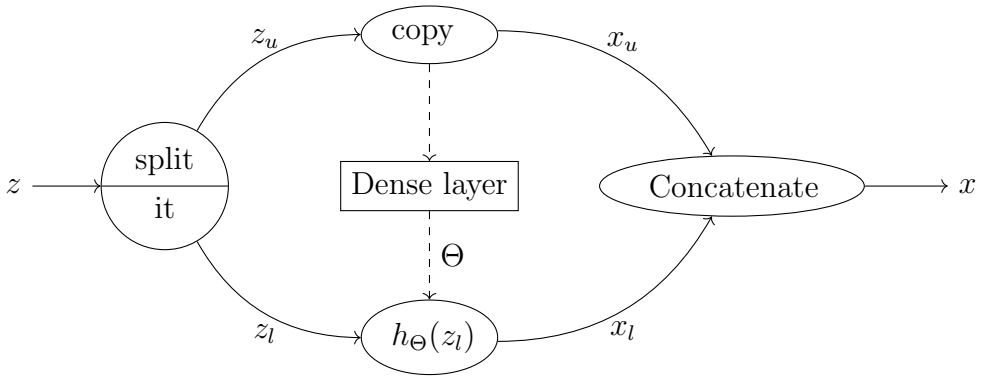


Fig. 5: Scheme of a coupling block in an INN

The map from z to x can now easily get inverted by taking the inverse of h_Θ. To ensure that every dimension of z gets transformed at least once, we need at least two blocks and we have to permute the dimensions in between. This can be done by (random) permutation matrices or by continuous rotation matrices. For the latter ones we have to ensure that our latent space has the right rotation symmetry.
To implement our INN, we employ the FrEIA framework by the Visual Learning Lab Heidelberg [20].

Rational Quadratic Spline Transformation

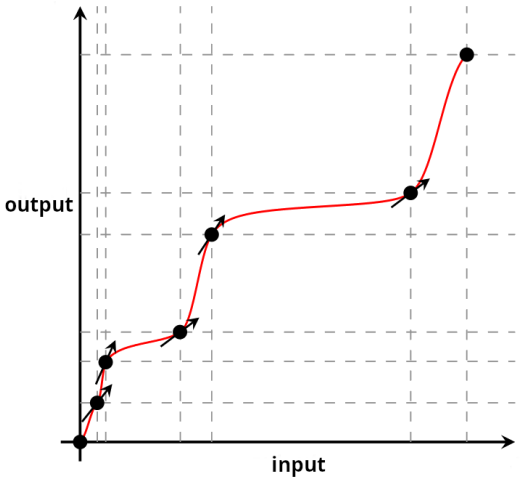


Fig. 6: Example of rational quadratic spline fitting, taken from [22]

One type of invertible transformations are polynomial splines. These are just piecewise polynomial functions with differentiable transitions between the pieces.

They monotonically map $[0,1] \to [0,1]$. An alteration of polynomial splines are rational quadratic splines, first introduced by Gregory and Delbourgo [23]. This means our function is a ratio of two quadratic functions. The parameters of this spline are the positions (x, y) of the *knots* the corresponding polynomial is linked to and the boundary derivatives [24]. The first knot is fixed at (0,0) and the last one at (1,1). This way of fitting is illustrated in Figure 6. While this transformation is powerful, it is also computationally cheap to invert and to calculate its Jacobian of. We use rational quadratic splines for all our results obtained with INNs.

5 Monte Carlo Integration with Importance Sampling

Monte Carlo methods have the simple principle of sampling random numbers and they can be extremely powerful. Though when emphasising the precision of the result, in some cases using Monte Carlo methods can lead to excessive computational power consumption. In our case we consider the Monte Carlo integration. Here we approximate the integral I of function f by evaluating f at N uniform samples from $U_N = \{s_i \,|\, s_i \in \Omega, i \in \{1, ..., N\}\}$ in the integration space Ω with volume V:

$$I \approx E_N := V \frac{1}{N} \sum_{i=1}^{N} f(s_i) = V \langle f \rangle_{U_N} \tag{5.1}$$

Thus the variance of the approximation goes with the variance of function f.

$$\mathrm{Var}(E_N)_{U_N} = V^2 \frac{\mathrm{Var}(f)_{U_N}}{N} = V^2 \frac{\langle f^2 \rangle_{U_N} - \langle f \rangle^2_{U_N}}{N} \tag{5.2}$$

This easily leads to crucial inefficiencies, when integrating complex functions in higher dimensions, since one has to compensate with the number of evaluations N. For us the samples s_i are individual *events* and so we call $f(s_i)$ the corresponding *event weight* w_i. In the end our integrator shall also be a phase space sampler. Thus, besides the precision of the integral, we also wish the events all to be unit-weighted. The most simple approach is to employ a hit-or-miss algorithm, converting a weighted set to an unit-weighted set of events. This *unweighting* goes with efficiency

$$\epsilon_{uw} := \frac{\langle w \rangle_{U_N}}{w_{max}}. \tag{5.3}$$

Now the variance of the estimate and the unweighting efficiency are two different metrics for the quality of our result. Typically they go with one another, but they do not have to lead to the same optimisation measures. Instead of the learning rate we will rather use the width of the weight distribution as measure of success.
Importance sampling [25] is one method to reduce or even eliminate the non-statistical error, by sampling from a different distribution: We increase the number of samples, where function values are big and thus contribute the most to the integral and we decrease it in regions with small values. Technically speaking, we would like to sample from a distribution $g(x)$, which has the shape of the integrand f itself:

$$E'_N := V \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{g(x_i)} = V \langle f/g \rangle_{G_N} \tag{5.4}$$

with G_N a set of N samples x_i from $g(x)$. From the variance

$$\mathrm{Var}(E'_N) = V^2 \frac{\mathrm{Var}(f/g)_{G_N}}{N} = V^2 \frac{\langle (f/g)^2 \rangle_{G_N} - \langle f/g \rangle^2_{G_N}}{N} \tag{5.5}$$

we now see that in the ideal case $f(x) = g(x)$ the variance would vanish.
One algorithm which uses this method is vegas.

5.1 vegas

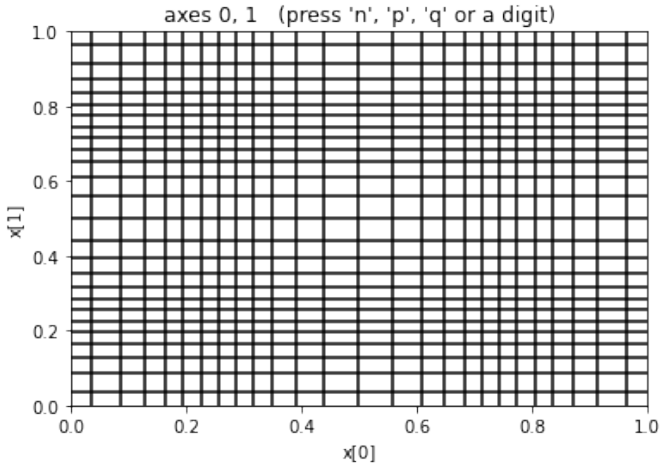


Fig. 7: Example of vegas grid after training on the integrand in Equation 5.6

vegas is an algorithm invented in 1978 by G. P. Lepage [1], to give better Monte Carlo estimates of integrals. The principle attempt is to flatten the integrand by automatic transformation to the integration volume. This is done by taking the volume as a multidimensional grid and than adapting the size of each cell inversely to the corresponding relative value of the integrand in this region. After this is done, a Monte Carlo estimate of the integral can be given with the transformed variables. Especially important for us is that the Jacobian of this transformation is again passed as weight corresponding to one sample/event.
The integrand does not have to be analytical or continuous, though - and here comes the main weakness of vegas - vegas does assume that the integrand is factorisable. Taking for example the double gaussian integrand

$$f(x,y) = e^{\frac{-(x-0.25)^2-(y-0.25)^2}{0.25}} + e^{\frac{-(x-0.75)^2-(y-0.75)^2}{0.25}} \tag{5.6}$$

we have two regions on the diagonal, being most relevant for the integral. What vegas is forced to do is adapting its grid as shown in Figure 7, e.g. it maps four instead of two gaussians in the integration space.

One among other features of interest for us is the adaptation to Monte Carlo errors. This means vegas does not have to sample in every grid the same number of samples. Instead it can redistribute the samples such that it samples most of the points in the regions, where its Monte Carlo estimate has the biggest errors [26].

5.2 INN Generating

From Equation 5.5 we can conclude, that our goal is to shape our distribution, we sample from, like the integrand. Transforming the latent space distribution $p(z)$ with our INN, we want the INN Jacobian $g(r)$ to have the shape of the integrand $f(r)$:

$$f(r) \sim p_{true}(r) \underset{\text{training}}{\leftarrow} p_{model}(r) = p(z) \det\left(\frac{\partial z}{\partial r}\right) = p(z) \det\left(\frac{\partial m^{-1}(r)}{\partial r}\right) =: g(r) \tag{5.7}$$

where we used the change of variables formula.
As objective for the training we thus need some kind of a metric for the discrepancy of the Jacobian and the integrand. The divergence gives us the *distance* between two distributions. Though there are many different kinds of formulas for divergence. Here we will refer for example to the *Pearson* χ^2 *divergence*:

$$D_{\chi^2} = \int \frac{(p(r) - q(r))^2}{q(r)} dr \tag{5.8}$$

or the *Kullback-Leibler divergence*:

$$D_{KL} = \int p(r) \log\left(\frac{p(r)}{q(r)}\right) dr \tag{5.9}$$

But for all our results we will stick to the so called *exponential divergence*:

$$D_e = \int p(r) \log\left(\frac{p(r)}{q(r)}\right)^2 dr \tag{5.10}$$

Introduced was this type of training first by Bothmann, Janßen, Knobbe, Schmale and Schumann [27], but then also by Gao, Isaacson and Krause [2].
For our loss we have to write this integral as sum over our samples. Since our samples generally describe a non-uniform distribution, we have to divide by the density of the distribution at these points:

$$D_e = \int \frac{g(r)}{g(r)} f(r) \log\left(\frac{f(r)}{g(r)}\right)^2 dr = \mathbb{E}_{r\sim g(r)} \left(\frac{f(r)}{g(r)} \log\left(\frac{f(r)}{g(r)}\right)^2\right) \tag{5.11}$$

$$= \sum_{r\sim g(r)} \frac{f(r)}{g(r)} \log\left(\frac{f(r)}{g(r)}\right)^2 \tag{5.12}$$

Though by mistake we forgot the factor of $1/g(r)$ in our implementation and noticed it only when working with I-flow in section 9. This makes the loss the same after a flat initialisation, but from the point on, when the network learned the distribution roughly this makes a bigger difference.

5.3 INN Recycling

Another approach for a loss function is as usually based on the Maximum Likelihood principle in Equation 3.9. In this case we just have to maximise $|g(r)|$, derived from

Equation 5.7. As prior $p(z)$ the common choice is a normal distribution $\mathcal{N}(0, \mathbb{I})$. Since we take the logarithm for our loss, we can neglect the normalisation of the normal distribution. Thus our loss is shifted by a constant and can also be negative. We call this training *recycling* since we can reuse the sampled data, generated by the training above. Concretely we have to correct the loss of r, to train with the target distribution $p_{true}(r)$. This is done by weighting the loss with the normalised weights $\hat{w_{tr}}(r) = \frac{w_{tr}(r)}{\langle w_{tr} \rangle_r}$, with

$$w_{tr}(r) p_{model}(r) \stackrel{!}{=} p_{true}(r) \sim f(r) \;\Rightarrow\; w_{tr}(r) := \frac{f(r)}{p_{model}(r)}. \tag{5.13}$$

Taking all together our loss L for a Gaussian latent space is then defined as

$$L = \text{mean}\left(\hat{w_{tr}}(r)\frac{z(r)^2}{2}\right) - \text{mean}\left(\frac{\hat{w_{tr}}(r)\log(g(r))}{n_{dim}}\right) \tag{5.14}$$

We have to divide the second term additionally by the number of dimensions n_dim, since the first term takes the mean over $z(r)$, which is multidimensional, while $g(r)$ is only one-dimensional.
In the case of integrating a matrix element this type of training can be especially useful, since evaluating it, often is the bottle-neck of the integrator. But for recycling we do not need to evaluate it again, instead we use the network in the opposite direction.

6 Integration pipeline for the cross section

Bringing all together we can build up a pipeline of all modules to obtain an estimate of the cross section for our process - once with vegas and once with the INN.

vegas pipeline

The pipeline with vegas is shown in Figure 8. vegas has its own random number generator and thus needs no input. Since RAMBO takes only variables between 0 and 1 vegas returns variables in a d-dimensional hypercube (r-space). With n the number of final states we have $d = 3n - 2$, since we have $3n - 2$ degrees of freedom including the momentum fractions. RAMBO transforms this into four-momenta p in the *phase space*. Its weights are directly transferred into the evaluation of the integrand, which is the differential cross section. Thus we define the integration volume to be the r-space, e.g. the integrand is defined in this space.
For the evaluation we take Equation 2.20 and for dX' we take the transformed phase space volume, given in the RAMBO weights w^{rambo}. The result gets returned and vegas uses it to transform its integration space. After every training iteration vegas also returns the integral estimate by summing over its evaluations. Here it also has to divide by the Jacobian of its transformation, which is hidden in the vegas weights w^{vegas}.

INN pipeline

With the INN replacing vegas we have in principle exactly the same architecture, showed in Figure 9. Again the integrand $f(r)$ is defined in the r-space and in the loss it is compared with $g(r)$.
On the other side there are different implementations possible: In some experiments we just used a flat latent space and thus $m(z)$ is defined as the pure INN. In most of the experiments we used a Gaussian latent space like in the scheme. Usually it is hard for a network to learn to set borders in the output like the r-space. Thus in this cases we added in the INN the transformation which turns a Gaussian into a uniform distribution - namely the error function. It is defined as

$$\text{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt \tag{6.1}$$

and maps $\mathbb{R}$ to $(-1, 1)$. So we only have to scale and shift the output to $(0, 1)$.

