Machine learning techniques applied to Monte Carlo integration

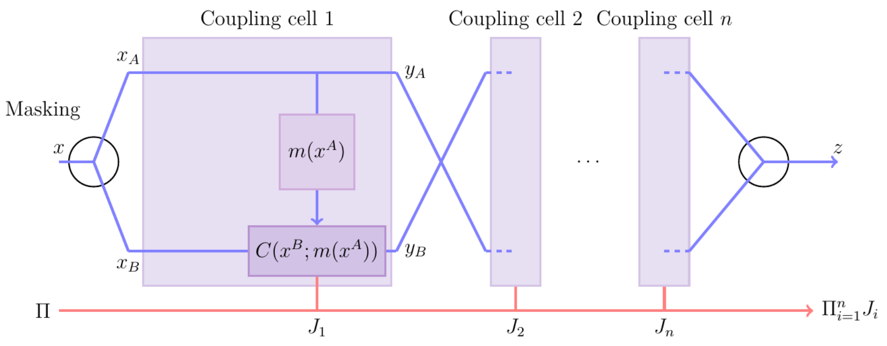
A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in High Energy Physics

by

Niklas Götz

September 2020



Supervisors :

Dr. Nicolas Deutschmann

Dr. Valentin Hirschi

Dr. Achilleas Lazopoulos

Prof. Dr. Charalampos Anastasiou

Department of Physics
Institute of Theoretical Physics
ETH Hönggerberg
Wolfgang-Pauli-Str. 27
8093 Zürich
Switzerland

Machine learning techniques applied to Monte Carlo integration

Niklas Götz

Abstract

The numerical calculation of cross sections through the integration of matrix elements often suffers from high computational cost or low precision due to the presence of peaks and cuts. In this thesis, we present a machine learning method in order to improve the efficiency of Monte Carlo integrations without prior knowledge of the integrand. This neural importance sampling algorithm is based on normalizing flows in the form of piecewise-quadratic coupling cells. It is an adaptive method that is trained on the integrand and provides a transformation which decreases its variance. This algorithm is implemented and tested on multidimensional test functions and tree level amplitudes. Its performance is compared with the commonly used VEGAS importance sampling. The importance of hyperparameter optimisation is demonstrated and an insight on the importance of different hyperparameters is given. The present implementation can be run on GPU, and is combined with an efficient phase space generator which allows fast and parallelised integration.

Acknowledgements

I would like to express my gratitude to my primary supervisors, Dr. Nicolas Deutschmann and Dr. Valentin Hirschi, who guided me throughout this project. I am very grateful for their support and dedication throughout these months of work, in these unprecedented circumstances. Without their crucial assistance, this project would have not been possible. I would also like to thank Dr. Achilleas Lazopoulos for giving me the opportunity to work on this fascinating topic, as well Dr. Ben Ruijl for his technical support.

I am very grateful for all the inspiring researchers who shared their knowledge with me in the past six years. Their exemplary dedication to science has been a guiding principle to me.

I would also like to thank all my friends from the High Energy Physics Master's Program. Their passion for research and their friendship deeply inspired me and will always be one of my dearest memories of these two years.

Additionally, I would like to express my deep gratitude towards my partner, for her support, understanding and advice. Her contribution to everything I have achieved since we met is invaluable. I am also thankful to my father, who never doubted my potential and always encouraged me to work hard for my dreams.

I would also like to thank the Studienstiftung des deutschen Volkes, the Max Weber-Programm and the Fondation de l'École polytechnique for funding my studies and supporting my personal growth and development.

Contents

Introduction **1**

1 Numerical integration **4**

1.1 Monte Carlo method 4

1.2 Importance sampling 5

1.3 Change of variables 6

1.4 The VEGAS algorithm 7

2 Neural importance sampling **10**

2.1 Basics of Neural Networks 10

2.2 Normalising Flows 13

2.3 Coupling cells 14

2.3.1 Number of coupling cells 16

2.3.2 Affine Coupling Transforms 17

2.3.3 Piecewise-Linear Coupling Transforms 18

2.3.4 Piecewise-Quadratic Coupling Transforms 19

3 Neural importance sampling in two dimensions **21**

3.1 Implementation 21

3.2 Comparison of piecewise-linear and piecewise quadratic coupling cells 26

3.2.1 Performance of the piecewise-linear coupling cell 27

3.2.2 Performance of the piecewise-quadratic coupling cell 28

3.2.3 Analysis of the behaviour of piecewise-quadratic coupling cells 31

3.3 Comparison to the VEGAS algorithm 35

4 Investigation of the quadratic-coupling cells in higher dimensions **37**

4.1 Setup 37

4.2 Results 39

4.3 Analysis of the impact of hyperparameters 43

5 Phase Space Generation **47**
5.1 RAMBO 47
5.2 RAMBO on diet 50
5.3 GPU compatible implementation 52
5.4 Phase Space Cuts 53
5.5 Hadronic Processes 55

6 Neural importance sampling for cross section integrations **57**
6.1 Modification of the loss function 57
6.2 Comparison of the neural importance sampling and VEGAS 58
6.2.1 $e^+e^- \to \mu^+\mu^-$ 58
6.2.2 $gu \to gu$ 59
6.2.3 $uc \to ucg$ 60
6.2.4 $d\bar{d} \to d\bar{d}$ via Z 61

Conclusions and outlook **65**

Appendix **68**

References **76**

List of Figures **80**

List of Tables **83**

Introduction

The ongoing experimental search for new signals at colliders such as the LHC is not only empowered by progress in experimental methods, but also by the ability to generate precise predictions from the theoretical models one wants to test. Such predictions often heavily rely on numerical simulations, which are a source of uncertainties. In order to be able to reliably test theoretical models, it is therefore necessary to reduce the uncertainties as far as possible.

Event generators are an important example for such numerical simulations. They link theory and experiment by generating virtual collider events using Monte Carlo (MC) methods [1, 2]. They can be exploited to produce physical events that are comparable to the ones that are directly observed at colliders. By combining these events it is possible to predict observables in an equal fashion as in experiment, where both the prediction and the experiment are affected by uncertainties. However, as the increasing precision of experiments leads to a higher need of precision for the numerical simulations too, the consumption of more and more computational resources grows at an unsustainable pace [3]. The computational cost for reaching a satisfying precision can be very high, especially for complex multi-jet cross sections. During the search for New Physics in hard-scattering events, processes with a high multiplicity of final-state particles are of high interest, especially those with many hard jets or intermediate elements. Examples for this are signals of micro-black-hole decay to many particles in proton-proton collisions [4], or 6-7 jet events from R-parity violating supersymmetric processes [5]. Such complex interactions demand a precise evaluation of the partonic scattering matrix which features many final state particles and can involve thousands of Feynman diagram contributions. Due to the appearance of resonances, regularised singularities, quantum-interference and kinematical cuts, it is extremely challenging for many adaptive algorithms to increase the efficiency. For such multi-jet events, it has been shown that the unweighting efficiency is an important factor for the computational load [6, 7]. The unweighting efficiency determines how many of the sampled events can be used for the integration. In traditional MC integration, a random phase space point x_i is interpreted as an event, and weighted by the value of the differential cross section at this point, $w_i = f(x_i)$. The integral is then the mean of all weights over N events, $< w >_N$. In order to perform unweighting, one aims at choosing a subset of events which follows the probability distribution function which is given by the weights itself, i. e. that the event x_i occurs with a likelihood of $\frac{f(x_i)}{f_{max}} = \frac{w_i}{w_{max}}$. The unweighting efficiency refers then to the average likelihood of accepting an event with this "hit-or-miss" algorithm. As a consequence, most generated events are not used if the unweighting efficiency is low. Although it is possible to modify the Monte Carlo integration in such a way that the unweighting efficiency is optimised, this requires specific knowledge of the integrand and is not a solution in cases in which the specific structure is not known.