Plotting

For the plotting we wish to see the pure samples expressing what the network already learned. This is simply done by sampling after training and plotting a histogram. Additionally we plot the true data distribution by weighting them with w_{tr} from Equation 5.13. For vegas this is $\frac{f(r)}{w^{vegas}(r)}$ and for the INN $\frac{f(r)}{g(r)}$.
Apart from that we can make a check, if nothing goes wrong by weighting the samples such that the uniform distribution results.

$$w_{\mathcal{U}}(r) p_{model}(r) \stackrel{!}{=} \mathcal{U}_{[0,1]} \;\Rightarrow\; w_{\mathcal{U}}(r) = \frac{1}{p_{model}(r)} \tag{6.2}$$

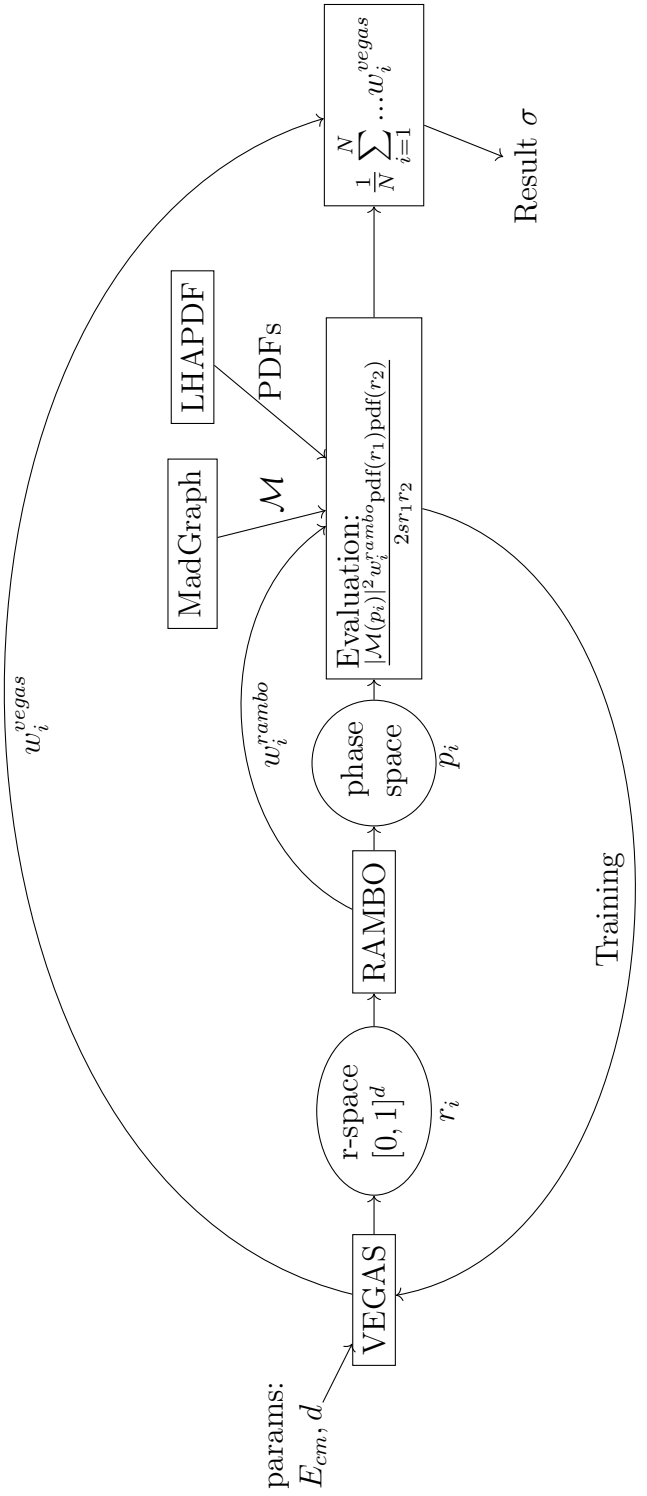


Fig. 8: Scheme of integration pipeline with *vegas*

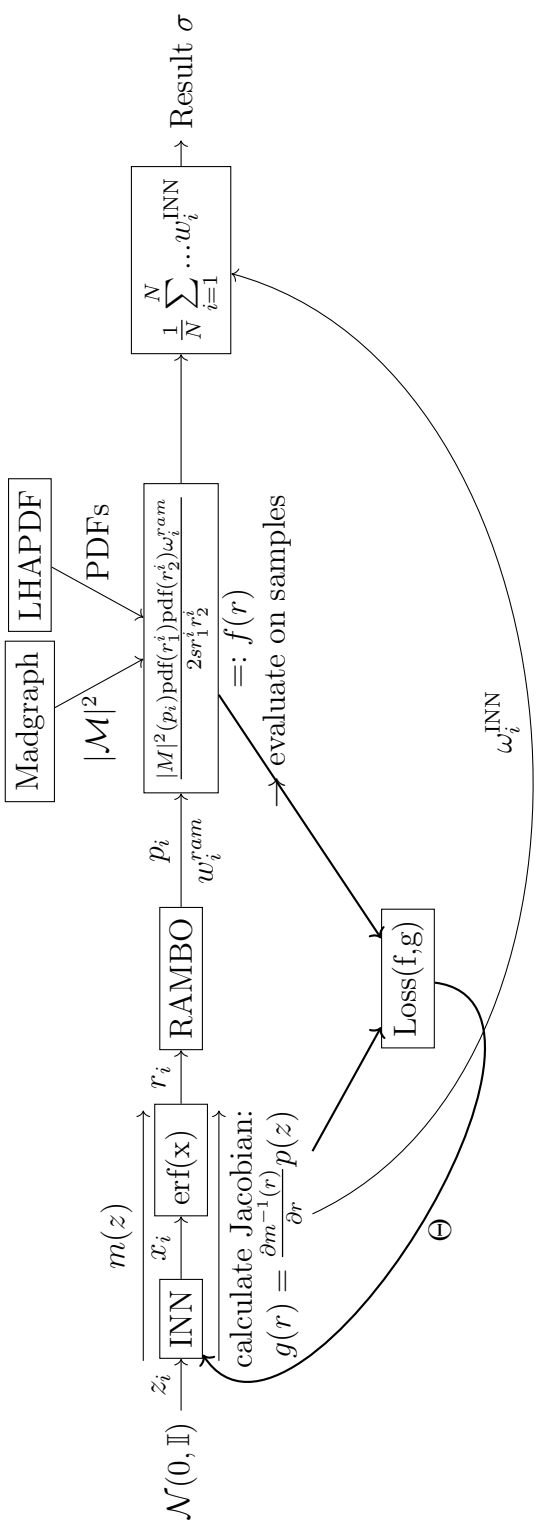


Fig. 9: Scheme of integration pipeline with INN

7 vegas benchmark

After finding a way of working with vegas we tried to estimate the cross section of our process $gg \to ggg$.

7.1 gg → ggg

For the integration we choose physical parameters, listed in the left part and vegas arguments, listed in the right part of Table 2.

physical param	chosen value
$\sqrt{s}$	13TeV
p_T-cut	20GeV
ΔR-cut	0.4GeV
pdf-set	MSTW2008lo68cl_nf3

vegas param	chosen value
iterations	10
evaluations	100000
sampling	110000

Tab. 2: Parameters for vegas baseline

Plots

After sampling with the trained vegas model, we can plot different observables. A list of all with the formula to calculate it from the four vectors is given in Table 3. We list here also how many plots for the three final gluons we can make all together.
In blue we plot the raw number of phase space samples (without any weights).

observable	formula	number
p_z	-	3
p_T	$\sqrt{(p_x)^2 + (p_y)^2}$	3
η	$\frac{1}{2}\log\left(\frac{\lvert\mathbf{p}\rvert+p_z}{\lvert\mathbf{p}\rvert-p_z}\right)$	3
ϕ	$\tan^{-1}\left(\frac{p_x}{p_y}\right)$	3
ΔR	$\sqrt{\Delta\phi + \Delta\eta}$	3
E_{lab}	$\sum_i (p_T^i \cdot \cosh(\eta^i))$	1
$\sqrt{\hat{s}}$	$\sqrt{E_{\text{lab}}^2 - (\sum_i p_z^i)^2}$	1
Momentum fraction	$\frac{E_{\text{lab}} \pm \sum_i p_z^i}{\sqrt{s}}$	2

Tab. 3: List of observables we can plot

The samples weighted with $w_{\mathcal{U}}$ are shown in red. A target distribution we built from the madgraph framework is plotted in green. This can be considered as true data, since the final sampling distribution gets unweighted, e.g. the distribution is corrected.
In the plots for the MadGraph samples and the raw phasespace samples we implement $\frac{1}{\sqrt{N}}$ errorbars (if N is the number of samples in one bin). For the weighted

distribution we can not argue in the same way, since there can be weight-outliers. Therefore we calculate the error for each bin ϵ_{bin} independently by:

$$\epsilon_{\text{bin}} = \frac{N}{\sqrt{\frac{\sum_w^N w}{\max(w)}}}$$

With this implementation, if all weights in one bin are equal, ϵ_{bin} again reduces to $\frac{1}{\sqrt{N}}$; but in the limit, when the data in one bin is getting dominated by a single high-weighted event, ϵ_{bin} is going to N.
Additionally we add two subplots, where we plot the deviations of the pure samples and the Madgraph data from our true distribution, obtained by weighting the pure samples. In the upper subplot we plot the ration of true over test and in the lower one we plot the deviation divided by the true values.

Results

For first we show here only the p_T-distribution for the first outgoing gluon.

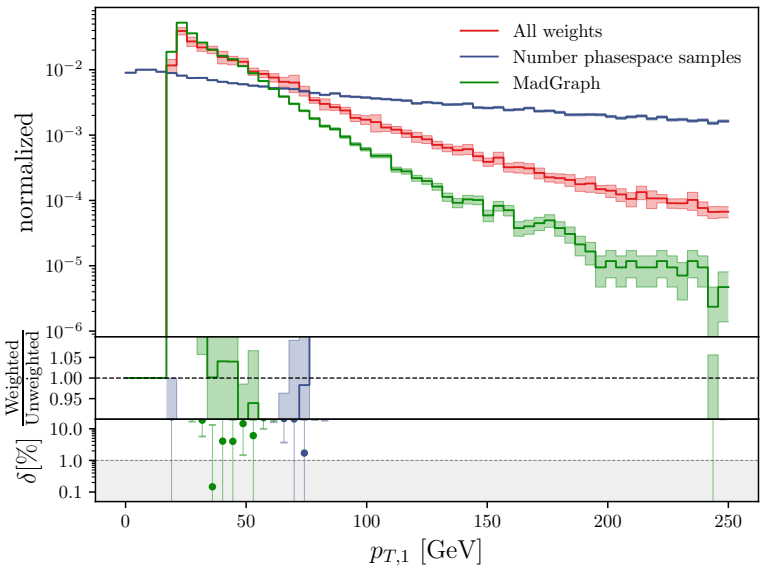


Fig. 10: p_T distribution of the first gluon in the final state with parameters as in Table 2

Viewing only the p_T-distribution of the first particle in Figure 10 we already see that vegas did not learn very much in the training: The blue line is not close to the true distribution, created by the weights shown in the red line. Also the cut is not yet visible there. But most importantly it makes clear that our configuration is not the same as the Madgraph one, because the true distribution created by our weights deviates from the Madgraph distribution.
To find the difference we simplify our integrand: In our and in Madgraphs fortran code, we set parts of it to one, while we either plot

- samples from a trained vegas model as before, or
- samples from a uniform hypercube (r-space) fed into RAMBO.

$\mathcal{M}$=PDF=1

We set the PDFs and the matrix element to 1 , leaving only $d\sigma \propto 1/(2r_1r_2s)$ as well as the RAMBO weights. Now, sampling uniformly, the observable distributions are in good accordance (Figure 11).

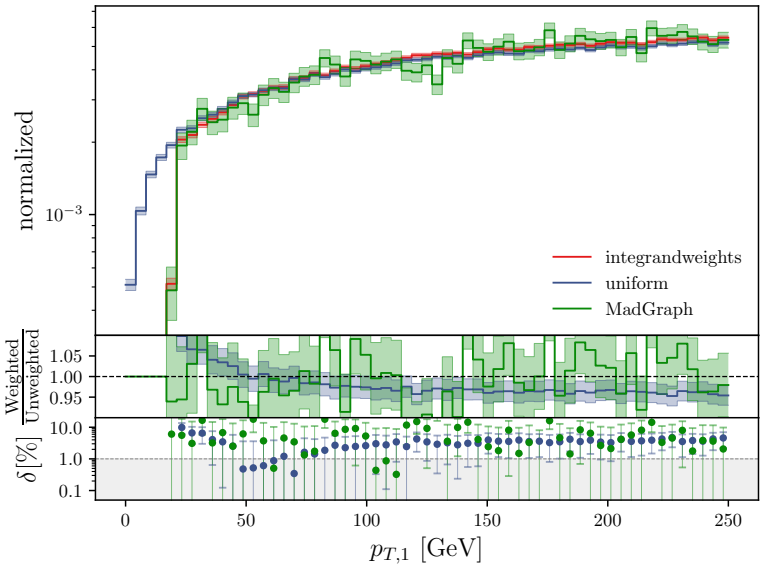


Fig. 11: p_T-distribution, sampled uniformly and $\mathcal{M}$=PDF=1

PDF=1

We again insert the matrix element. Sampling from vegas, our weighted distribution is very close to the Madgraph distribution (Figure 12).

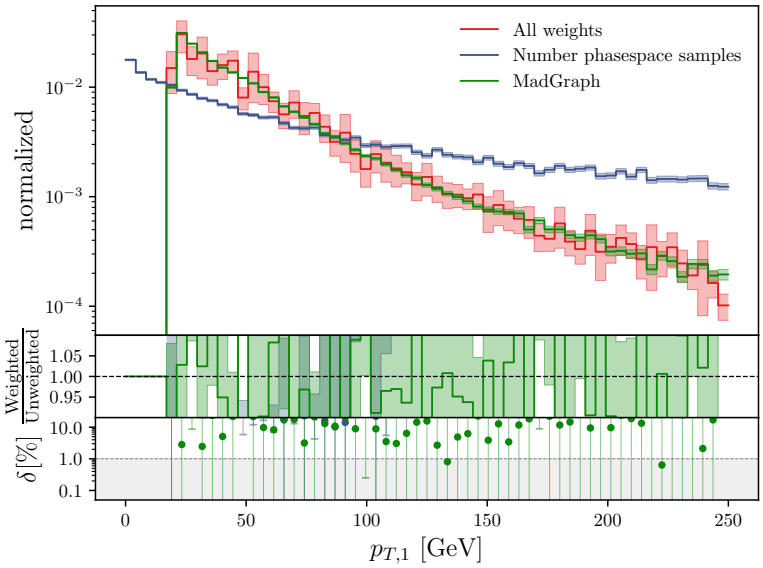


Fig. 12: p_T-distribution, sampled with vegas and PDF=1

$\mathcal{M}$=1

Now we set the matrix element again to one and instead insert the full PDFs. The problem arises again, for both, sampling uniformly as well as with trained vegas (Figure 13).

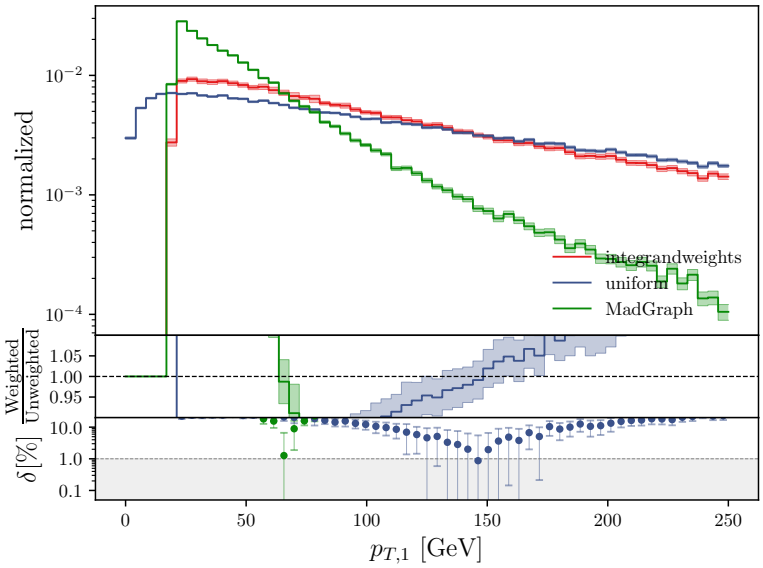


Fig. 13: p_T-distribution, sampled with vegas and $\mathcal{M}$=1

Thus the problem lies in the PDFs and we tried some changes: First we used the exact same pdf set as Madgraph. Though the behaviour does not change. Secondly we also set the energy scale of the PDFs in Madgraph and in our code to the mass of the Z-Boson. Again, the deviation remained.