Adaptive MC methods are a valuable alternative [8–15]. They adapt to the integrand by optimising the integration procedure to the form of the integrand. They often achieve this by importance sampling. Importance sampling means that one does not sample the random points uniformly over the integration domain, but more frequently in regions where the value of the integrand is high and gives a higher contribution to the value of the integral. One commonly used technique for this is the VEGAS algorithm [8]. It histograms the integrand along the coordinate axis and optimises the size of the histogram bins in order to learn the distribution of the integrand. However, as it does this for each axis independently, this can lead to suboptimal behaviour: VEGAS assumes the factorizability of the integrand, a requirement which fails if the variables have complex correlations. Foam [16] is a popular alternative based on stratified sampling, that means dividing the integration domain in subdomains. It uses an adaptive strategy to attempt to model the correlations of the variables, but the number of required sampling points grows exponentially with the number of dimensions. Foam and VEGAS are not satisfying in general, as often high-dimensional phase space integrals with non-trivial correlations between dimensions are required in important theory calculations. Therefore, it seems necessary to restrain from general adaptive approaches and to focus on specialised tools for the integration of specific processes, like WHIZARD [17] and MadGraph5_aMC@NLO [18]. These tools optimise the integration of differential cross sections by taking into account the knowledge about the process.

The topic of adaptive Monte Carlo methods has gained more attention due to the success of the field of machine learning (ML), which introduces new methods and tools and became also relevant for many areas of high-energy physics [19]. Regarding event generation, these techniques were used first for the integration in the context of high-energy physics in the form of boosted decision trees and generative deep neural networks to improve the performance of MC integration [20]. This improved the integration of non-separable high dimensional functions, for which traditional algorithms failed. Other authors proposed to use a dense neural network (DNN) in order to learn the phase space directly, which shows promising results [21]. A variable transformation was performed, which demonstrated that it is possible obtain significantly larger efficiencies for three body decay integrals than standard approaches. Theoretically, an algorithm based on a neural network (NN) can be inverted after the training in order to be used for the

MC integration as a sampler. The inversion of the NN requires inverting its Jacobian, which incurs a computational cost that scales as $\mathcal{O}(D^3)$ for D-dimensional integrals. Therefore, it is extremely inefficient to use a standard NN-based algorithm for sampling, as especially multi-jet events have a very high-dimensional phase space. This is because an n particle final state phase space is a $D \sim 3n - 4$ dimensional integral.

The goal of this thesis is to study NN architectures that can be trained on integrands in order to reduce the variance during the Monte Carlo integration and by this, increasing its precision. A ML algorithm based on normalizing flows (NF) provides a promising candidate for this. Normalizing flows are bijective mappings between statistical distributions [22, 23]. The idea of combining them with a NN for the MC integration was first proposed for non-linear independent components estimation (NICE) [24, 25], and later generalized [26, 27]. The crucial point in this approach is the introduction of coupling cells (CC), which allow the usage of NNs in the construction of a bijective mapping between the target and initial distributions in such a way that the Jacobian can be now evaluated in $\mathcal{O}(D)$ time. In this thesis, an approach based on these advances will be studied, similar to the recent explorative studies performed with these techniques [28–30].

Chapter 1 will remind about the MC integration method and the VEGAS algorithm, which will be the primary comparison to the ML-based algorithm discussed in this thesis. Chapter 2 outlines the structure of a NN, the definition of normalizing flows and how both can be combined efficiently in order to achieve an adaptive MC method. Chapter 3 discusses the implementation of the coupling layers for a two-dimensional integral, and compares the performance of the piecewise-linear to the piecewise-quadratic coupling cells. Chapter 4 discusses modifications necessary for the application in higher dimensions as well as the procedure of hyperparameter search, and demonstrates the performance of the approach in comparison to the VEGAS algorithm for gaussian multi-peaks. Chapter 5 introduces phase space generation algorithms, in order to be able to use the importance sampling in combination with matrix elements generated by MadGraph 5. The phase space generator maps the output of the mapping by the normalizing flows to phase space points at which the matrix element is evaluated, and is therefore a central component both for speed as for precision of the approach. An efficient implementation is demonstrated. Finally, Chapter 6 demonstrates the performance of the proposed approach on selected processes of high energy physics.

Chapter 1

Numerical integration

In order to investigate MC methods in the context of machine learning, it is necessary to gain an understanding of the sources of uncertainties of numerical integration. Only the Monte Carlo method is of interest for us, as other integration rules become very costly for higher dimensions.

1.1 Monte Carlo method

We start considering to integrate a function $f(\mathbf{r})$ over the d-dimensional unit hypercube Ω - for the application in high-energy physics, this is could be the differential cross section which has to be integrated over the phase space. In the case that the analytical solution is unknown, this integral can be approximated by

$$\hat{I}_X^{(N)} = \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_i) \quad \xrightarrow[N\to\infty]{} I = \int_\Omega d\mathbf{x} f(\mathbf{x}). \tag{1.1}$$

Here, the $\mathbf{x_i}$ are randomly drawn, such that they are uniformly distributed in Ω (with a normalised uniform distribution). The expression for the precision follows from another perspective on the integral, by seeing the integral as the expectation value for the value of f for a random, uniformly distributed $\mathbf{x}$.

$$I = \mathbb{E}_{X\sim U(\Omega)}(f(X)) = \mathbb{E}_{X\sim U(\Omega)}(\hat{I}_X^{(N)}) \tag{1.2}$$

Now, with this perspective one is able to express the variance of the estimator for the integral given by the MC method.

$$\mathrm{Var}(\hat{I}_X^{(N)}) = \frac{1}{N^2}\sum_{i=1}^{N} \mathrm{Var}(f) = \frac{\mathrm{Var}(f)}{N} \tag{1.3}$$

$\mathrm{Var}(f)$ may be determined by the unbiased sampling variance

$$\mathrm{Var}(f) \approx s_N^2 = \frac{1}{N-1}\sum_{i=1}^{N}(f(\mathbf{x_i}) - I_X^{(N)})^2. \tag{1.4}$$

As long as the sequence $\{s_n\}$ is bound, the MC method gives an approximation to the integral and converges to the true value of the integral for $N \to \infty$. Therefore, the uncertainty estimate of the MC approximation becomes

$$\sigma(\hat{I}_X^{(N)}) = \frac{\sigma(f)}{\sqrt{N}} = \frac{s_N}{\sqrt{N}} \tag{1.5}$$

and decreases asymptotically for $N \to \infty$.

1.2 Importance sampling

The last section showed us that for infinite amount of sampling points, the MC approximation will eventually give an exact value for the integral. However, this is in general not satisfactory.

Firstly, it is interesting to look at the effect of adding additional dimensions to the problem. The MC method does not suffer from the curse of dimensionality directly; for an arbitrary amount of dimensions, the uncertainty of the approximation will be halved if the amount of sampling points will be increased by a factor of 4. Nevertheless, the problem with the coverage of the integration space stays. In order to reach a statistically significant statement about the integral, all features, including for example small peaks, have to be hit by the sampling. This requires sufficiently dense sampling. However, the density of the N sampled points falls with the number n of dimensions as $N^{\frac{1}{n}}$. This is however only relevant for functions which suffer from strongly varying features; a rather flat function can be safely approximated by an integral derived from a sparsely distributed points, thus not suffering from any problems by dimensionality.

Secondly, amplitudes in High Energy Physics can become hard to integrate, especially when they are of higher order or have many-particle final states or suffer from divergences (as an example, see [31]). This leads to a very costly evaluation of the function f, so that it is unfavourable to improve the integration precision by sampling more points, which leads only to an improvement proportional to $\sqrt{N}$. This favours other strategies instead, as their computational cost can be lower than of evaluating the integrand.

A more efficient approach for reducing the uncertainty is therefore to reduce the variance of f itself. This is one possible way of approaching the idea behind importance sampling. Modifying f by replacing it with f/g in such a way that it becomes similar to a constant function will decrease the variance, and it will become zero for f/g being constant function. This would turn the MC approximation exact. This can be realised by using a non-uniform probability distribution function $p(\mathbf{x})$ for the generation of the $\mathbf{x}_i$. The approximation then becomes

$$\hat{I}_{X,p}^{(N)} = \frac{1}{N} \sum_{i=1}^{N} \frac{f(\mathbf{x}_i)}{p(\mathbf{x}_i)} \tag{1.6}$$

This change of a sampling density results in a weighted integration by a Riemann sum with the measure $\mathrm{d}\mathbf{x}p(\mathbf{x})$. The idea of importance sampling is therefore to sample the evaluation points not with equal likelihood, but ideally proportional to the value of the integrand at the respective point. The sampling likelihood is therefore increased for high values of the integrand.