Gluon PDF approximation

On our supervisors advice, we decided to omit the LHAPDF package and to use instead an approximation for the gluon pdf function:

$$\mathrm{pdf}_{\mathrm{gluon}}(x) \approx \frac{1}{x^2}$$

Since we just want to create a baseline for our INN, it is no loss to use only an approximation.

Finally the red and the green curve are in good accordance for every observable. Additionally to the p_T-distribution in Figure 14 we show here also the η-distribution in Figure 15 and the ΔR-distribution in Figure 16. In all of these we see that the true distribution under the label *Physics?* is the most noisy one. These uncertainties have to be caused by the weights w_{tr}. This makes it hard to compare it with the Madgraph data more accurately.
The blue curve shows that vegas still did not learn out. The final estimates for the integral are in the same order of magnitude, though the difference is much bigger than the error of both of them: For madgraph we get $(15.296 \pm 0.015)\mathrm{GeV}^{-2}$ and

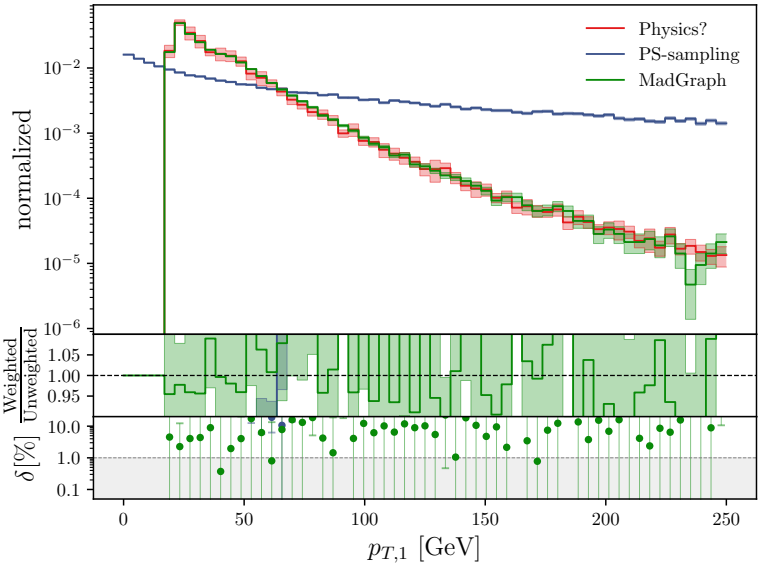


Fig. 14: p_T-distribution, sampled with vegas and the $1/x^2$ pdf approximation

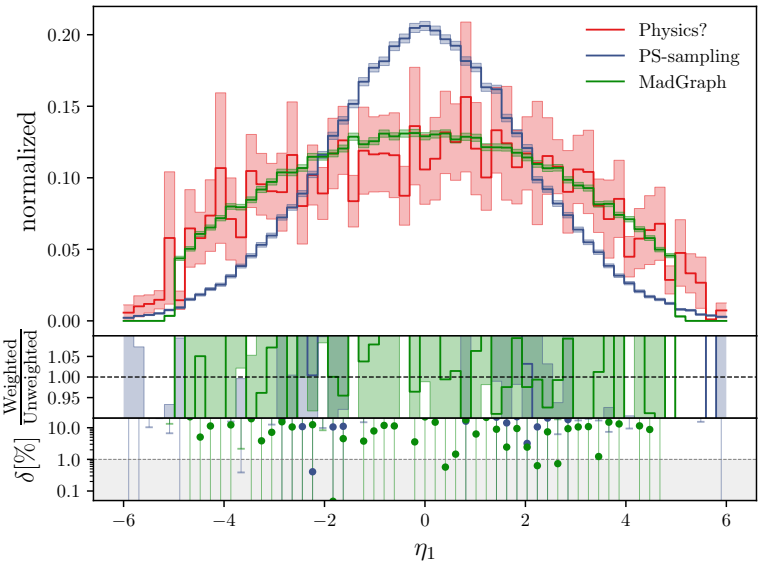


Fig. 15: η-distribution, sampled with vegas and the $1/x^2$ pdf approximation

for our vegas architecture $(12.78 \pm 0.16)\text{GeV}^{-2}$. This deviation is hardly explainable by us. Though one explanation could be that vegas underestimated the error dramatically, because it did not *see* some narrow peaks in the integrand and thus could not quantify the true variance.

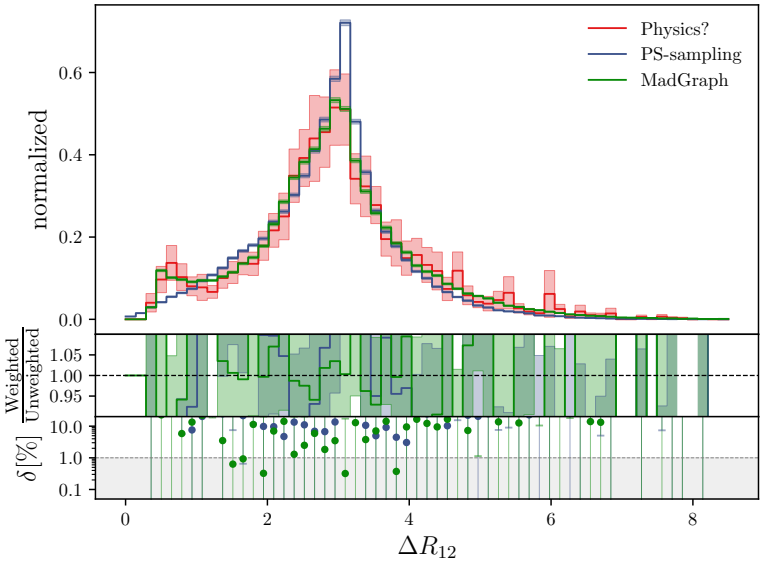
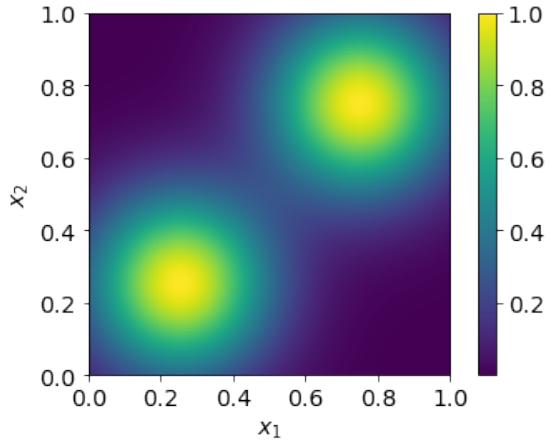


Fig. 16: ΔR^{12}-distribution, sampled with vegas and the $1/x^2$ pdf approximation

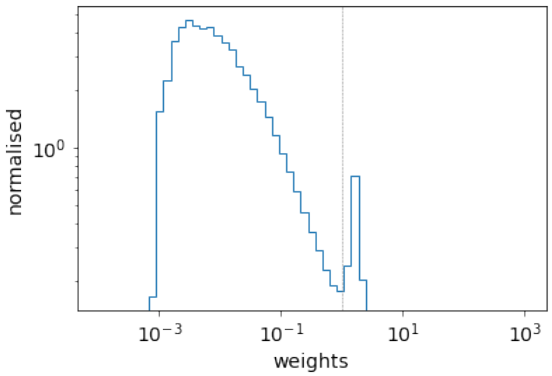
7.2 Toy example

Double Gaussian

With our INN we will start with the toy function, already described in Equation 5.6. We can easily extend this function into higher dimensions, but let us first stick to two dimensions. The function is plotted in Figure 17a. As described in subsection 5.1,



(a) Scattering plot of target function



(b) Normalised vegas weight distribution after integrating the double Gaussian toy function

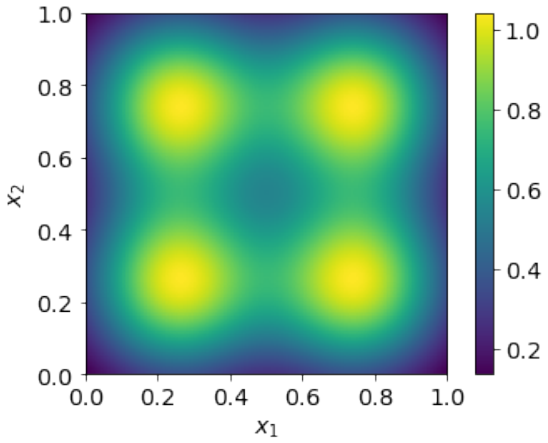
Fig. 17: Toy function from Equation 5.6 and trained vegas weights

vegas assumes the integrand to be factorisable, what the double Gaussian is not. When we integrate it, vegas needs for 10 iterations, each of $2 \cdot 10^5$ evaluations, round about 47 seconds on a common CPU. The result of the last estimate is 0.3328 ± 0.0006. A suitable measure of precision is the relative error of the estimate. This is the standard deviation divided by the true integral value. Since the error goes with the inverse root of the number of evaluations N_{eval} (see Equation 5.2),

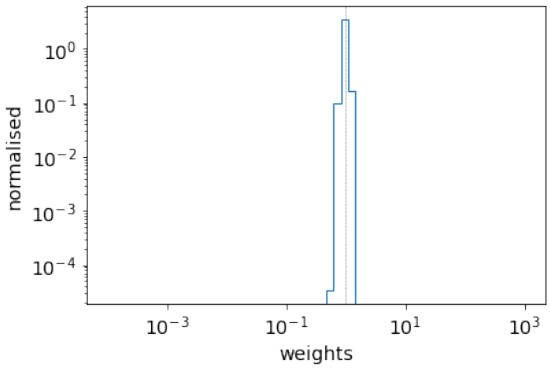
we additionally multiply it with $\sqrt{N_{eval}}$ and call this value ΔI. For this run we get $\Delta I = 0.74$.
In Figure 17b of the normalised weights we can recognise the difficulties, vegas has with these kind of functions. The range of weights goes over three magnitudes, because the wrongly modelled Gaussians have to get down-weighted.

Four Gaussian

To see, if we can beat vegas also for factorisable functions we integrate a toy function with four Gaussians, shown in Figure 18a. Again we train 10 iterations, each of $2{\cdot}10^5$ evaluations. vegas obviously does a lot better this time. Though it needs now 122



(a) Scattering plot of target function



(b) Normalised vegas weight distribution after integrating the four Gaussian toy function

Fig. 18: Toy function with four Gaussians and trained vegas weights

seconds for the same procedure, the normalised weight distribution in Figure 18b shows only a small width. The last integral estimate is 0.66666 ± 0.00011 which makes a ΔI of 0.074.

8 INN toy integration

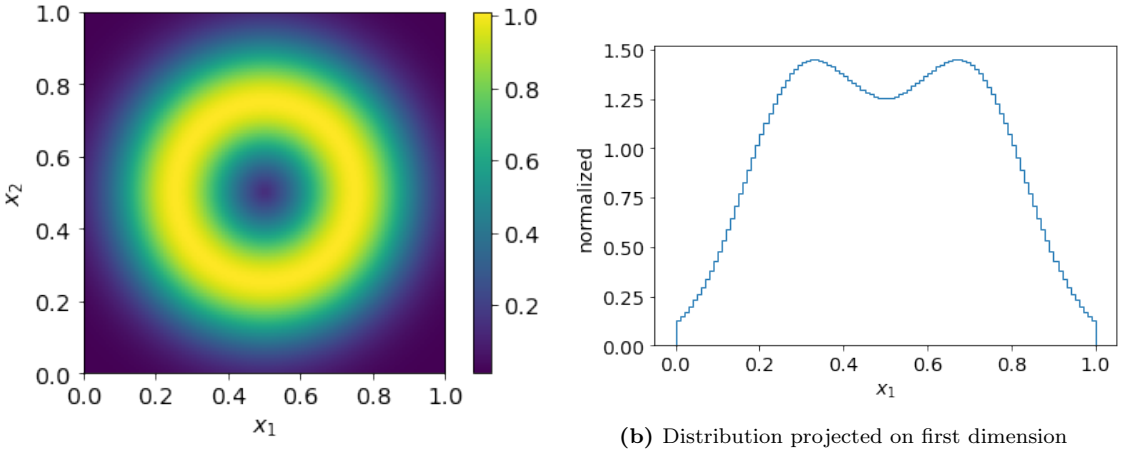
Training the INN can be done in the two ways described in subsection 5.2 and 5.3. As with vegas we now reduce our pipeline by taking the double Gaussian toy function as integrand.
First we want to learn, how generating and recycling work for their own separately.

8.1 Getting to know INN Generating

Gaussian Ring

For learning the difficulties of the INN training we implement a second toy function: A Gaussian ring, shown in Figure 19.

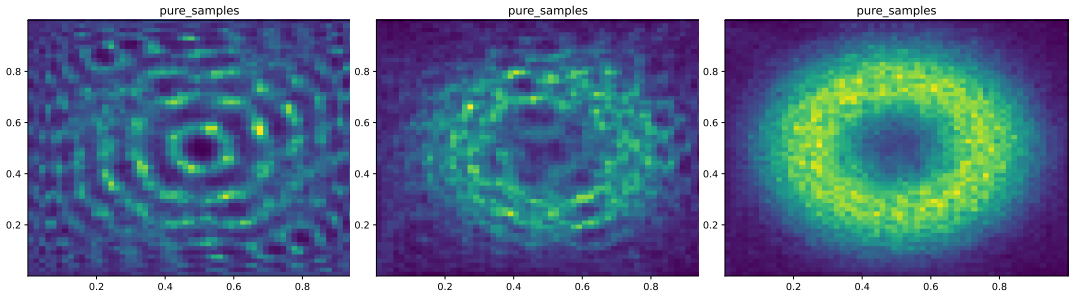
We try first what happens, if we let train the network by generating for a very long time. Therefore we use the 1cycle policy as introduced in section 3.3. Hereby we take 10^{-6} as initial learning rate and increase it in the first third to 10^{-5}. After



(a) Scattering plot

(b) Distribution projected on first dimension

Fig. 19: Toy function of Gaussian ring

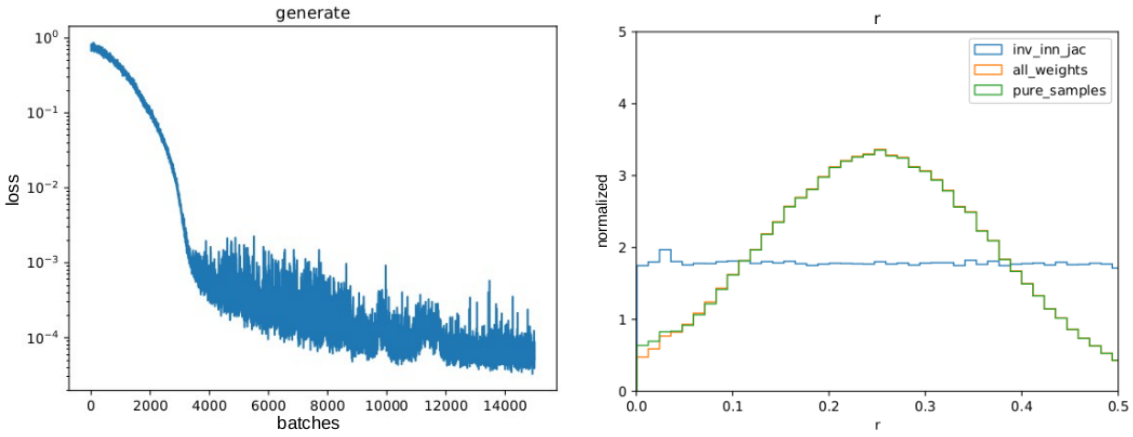


(a) After 1000 batches **(b)** After 2000 batches **(c)** After 3000 batches

Fig. 20: Generating with 1cycle policy on Gaussian ring - beginning

every 1000 batches, each of them of size 1024, we plot the intermediate results. In Figure 20 we can nicely observe the distribution creating a ring.

After the full training we project the samples on the radius (Figure 21b) and see the pure samples in green very close to the true distribution in red. In Figure 21a we see the loss decreasing very fast in the first third and then at some point becoming very noisy. Though this does not seem to damage the results.



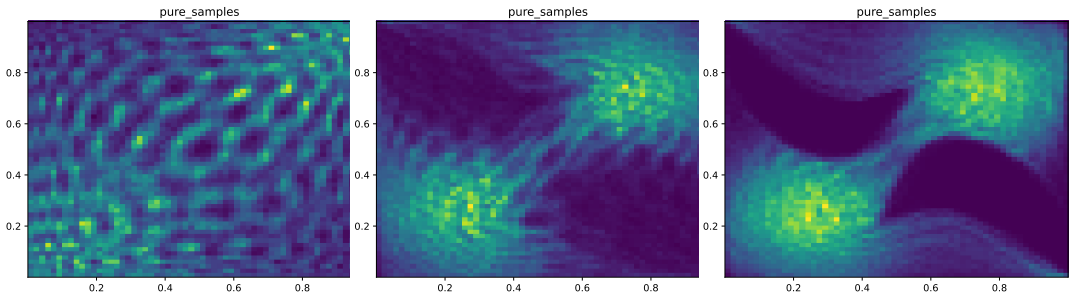
(a) Loss curve for 15000 batches of generating

(b) Final result radius projection

Fig. 21: Generating with 1cycle policy on Gaussian ring - final results

Double Gaussian

We do the same thing with the double Gaussian toy function and observe a different behaviour in Figure 22. After 900 batches the network starts to evolve edges in the distribution, concreting over the training. Analysing the inverse Jacobian after 1200 batches in Figure 23a we see the network does not sample in some regions with low integrand at all. In these regions the Jacobian becomes peaky and starts dominate also the true distribution (orange line in Figure 23b).

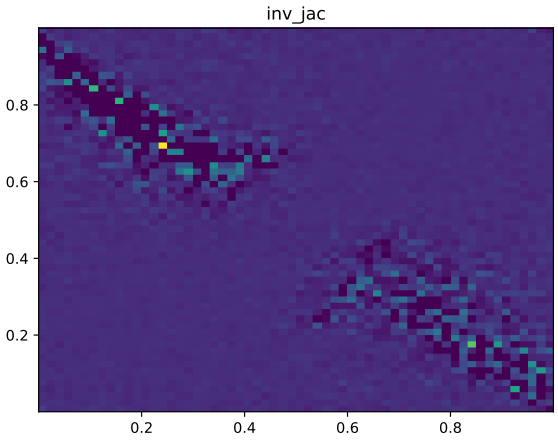


(a) After 300 batches

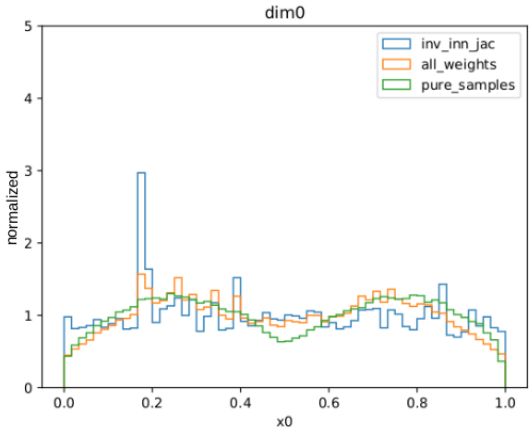
(b) After 900 batches

(c) After 1500 batches

Fig. 22: Generating with 1cycle policy on double Gaussian - beginning



(a) Samples weighted with inverse Jacobian after 1200 batches

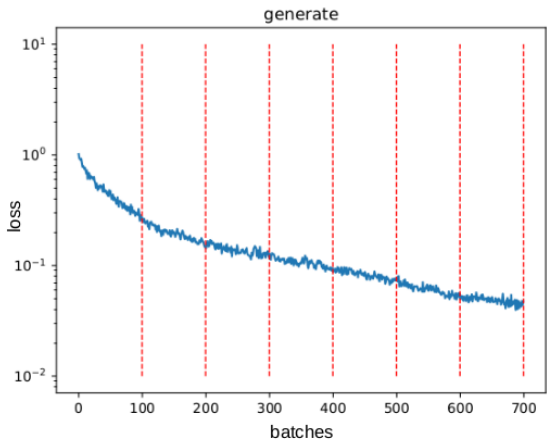


(b) Projected samples with weights after 1500 batches

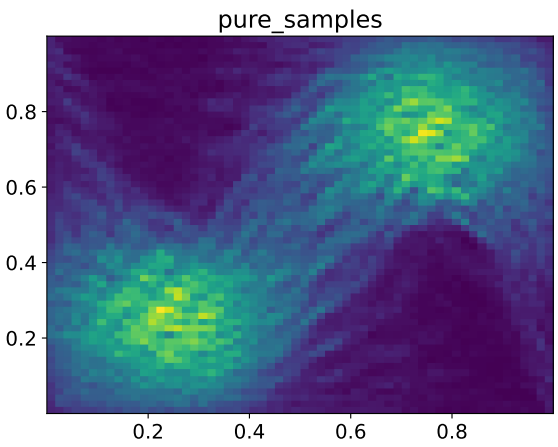
Fig. 23: Generating with 1cycle policy on double Gaussian - unstable training

Baseline

Since we want to avoid unstable training as above, we now set the learning rate to constantly 10^{-5}. To set a baseline we train for 700 batches, red lines are just marking every new hundred batches.



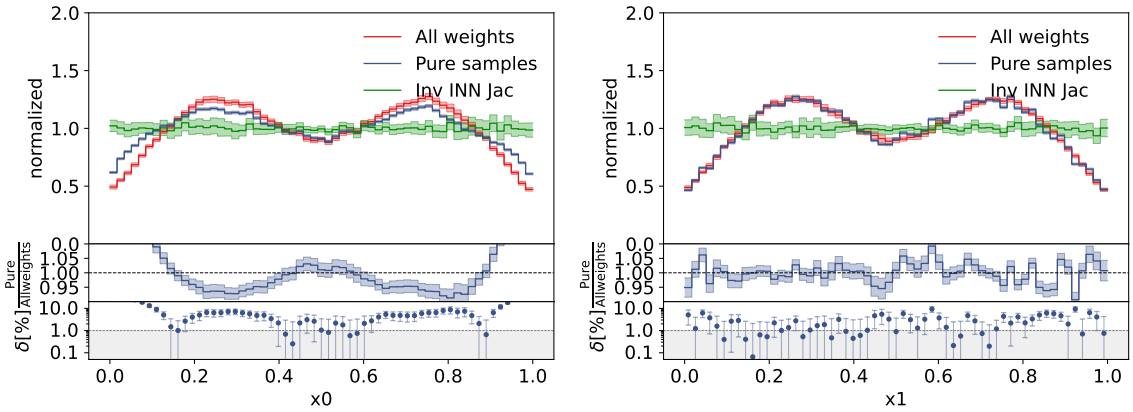
(a) Loss curve



(b) Sample distribution

Fig. 24: Results of generating 700 batches

The results in Figure 24 do not look satisfying since the network did not train for long enough. But because vegas is very fast, we do not want our network to train for arbitrarily long time. To have a more quantitative baseline, we insert the errors in the projections in Figure 25 as in subsection 7.1. Additionally we can take a look on the distribution of the weights in Figure 26. The different lines represent the intermediate states after every 100 batches. What we see is, that the network is improving only slowly.
After every period (red marker) we also calculate the integral estimate with roughly 10^5 evaluations. The last estimate for the integral of this run has a relative error of



(a) Projection on 1st dimension

(b) Projection on 2st dimension

Fig. 25: Projections of sample distribution

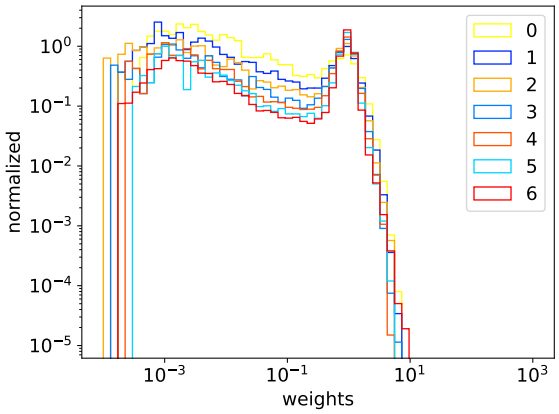


Fig. 26: Normalised weight distribution after every 100 batches. The labels numerate these intermediate states (0 means after the first 100 batches).

$11 \cdot 10^{-4}$ and thus a ΔI of 0.36.

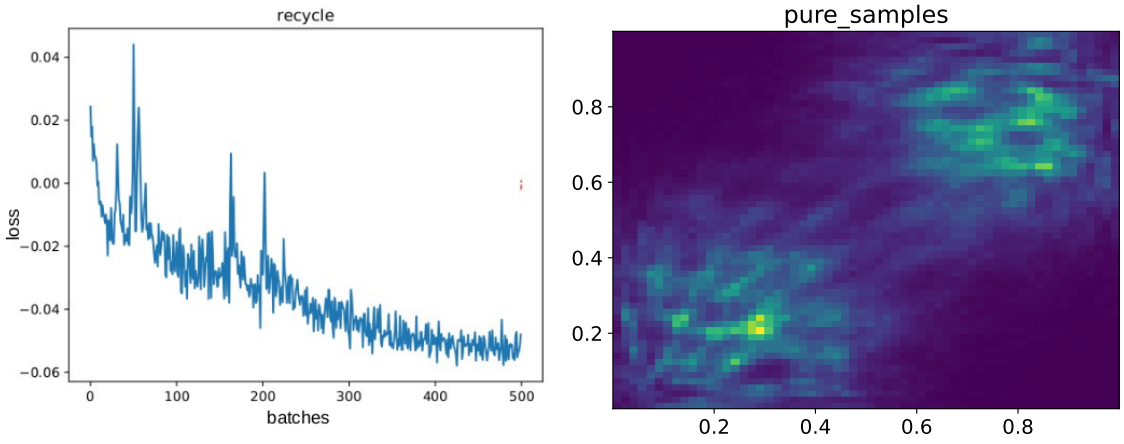
8.2 Getting to know INN Recycling

To train with recycling, there are several ways to obtain training data. What we do, is to use the unlearned model to sample uniformly and than multiply with the integrand as weights as described in subsection 5.2.

Overfitting

First we try to train on only five batches of size 1024 a 100 times (100 *epochs*). In Figure 27a we see the loss dropping quite low, though the final distribution in Figure 27b looks extremely blurred.

Most of the runs with more training data did not show such a low loss. This is why we assume, the model tending to overfit, when training on a data set with less than round about 100 batches. I. e. our model sees too little data, to be able to generalize in a appropriate way.

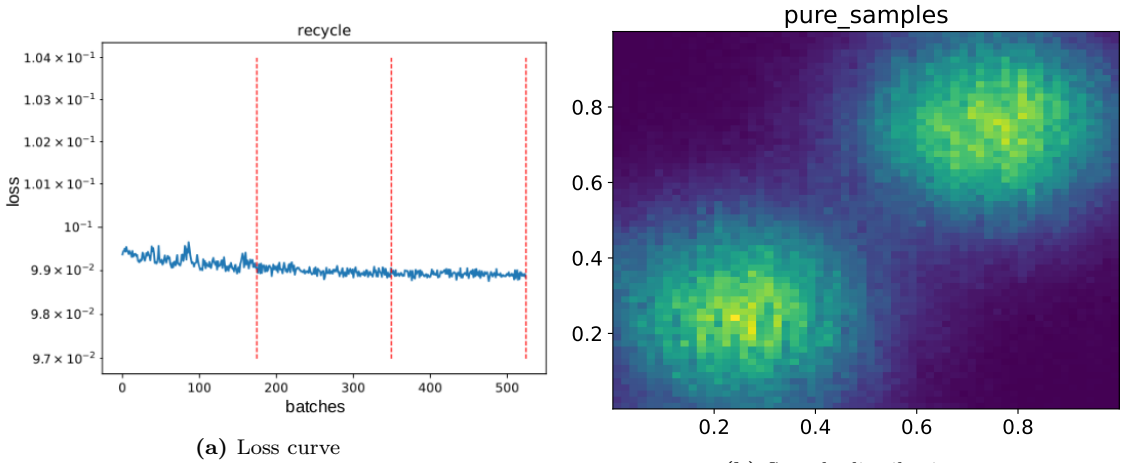


(a) Loss curve **(b)** Sample distribution

Fig. 27: Results of recycling for 100 epochs on 5 batches

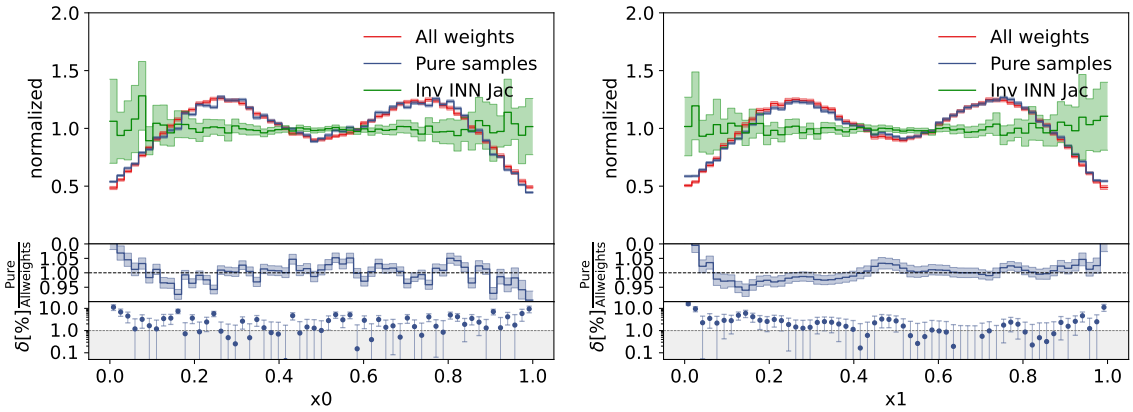
Baseline

We can lead the recycling to very fast converging with the 1cycle policy as in the following case. Here we already turned on the training while generating the training data. Now the training has the same training length as the generating baseline, but this time three epochs over 175 batches. We start with a learning rate of $8 \cdot 10^{-5}$ and raise it as usual by a factor of ten.
The loss in Figure 28a now reduces only down to -10^{-3}, but to plot the loss in a logarithmic scale we add the constant 0.1. The final distribution in Figure 28b looks close to the true one, but in the first dimension in Figure 29a there are a few more artifacts left than in the second dimension. In Figure 30 we see that the recycling makes the distribution narrow quite quickly. The last integral estimate has a ΔI of 0.11.



(a) Loss curve **(b)** Sample distribution

Fig. 28: Results of recycling for 5 epochs on 100 generated batches



(a) Projection on 1st dimension

(b) Projection on 2st dimension

Fig. 29: Projections of sample distribution

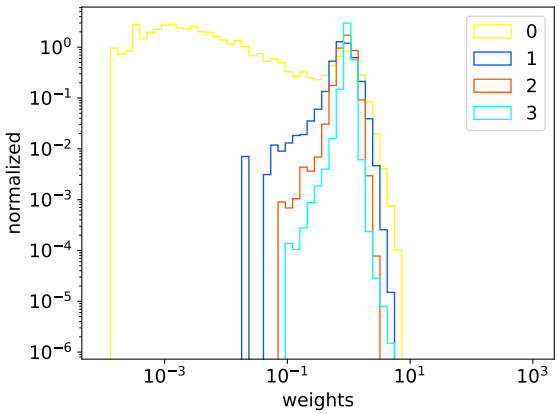
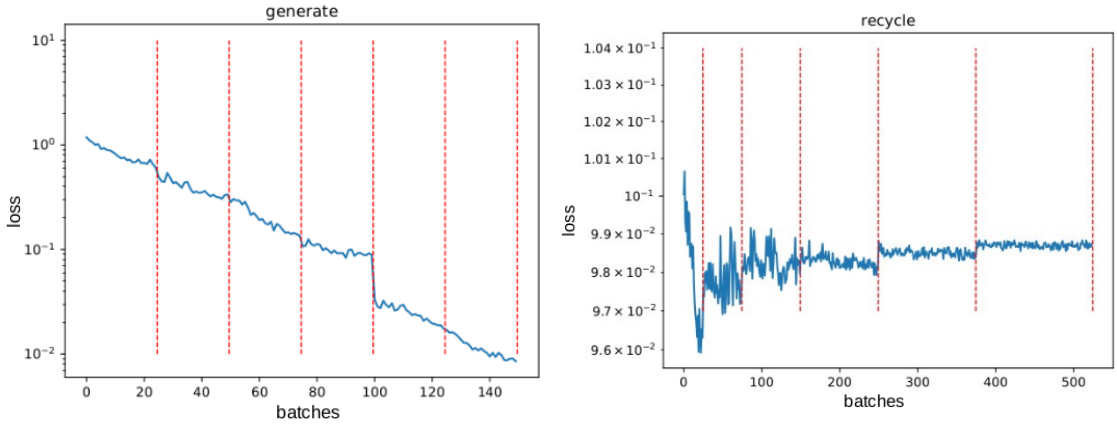


Fig. 30: Weight distribution after generating (0) and every epoch (1-3)

8.3 Combined training

Our goal is now to combine these two types of training in a more suitable way. We saw that recycling is faster and trains more stable. Though we want to generate the training data by ourself and use the fact, that meanwhile we can already improve the model.
After experimenting a bit we find a good compromise: We stick to the constant learning rate of 10^{-5} for generating and schedule the recycling by the 1cycle policy. After every generated 25 batches we switch to recycling, marked by the red dashed lines in Figure 31a. In the recycling periods we train over all data generated up to this point, visible again by the red markers in Figure 31b. All in all the network processes 675 batches.

The generating loss curve shows that at some points recycling is boosting the training a lot. Instead the generating periods seem to push up the recycling loss, but we assume this is a sign of preventing overfitting.
In Figure 32a the intermediate states after generating are plotted in yellow merging to red and after every recycling period in blue to turquoise. Comparing the width of the final weight distribution in turquoise with the recycling baseline in Figure 30 we see very small improvement of the overall shape, but also some outliers appearing,



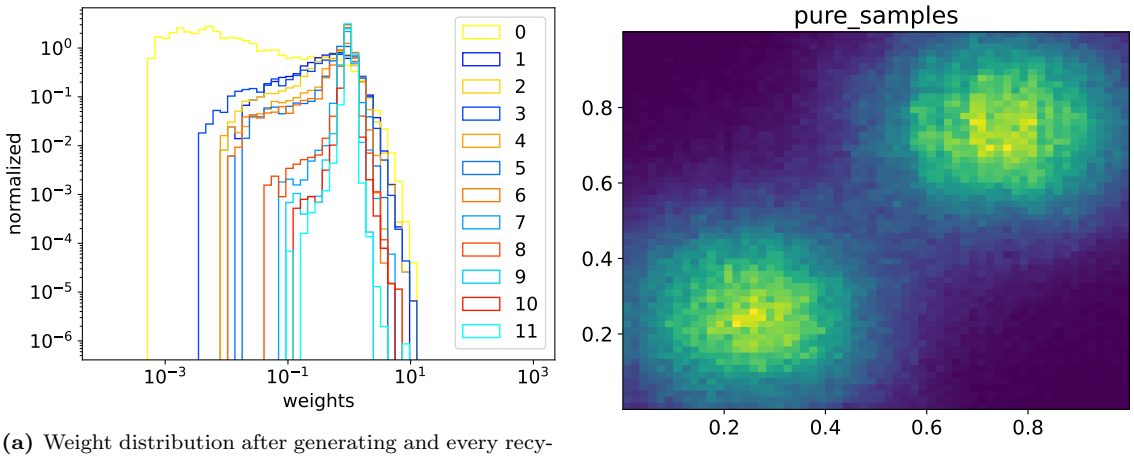
(a) Loss curve

(b) Sample distribution

Fig. 31: Loss for recycling every 25 batches over total data - 6 times

which increase the range of weights. The relative error of the last integral estimate makes the same small improvement with $\Delta I = 0.091$. In Figure 33a and 33b we see some outliers, but overall little significant deviation.

All in all this run is a success, but with over a minute of training this is still much

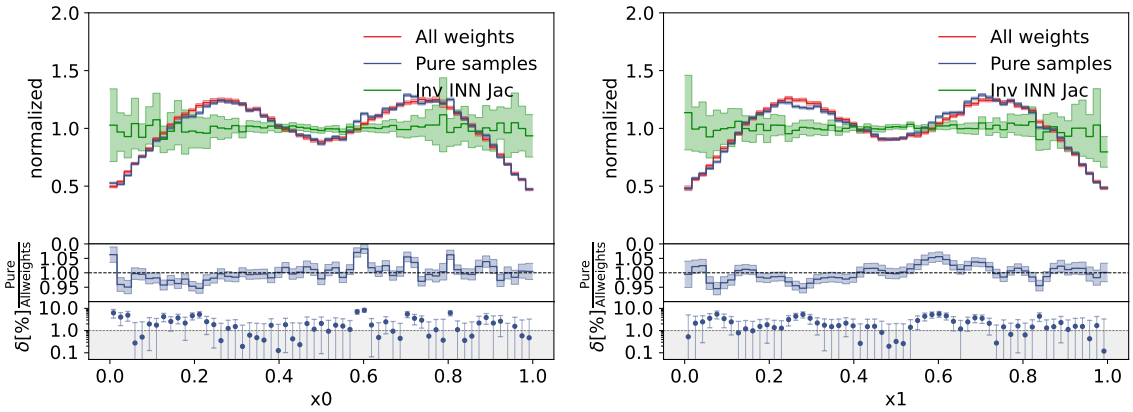


(a) Weight distribution after generating and every recycling epoch

(b) Projection on 1st dimension

Fig. 32: Results of recycling after every 25 generated batches over total data - 6 times

longer than vegas needs for this function. This is the reason, why we also look at the work of the group we took the concept of INN generating from.



(a) Projection on 1st dimension

(b) Projection on 2st dimension

Fig. 33: Projections of sample distribution

9 Outperforming I-flow benchmark

There is another benchmark we can create. The training described in subsection 5.2 was introduced also by Gao, Isaacson and Krause [2], who called their network *I-flow*. We can use their implementation, published on gitlab [28], for our toy function. To show that we are indeed able to make improvements to the model, we shall first copy I-flow's architecture to our code.

9.1 I-flow setup

Efficiency

For a two-dimensional function I-flow uses a network which is a hundred times smaller than what we used to work with. In Table 4 we list up our old hyper parameters and the ones I-flow uses for a 2d-problem. All in all this makes 1123070

hyper parameter	so far used	I-flow
Number of blocks	10	2
Number of layers	3	5
Layer size	256	32
Number of spline bins	60	16

Tab. 4: List of hyper parameters for network architecture

parameters to learn for us and only 9698 for I-flow.

Tensorflow library

I-flow is coded in tensorflow library [29]. Thereby some difficulties do arise, when we try to copy it to the pytorch library. Some of them are related to the different initialisation of layers in tensorflow on the one hand and the different initialisation of the splines on the other hand.

Latent space and permutation

I-flow uses a flat latent space, where it samples the input for the network from. For two dimensions I-flow uses only two blocks. While we used a rotation matrix between our blocks, they take here just a fixed matrix which flips the channels.

I-flow loss function and one mistake of ours

Per default I-flow employs the 'exponential' divergence as we do it. Although at this point we noticed a bigger mistake in our implementation: We forgot to eliminate the factor our distribution of samples brings in. Thus in our implementation the factor $1/g(r)$ was missing as described in subsection 5.2. This means that our results for generating could have been worse than they should be.
Another difference in the code of I-flow is the normalisation of f and g:
In the code they take the mean of f/g, reusing this again for calculating the integral. This mean is normalising f(r), while they do not normalise g(r). We call this normalisation *norm1*:

$$f'(r) := \frac{f(r)}{\langle f/g\rangle_r},\ g'(r) := g(r) \Rightarrow \frac{f'(r)}{g'(r)} = \frac{f(r)}{g(r)}\langle f/g\rangle_r \tag{9.1}$$

This again is not completely comprehensible for us, since this is not necessarily the same as our normalisation *norm2*:

$$\frac{f(r)}{\langle f\rangle_r}\cdot\frac{\langle g\rangle_r}{g(r)} =: f''(r)\cdot\frac{1}{g''(r)} \tag{9.2}$$

Training type

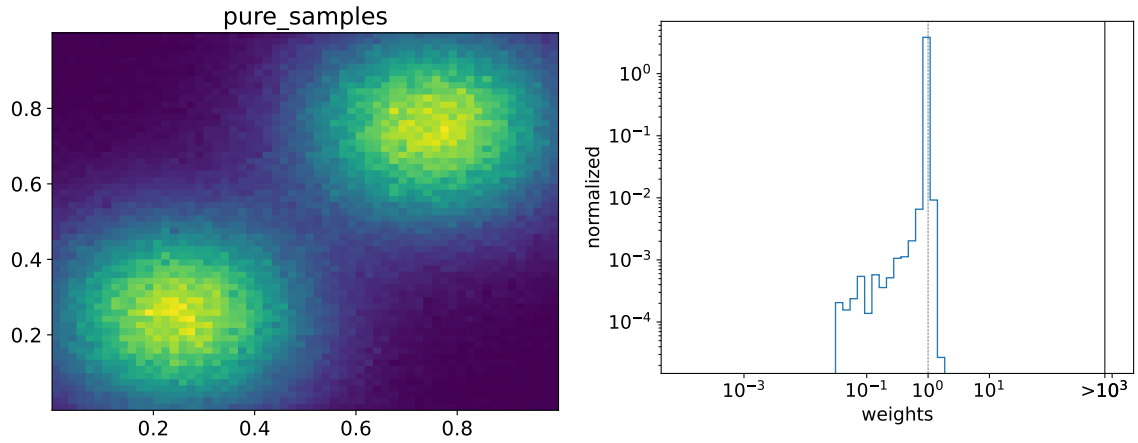
They only use generating as training. Though in the following we will see that their training is extremely stable and even with a ten times higher learning rate it converges reliably.

9.2 I-flow training

Let us see how well I-flow does in fitting the double Gaussian toy function. From this point on the batch size is per default set to 5024. This makes the GPU even faster to process the code.
We go first up to 1000 batches.

Default configuration

In the default setting I-flow trains with a constant learning rate of 10^{-3}.
The results are for us surprisingly good: In Figure 34a the sampling distribution looks very smooth and does not show any artifacts or similar defects. The weights in Figure 34b are extremely narrow, especially on the right side, there is almost no tail.
In Figure 35, we see the generating loss going down to 10^{-3}.
For this run I-flow gives for the last integral estimate with 25000 evaluations a relative error of $2.7\cdot10^{-4}$, which makes a ΔI of 0.043 and therefore better than what we got so far.



(a) Sampling distribution

(b) Weight distribution

Fig. 34: I-flow results after 1000 batches with *norm1* and learning rate of 10^{-3}

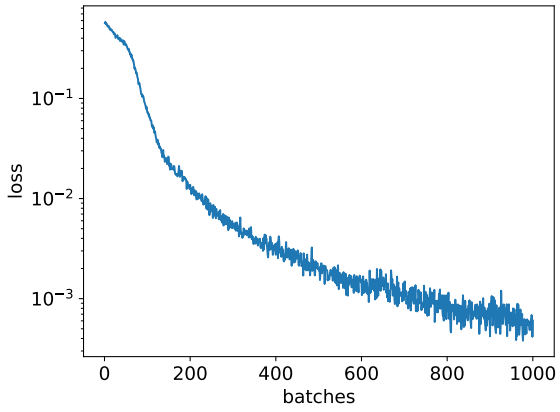


Fig. 35: Loss curve for I-flow in default configuration

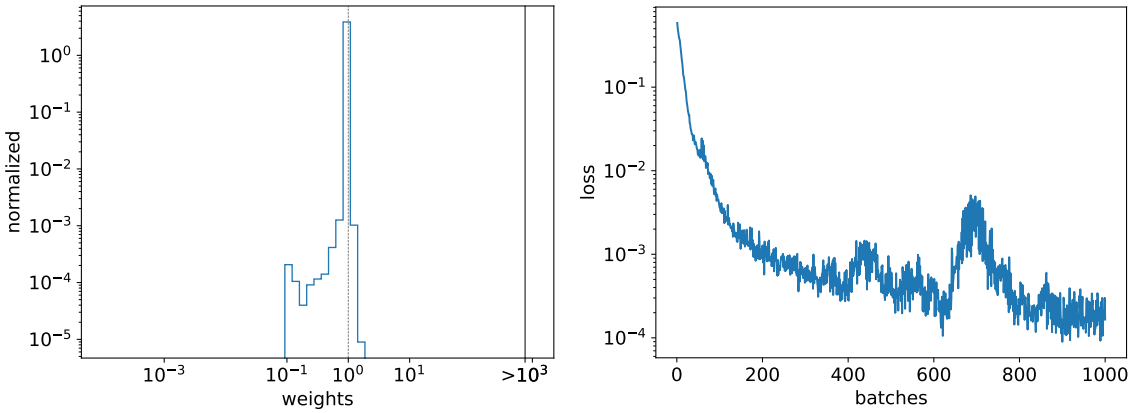
Increasing learning rate

Next we want to see how stable the I-flow training really is and let it run with a ten times higher learning rate. Figure 36b shows I-flow does still converge very well, only a few bumps show up. The final loss as well as the weight distribution in Figure 36a look even better than before. Also the ΔI is lowered to 0.016.

Different normalisation

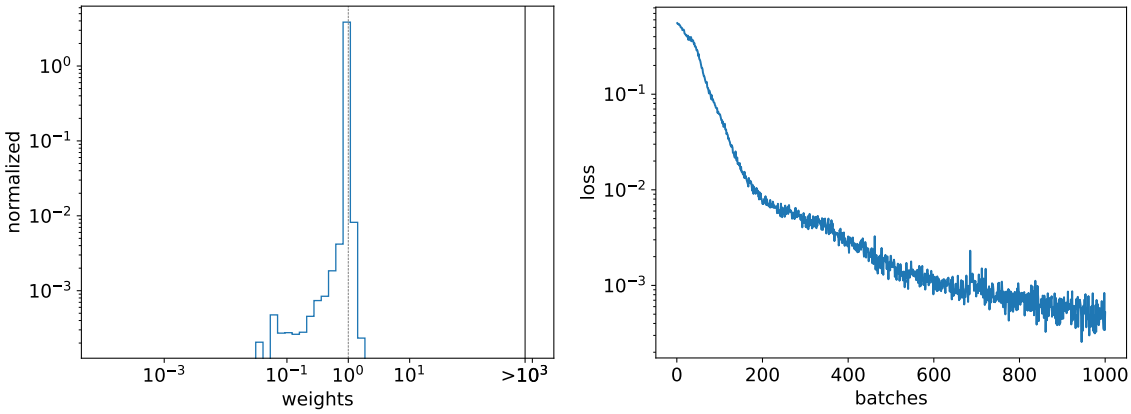
Next we reset the learning rate to 10^{-3} and change the normalisation, to see if I-flows choice was made for reasons of performance. *norm2* is indicated in equation 9.2 and is also the default one, we used for our loss implementation before.

Instead of a crucial deterioration, we rather see small enhancements in comparison to the default configuration. The width of the weight distribution in Figure 37a is smaller and so is the ΔI of the last estimate with 0.032. This means, there are most probably other reasons for the I-flow normalisation and we can stick to ours.



(a) Weight distribution **(b)** Loss curve for 1000 batches

Fig. 36: I-flow results for learning rate of 10^{-2}



(a) Weight distribution **(b)** Loss curve over 1000 batches

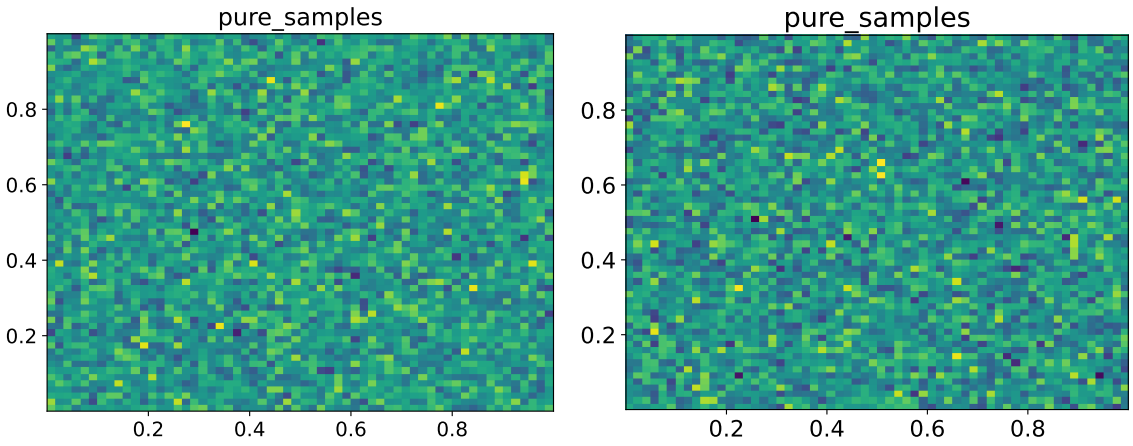
Fig. 37: I-flow results for normalisation as in eq. 9.2

9.3 Torch-flow - mimicking I-flow in pytorch

To get a copy of I-flow in pytorch we borrow all the hyper parameters I-flow uses and start to change also the initialisation in the layers and splines towards I-flow.

Initialisation

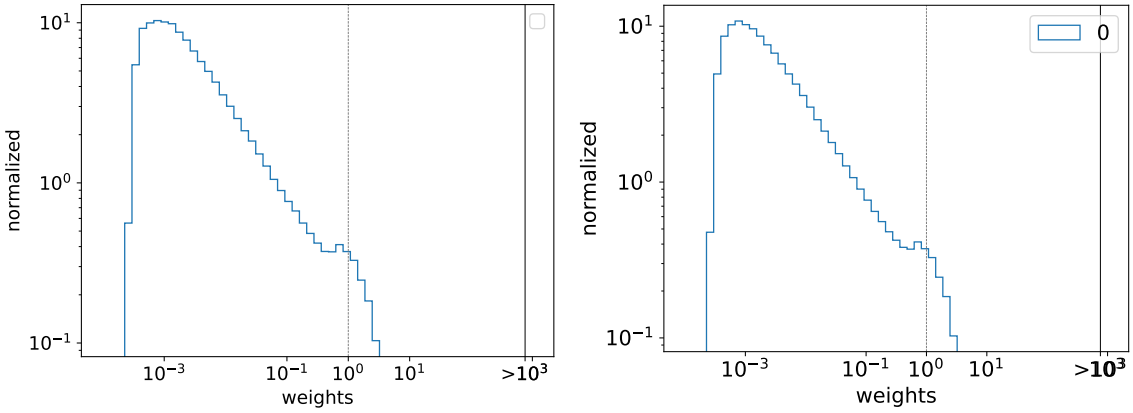
I-flow has a flat initialisation. To get a similar result, we have to set the initial derivatives in the splines to one and also the last layers weights and biases to zero. Now in Figure 38 there is now difference in initialisation visible anymore and the weights look almost the same too.



(a) I-flow sampling without training

(b) Torch-flow sampling without training

Fig. 38: Comparing initial states of I-flow and our copy Torch-flow



(a) I-flow weight distribution without training

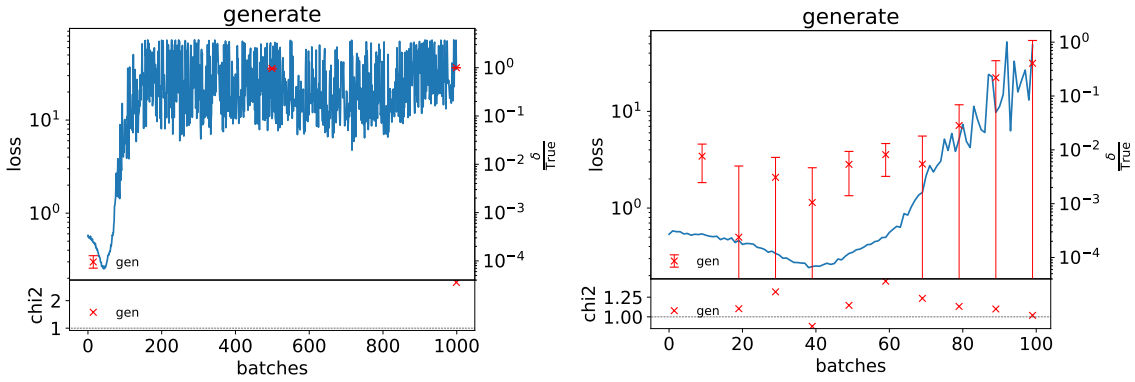
(b) itorch

Fig. 39: Torch-flow weight distribution without training

Training

Since we still want a precise copy of iflow in pytorch, we implement also the normalisation in the same way (Equation 9.1). Unfortunately our training diverges as can be seen in Figure 40a. The first 50 batches look exactly like in the I-flow loss curve, but then the loss turns upwards.
To observe the divergent behaviour, we look on the first 100 batches more carefully. In Figure 40b the part of the first 40 batches is a bit scaled, but it is still the same behaviour.

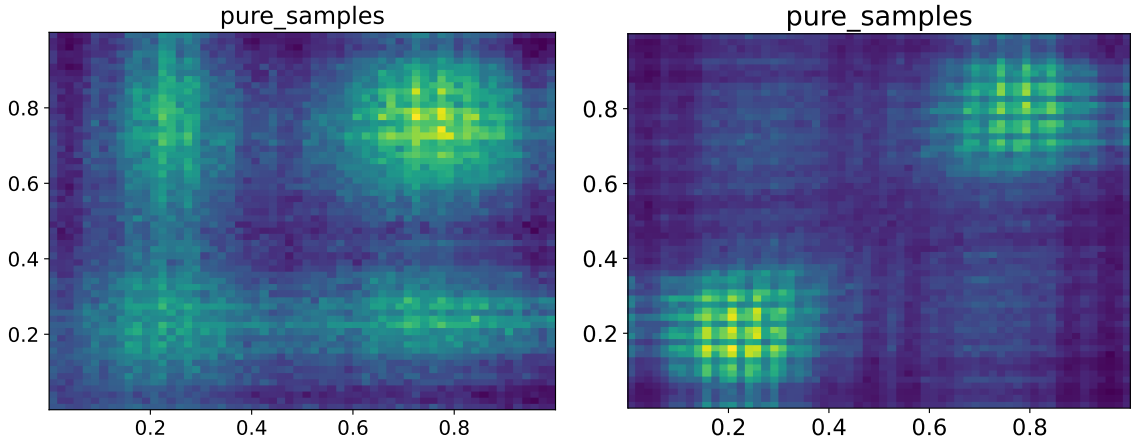
When comparing the sample distributions after 40 batches with a I-flow result after 40 batches in Figure 42, the plots look very similar. In Figure 41b we see already a tendency of our model to crook the lines, what I-flow does not do. This reinforces over the training and in the end only a thin line remains.



(a) Loss curve for 1000 batches

(b) Loss cruve for 100 batches

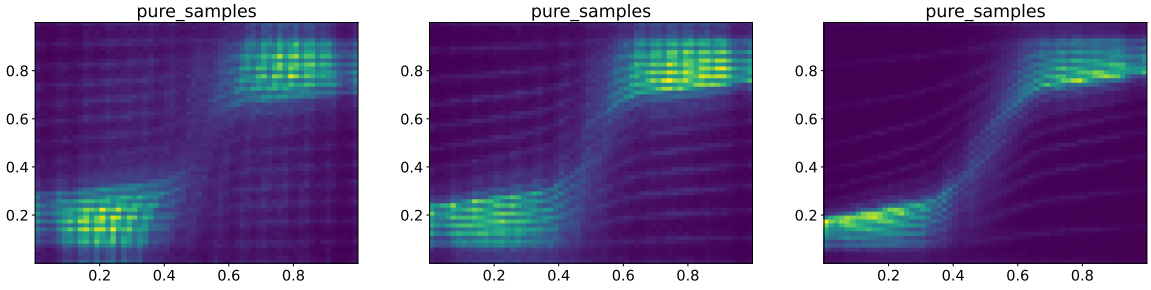
Fig. 40: Torch-flow loss curves for training with *norm1*. Additionally the relative deviations of the intermediate integral estimates and the resulting χ^2-values are plotted with the corresponding axis on the right side.



(a) I-flow after 40 batches

(b) Torch-flow after 40 batches

Fig. 41: Comparing I-flow and Torch-flow after 40 batches



(a) Torch-flow after 50 batches

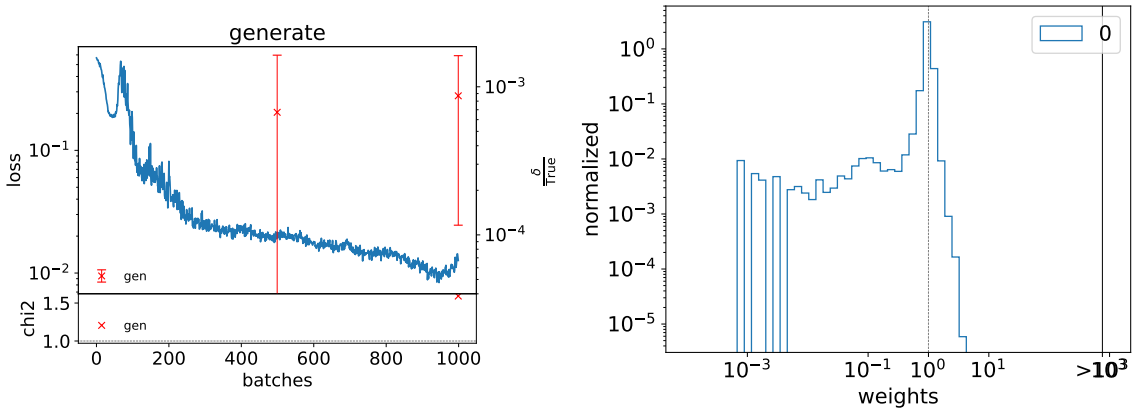
(b) Torch-flow after 60 batches

(c) Torch-flow after 70 batches

Fig. 42: Divergence of Torch-flow with *norm1*

Torch-flow with norm2

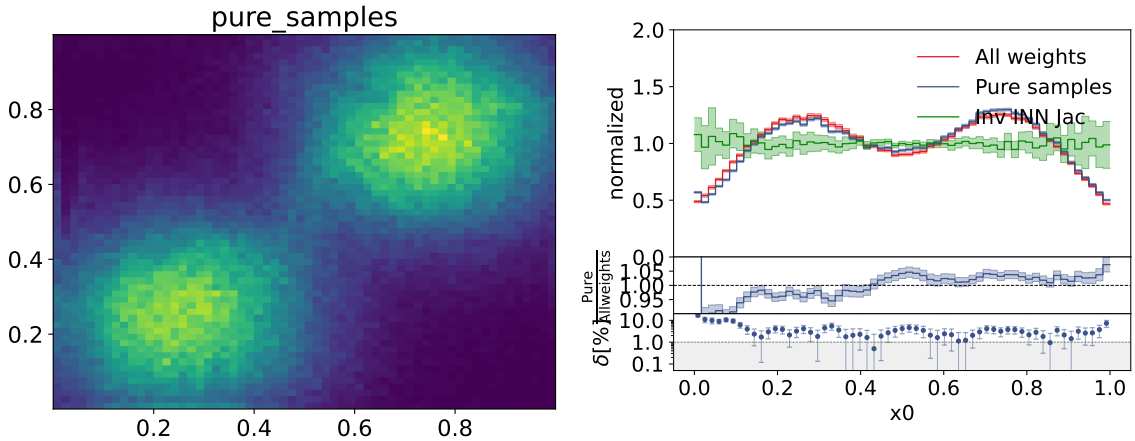
Since with I-flow we see no big difference running it with the other normalisation (*norm2*, Equation 9.2), it was good enough, if we managed to copy I-flow with this implementation. Indeed, this time the network does not diverge, but neither it shows the same performance as I-flow. Again the loss in Figure 43a behaves like I-flow for the first 40 batches and then, after a bump it sticks to a plateau at $\sim 10^{-2}$. The weights look far worse than what we saw for I-flow too. In the sample distribution in Figure 44 we can observe a bias to the direction of the second Gaussian.



(a) Loss curve with two integral estimates

(b) Weight distribution after 1000 batches

Fig. 43: Training of Torch-flow with *norm2*



(a) Sample distribution after 1000 batches

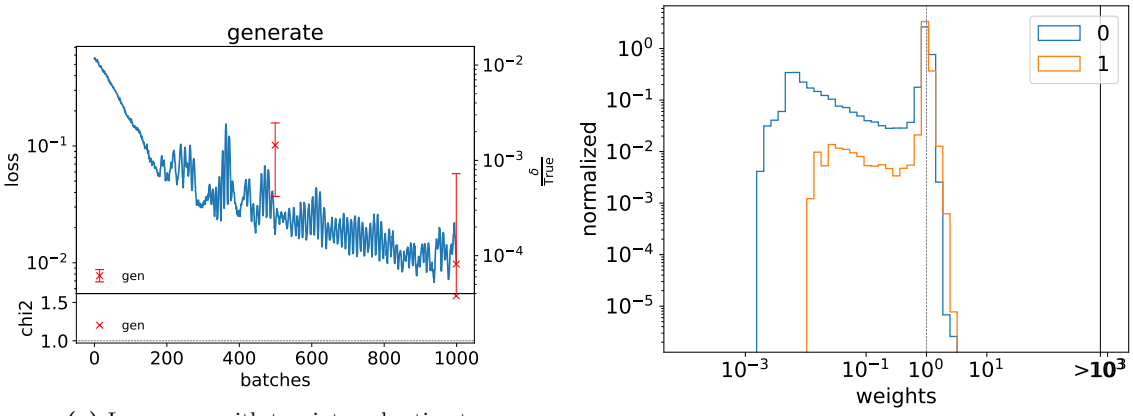
(b) Sample distribution projected on first dimension

Fig. 44: Results of Torch-flow with *norm2*

G-flow - Torch-flow with Gaussian latent space

We expect the continuous rotations between the blocks (*soft permuting*) to proof their strength in bigger networks for functions with more complexity. This is why we switch here to the Gaussian latent space and look how the model generates.

Unfortunately we have to note again a very unstable training and thus we reduce the learning rate by 1/4.
The loss in Figure 45a still evolves very noisy after 200 batches, but the weight distribution shows the right direction.



(a) Loss curve with two integral estimates

(b) Weight distribution after 500 and 1000 batches

Fig. 45: Training of G-flow with learning rate of $2.5 \cdot 10^{-4}$

9.4 BiG-flow - Bijective Gaussian iflow

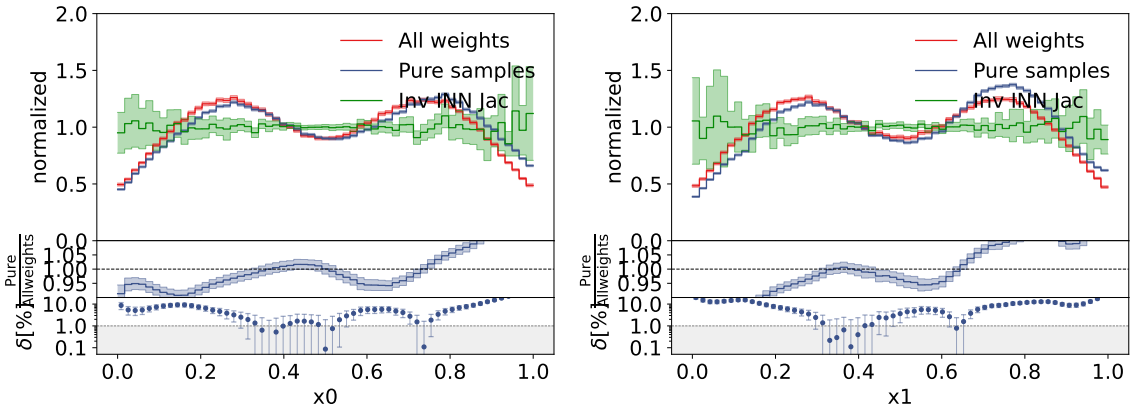
We keep the setting of "G-flow" and add our most promising improvement: INN recycling.

Only Recycling

First we want to find suitable hyper parameters for the recycling training. Therefore we again generate 100 batches with the learning rate set to zero.
For the following results we did train one epoch on the weighted 100 batches with a learning rate of 10^{-2}. In the sample distribution in Figure 47 we again see a systematic shift to the second Gaussian.

We can compare this to I-flow training for 100 batches with the same learning rate of 10^{-2}. The difference in the weight distribution is not very large, though the recycling seems to be a bit faster.

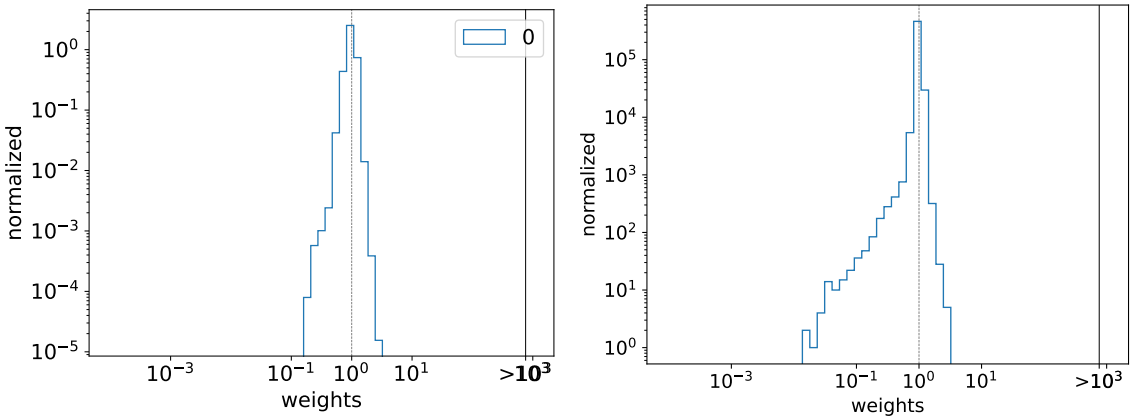
I-flow calculates an Integral estimate with a relative error of $1.7 \cdot 10^{-4}$ ($\Delta I = 0.027$), while the last estimate with BiG-flow with the relative error of $9 \cdot 10^{-4}$ ($\Delta I = 0.15$) is less precise.



(a) Projection on first dimension

(b) Projection on second dimension

Fig. 46: Sample distribution for BiG-flow recycling with learning rate of 10^{-2}



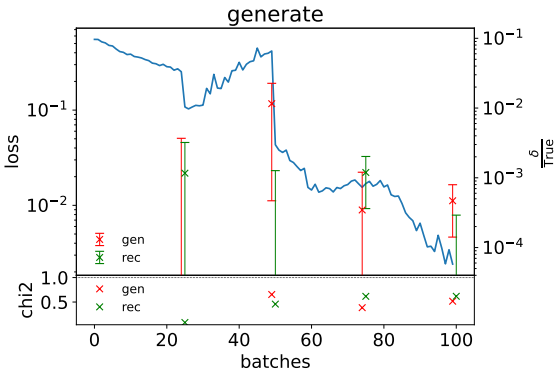
(a) BiG-flow, only recycling

(b) I-flow, learning rate of 10^{-2}, 100 batches

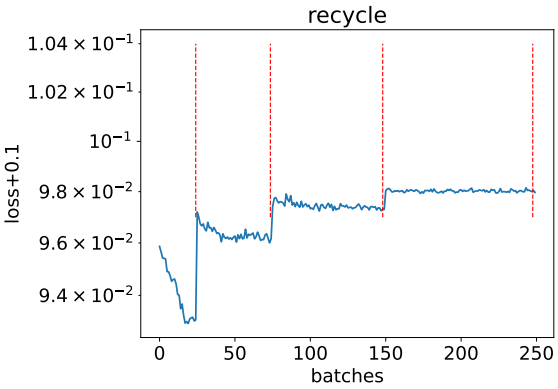
Fig. 47: Comparison of BiG-flow and I-flow in weight distribution

Combined training

As we had good experiences with switching between generating and recycling quite frequently, we tried the same schedule as in subsection 8.3, but four times of each. All in all we generated 100 batches and recycled 250, which took round about 43 seconds on a GPU. This time we scheduled the learning rate by a decay of 0.4 after 50 generated batches, respectively by the 1cycle policy for recycling.
This and the integral evaluations are also illustrated in the loss curves Figure 48. In the second generating period we see the loss turning again upwards, but then this gets stopped and reversed by the next recycling period. As we know it from before, the recycling loss is again increasing after every generating period - again, this rather seems to be a sign of correct training. The last integral evaluation gives a relative error of $3 \cdot 10^{-4}$, making $\Delta I = 0.047$ and thus it is not more precise than I-flow trained for 100 batches and with learning rate of 10^{-2}.

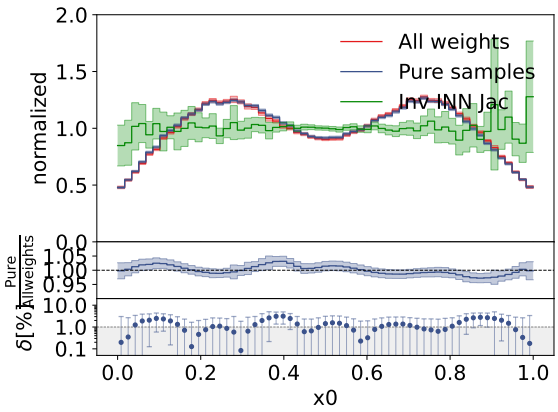


(a) Generating loss curve, with relative deviation of integral estimates

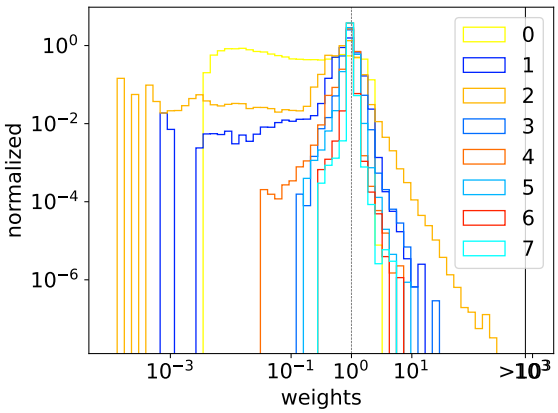


(b) Recycling loss curve

Fig. 48: Loss curves of combined training types with BiG-flow



(a) Sample distribution projected on first dimension

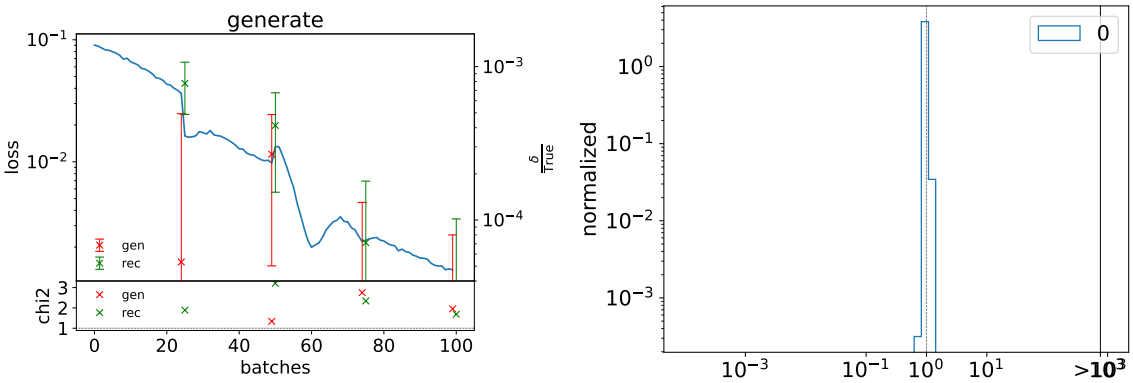


(b) Weight distribution after every period

Fig. 49: Results for combined training types with BiG-flow

Digression: Comparison with vegas

To compare our network with vegas also on a factorisable function, we integrate the four Gaussian toy function from Figure 18a. Judging by the weight distribution in Figure 50b, this function obviously simplifies the task also for our INN. The relative error of the last estimate is $7 \cdot 10^{-5}$ - this time with $2 \cdot 10^5$ evaluations like vegas in section 7.2. ΔI is then calculated to 0.034.



(a) Generating loss curve, with relative deviation of integral estimates

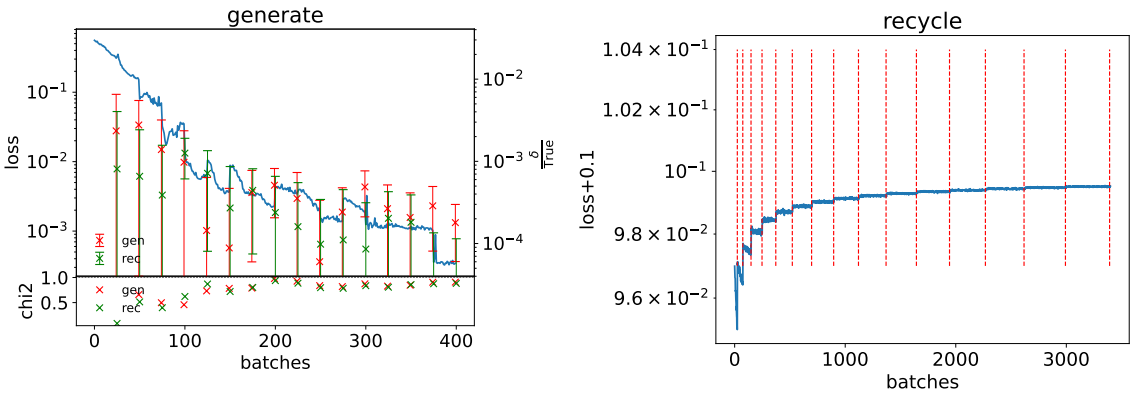
(b) Weight distribution

Fig. 50: BiG-flow as in section 9.4 trained on four Gaussian toy function

Long training

Lastly we look what the training converges to by training for longer time (∼8 minutes). We reduce the learning rate for both training types a bit, but use again the 1cycle policy for recycling and the exponential decay for generating.

The loss in Figure 51a goes down to values of 10^{-4}, though not so far as I-flow in



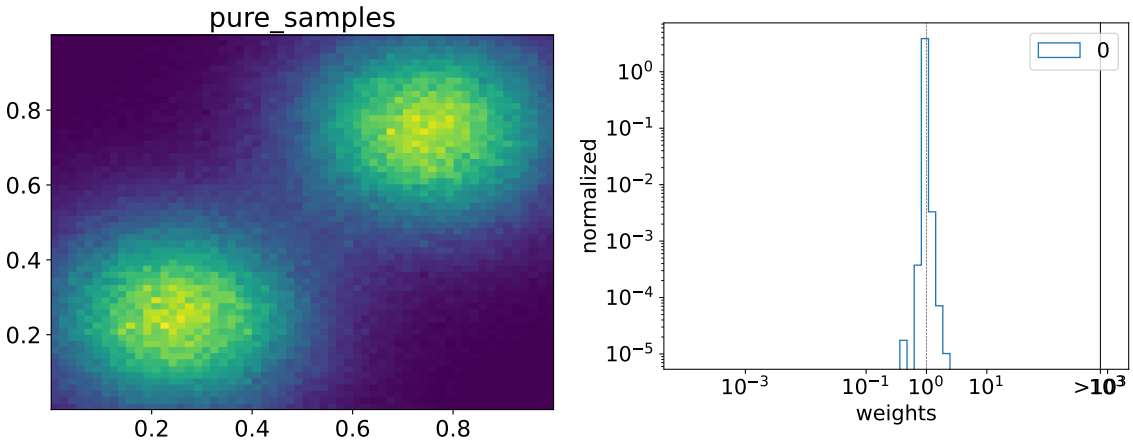
(a) Generating loss curve, with relative deviation of integral estimates

(b) Recycling loss curve

Fig. 51: Loss curves of combined training types with BiG-flow for long training

Figure 36b training with higher learning rate for 1000 batches. The last estimate, again determined with 25000 samples, is with a relative error of $1.0 \cdot 10^{-4}$ as precise as I-flow in the mentioned run. ΔI is accordingly low with 0.017.

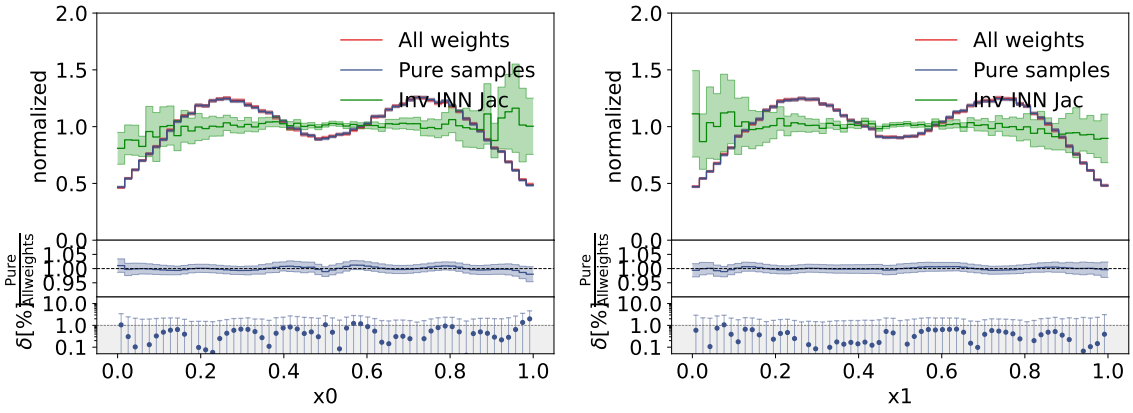
Though the weight distribution in Figure 52b is significantly more narrow than, what we have seen so far for the double Gaussian toy function. Also in Figure 53, there are no significant deviations visible.



(a) Sample distribution

(b) Weight distribution

Fig. 52: Results of combined training types with BiG-flow for long training



(a) Sample distribution, projected on first dimension

(b) Sample distribution, projected on second dimension

Fig. 53: Sample distribution projections of BiG-flow for long training

10 Summary

We built up a benchmark with vegas for the simple QCD process $gg \to ggg$. This we could compare with the commonly used framework for event generation Madgraph, which also estimates the integral. The result of Madgraph is (15.296 ± 0.015). Unfortunately vegas estimate deviates with $(12.78 \pm 0.16)\text{GeV}^{-2}$ from Madgraph by multiples of its standard deviation. One possible explanation is that vegas underestimated its error, because it did not sample in the regions, with very high integrand values. This can also motivated by the high complexity of the integrand and a look on the sample distribution of vegas, so in Figure 16: This deviates still strongly from the integrand.

Subsequently we switched to simpler toy functions in two dimensions. As consequence of vegas assumptions about the integrand, the double Gaussian toy function was a big challenge for it. The weight distribution in Figure 17b is extremely wide and also the integral estimate is not as precise as for factorisable toy functions. For the latter ones we picked the four Gaussian toy function and obtained a much more

narrow weight distribution, showed in Figure 18b. Though for both functions we managed to design a comparably fast training for our BiG-flow with equally good or better results. In Table 5 and 6 we compare the results of vegas and BiG-flow. For better comparability of the integral precision, we list ΔI, which is again calculated by the standard deviation of the estimate, relative to the true value and multiplied with the root of evaluation number. For assessing the weight distribution we count the number of bins filled by the weights in the plots linked in the tables.

This shows that at least in low dimensions our network can compete with vegas

Double Gaussian

	vegas (Fig. 17b)	BiG-flow (Fig. 49b)
Δw_{tr} [bins]	> 20	11
ΔI	$7 \cdot 10^{-1}$	$5 \cdot 10^{-2}$

Tab. 5: Comparison of vegas and BiG-flow in weight distribution and integral estimate on double Gaussian toy function

Four Gaussian

	vegas (Fig. 18b)	BiG-flow (Fig. 50b)
Δw_{tr} [bins]	4	3
ΔI	$7 \cdot 10^{-2}$	$3 \cdot 10^{-2}$

Tab. 6: Comparison of vegas and BiG-flow in weight distribution and integral estimate on four Gaussian toy function

even in integrating factorisable functions.

In subsection 8.1 we learned that our generating training strongly depended on the type of toy function. The double Gaussian lead with high learning rate often to unstable training, where our network converged not to the target function as in Figure 22 or even diverged. In Figure 23a we could observe that in these cases the Jacobian of the INN evolves numerically unstable in the regions, where the network starts to sample only very rarely. In these runs we were missing the term $1/g(r)$ in our loss, which cancels the contribution of the sampling distribution. When we noticed this, we assumed this might be the reason for this behaviour. Though when we continued with the correct loss in subsection 9.3 we observed still the same unstable training, when setting the learning rate too high.

For this reason we used for the generating baseline a low learning rate of 10^{-5}. After 700 batches it was clearly not converged, but we stopped the training to compare the efficiency. The same values as in Table 6 are calculated also for the two baselines of our combined training schedule in Table 7.

For the recycling training we found that the 1cycle policy for the learning rate worked the best. For this baseline we already used both types of training, but

strictly separated by just one switch.
Finally we trained the network, switching between both types of training periodically. Also here we kept the 1cycle policy for recycling. We obtained a few more outliers in the weight distribution than for the recycling baseline, though the integral estimate was a bit more precise as visible in Table 7. Also the error plots in the plots 33a and 33b the errors are bit smaller than in plots 29a and 29b.
Thus we have to state that our periodic alternation did help not significantly.

	generating (Fig. 26)	recycling (Fig. 30)	combined (Fig. 32a)
Δw_{tr} [bins]	> 20	15	17
ΔI	$4 \cdot 10^{-1}$	$11 \cdot 10^{-2}$	$9 \cdot 10^{-2}$

Tab. 7: Comparison of generating and recycling baselines with combined training in weight distribution and integral estimate on double Gaussian toy function

Lastly we let us guide by the publicly available code of I-flow, written in tensorflow and tried to mimic this network. This made our network training a lot faster, since we downsized it to only one percent of the size before. Unfortunately we did not manage to create a precise copy of I-flow in pytorch. The main problem was still the instability of our generating training as observed before. Though we fixed our mistake in the loss, our network would still diverge or when reducing the learning rate would not be able to decrease the loss as low as I-flow. We assume there might be still some differences in the implementation of the splines or in some built-in tensorflow methods.
When applying the two main changes, which we assume to show their superiority in higher dimensions and more complex integrands, we could improve our results. This is firstly the Gaussian latent space, where I-flow just uses a flat space and secondly the recycling training. We call the resulting architecture BiG-flow. A pure recycling training with data generated by a untrained model, already showed comparable results as I-flow. Only the integral estimate was not as precise as I-flow. Finally we again applied our combined training as before, once in a short training and then in a long one.
In Table 8 we compare I-flow training for 100 batches and a learning rate of 10^{-2} (I-flow$_{100}$), BiG-flow only recycling for 100 batches (BiG-flow$_{\text{rec}}$) and BiG-flow with combined training processing all in all 350 batches (BiG-flow$_{\text{short}}$). For the long runs we compare in Table 9 I-flow training with the same learning rate for 1000 batches (I-flow$_{1000}$) and BiG-flow processing all in all 3800 batches, but generating only 400 (BiG-flow$_{\text{long}}$).
What we can read out of Table 8 is that the recycling training does very well in decreasing the range of weights. Though in estimating the integral with high precision, I-flow is still much more efficient.
When we consider the evaluation of the integrand as bottle-neck of the process, as it is the case for the matrix element, the situation assessment changes: If we would assume the evaluation of the integrand to last around five to six times longer than processing a data point through the network, the two networks in Table 9 would be equally fast. Though BiG-flow is again superior in decreasing the range of weights.

Short

	I-flow$_{100}$ (Fig. 47b)	BiG-flow$_{\text{rec}}$ (Fig. 47a)	BiG-flow$_{\text{short}}$ (Fig. 49b)
Δw_{tr} [bins]	20	11	11
ΔI	$3 \cdot 10^{-2}$	$15 \cdot 10^{-2}$	$5 \cdot 10^{-2}$

Tab. 8: Comparison of I-flow and BiG-flow in weight distribution and integral estimate on short training

Long

	I-flow$_{1000}$ (Fig. 36a)	BiG-flow$_{\text{long}}$ (Fig. 52b)
Δw_{tr} [bins]	11	7
ΔI	$16 \cdot 10^{-3}$	$17 \cdot 10^{-3}$

Tab. 9: Comparison of I-flow and BiG-flow in weight distribution and integral estimate on long training

11 Conclusion/Outlook

We could show that the task of vegas can be solved by INNs at least equally well. On problems, where vegas makes wrong assumptions, our INN clearly stripped vegas out.
In the generating training we had to state problems until the very end. Nevertheless we managed to combine this training with a second type of training, such that both types of training would preserve their strengths and were not working against each other. While recycling seemed to narrowing the range of weights more quickly, the generating I-flow has a very high precision in it's estimates.
For some circumstances we showed that a connection with a second training has an advantage over training with only one type of training.

Unfortunately we did not manage in the frame of this thesis, to experiment with higher dimensions. The task here is to find a generalisation of how to increase the size of the network and in which manner does the training time increase with more and more dimensions. Also the schedule of changing the training type could get generalised. For instance the network could check if it reached a loss plateau and then switch to the other training.
After the networks trains reliably on higher dimensional toy functions, the comparison with our vegas benchmark on the gluon process is due. Since the benchmark clearly showed its weaknesses, we would expect that a BiG-flow architecture in seven dimensions should do better.
Additionally other processes can be chosen to experiment with. One question is, if the network has to be initialised completely for every new process or if a pre-trained version could be used for several different processes.

12 References

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

[2]C. Gao, J. Isaacson, and C. Krause, "I- flow: high-dimensional integration and sampling with normalizing flows", Machine Learning: Science and Technology **1**, 045023 (2020).

[3]M. Felcini (ATLAS Collaboration, CMS Collaboration), *Searches for Dark Matter Particles at the LHC*, tech. rep., 10 pages, 7 figures, Proceedings of the 53rd Rencontres de Moriond on Cosmology, March 17-24 2018, on behalf of the ATLAS and CMS collaborations (CERN, Geneva, Aug. 2018).

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

[5]D. J. Rezende and S. Mohamed, *Variational inference with normalizing flows*, 2016.

[6]I. Kobyzev, S. J. Prince, and M. A. Brubaker, "Normalizing flows: an introduction and review of current methods", IEEE Transactions on Pattern Analysis and Machine Intelligence **43**, 3964–3979 (2021).

[7]L. Dinh, J. Sohl-Dickstein, and S. Bengio, *Density estimation using real nvp*, 2017.

[8]D. P. Kingma and P. Dhariwal, *Glow: generative flow with invertible 1x1 convolutions*, 2018.

[9]M. E. P. D. V. Schroeder, *An introduction to quantum field theory*, `https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=729240` (Addison-Wesley, 1995).

[10]J. Alwall, R. Frederix, S. Frixione, V. Hirschi, F. Maltoni, O. Mattelaer, H.-S. Shao, T. Stelzer, P. Torrielli, and M. Zaro, "The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations", Journal of High Energy Physics **2014**, `10.1007/jhep07(2014)079` (2014).

[11]A. Buckley, J. Ferrando, S. Lloyd, K. Nordström, B. Page, M. Rüfenacht, M. Schönherr, and G. Watt, "Lhapdf6: parton density access in the lhc precision era", The European Physical Journal C **75**, `10.1140/epjc/s10052-015-3318-8` (2015).

[12]R. Kleiss, W. Stirling, and S. Ellis, "A new monte carlo treatment of multiparticle phase space at high energies", Computer Physics Communications **40**, 359–373 (1986).

[13]S. Plätzer, *Rambo on diet*, 2013.

[14]I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*, `http://www.deeplearningbook.org` (MIT Press, 2016).

[15]G. V. Cybenko, "Approximation by superpositions of a sigmoidal function", Mathematics of Control, Signals and Systems **2**, 303–314 (1989).

[16]M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function", Neural Networks **6**, 861–867 (1993).

[17]M. Straka, *Deep learning – úfal course npfl114*, `https://github.com/ufal/npfl114`, 2021.

[18]D. P. Kingma and J. Ba, *Adam: a method for stochastic optimization*, 2017.

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

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

[21]L. Dinh, D. Krueger, and Y. Bengio, *Nice: non-linear independent components estimation*, 2015.

[22]B. Stienen and R. Verheyen, "Phase space sampling and inference from weighted events with autoregressive flows", SciPost Physics **10**, `10.21468/scipostphys.10.2.038` (2021).

[23]J. A. GREGORY and R. DELBOURGO, "Piecewise Rational Quadratic Interpolation to Monotonic Data", IMA Journal of Numerical Analysis **2**, 123–130 (1982).

[24]C. Durkan, A. Bekasov, I. Murray, and G. Papamakarios, *Neural spline flows*, 2019.

[25]T. Kloek and H. K. van Dijk, "Bayesian estimates of equation system parameters: an application of integration by monte carlo", Econometrica **46**, 1–19 (1978).

[26]G. Lepage, *Vegas 5.1.1 documentation*, `https://vegas.readthedocs.io/en/latest/vegas.html`, Last accessed 14 February 2022, 2018.

[27]E. Bothmann, T. Janßen, M. Knobbe, T. Schmale, and S. Schumann, "Exploring phase space with Neural Importance Sampling", SciPost Phys. **8**, 69 (2020).

[28]C. Gao, J. Isaacson, and C. Krause, *I-flow repository*, `https://gitlab.com/i-flow/i-flow`, Last accessed 26 February 2022, 2022.

[29]M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, *Tensorflow: a system for large-scale machine learning*, 2016.

List of Figures

Acknowledgements

I want to thank the whole group for sharing with me a nice semester with good talks. My supervisors Tilman and Anja I want to thank for making a great job in directing us through the jungle of Machine Learning results and especially Theo for being a constant advisor on our side. My ally Simon I am grateful to for forcing me to learn how to handle git and joining me through all the desperate searches for bugs. Finally I want to thank also Mr Bittner for being extremely patient with us and writing polite mails, when we filled up again all the memories in the GPU farm.

Erklärung

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

Heidelberg, den 28.2.2022, Joran Valentin Köhler