An useful measure for the success of flattening the integrand is, as mentioned earlier, the unweighting efficiency. When generating N events, that means, evaluating the integrand at N random points following the chosen distribution, one can take the average of its event weights, that means, the value of the integrand and compare it with the maximum event weight in the integration region:

$$\epsilon_{uw} = \frac{< w_N >}{w_{max}} \tag{1.7}$$

For a vanishing variance, the unweighting efficiency converges to 1, as a variance of 0 means that the function value is everywhere equal to its mean.

1.3 Change of variables

Importance sampling can be realised by transforming the initially uniformly sampled points through a variable transformation. We start by rewriting the form of the analytic integral in 1.1. The integral over $\mathbf{x}$ is transformed into an integral over $\mathbf{y}$, where we have $\mathbf{x} = h(\mathbf{y})$, such that $\mathbf{y}$ is sampled uniformly and $\mathbf{x}$ is sampled non-uniformly.

$$I = \int_\Omega \mathrm{d}\mathbf{y} \frac{\mathrm{d}h}{\mathrm{d}\mathbf{y}}(\mathbf{y}) f(h(\mathbf{y})) = \mathbb{E}_{Y\sim U(\Omega)}(\frac{\mathrm{d}h}{\mathrm{d}\mathbf{y}}(Y) f(h(Y)) \tag{1.8}$$

Now one can again write an estimator for this expectation value:

$$\hat{I}_Y^{(N)} = \frac{1}{N}\sum_{i=1}^{N} \frac{dh}{d\mathbf{y_i}} f(h(\mathbf{y}_i)) \tag{1.9}$$

Again, the expectation value of this estimator is the true value of the integral, and the uncertainty of the estimator follows now directly from 1.5:

$$\sigma(\hat{I}_Y^{(N)}) = \frac{\sigma(\frac{\mathrm{d}h}{\mathrm{d}\mathbf{y}}(Y) f(h(Y)))}{\sqrt{N}}. \tag{1.10}$$

From this it is clear that the desired choice of h is such that Jacobian $\frac{\mathrm{d}h}{\mathrm{d}\mathbf{y}}$ is proportional to the inverse of f, as the numerator would then account to zero. If we assume knowledge about the structure of f, then it will be possible to choose a variable transformation which reduces the variation at a low computational cost. Examples for this are the multi-channel approach [32] which optimises the computation for multiple peaks. However, the situation gets more difficult when one wants to develop an adaptive MC integration, which tries to approximate the integral consistently and fast without prior knowledge of its structure. For later use, we want to define a loss function which measures the quality of the computed variable transformation.

As it is our aim to reduce the uncertainty of the MC method, without taking into account how many points we include into the evaluation, it is reasonable to choose the function uncertainty as loss function:

$$L = \sigma \left(\frac{\mathrm{d}h}{\mathrm{d}\mathbf{y}}(Y, \theta) f(h(Y, \theta)) | Y \sim \Omega \right) \approx \hat{L}(\{\mathbf{y}_i\}) = \sigma \left(\frac{\mathrm{d}h}{\mathrm{d}\mathbf{y}}(Y, \theta) f(h(Y, \theta)) | Y \{\mathbf{y}_i\} \right). \tag{1.11}$$

Here, in a first step, parameters were introduced to the function h. This will be later the entry point for an optimisation procedure. In a second step, the loss function is approximated by not being evaluated over the whole integration space, but only over a finite set of points chosen randomly from it. With this loss function at hand, it is now possible to create an algorithm which optimises the variable transformation and by this the MC method.

1.4 The VEGAS algorithm

Before we describe how machine learning becomes relevant for importance sampling, it is important to also mention a very common adaptive algorithm whose performance will be used as a benchmark during this work.

The VEGAS algorithm ([8, 9, 33]) is an adaptive algorithm for MC methods which performs importance sampling by approximating the optimal sampling density. As discussed in the last section, the right choice of sampling distribution is such that it approximates the integrand as best as possible. In order to achieve this, the algorithm iteratively approximates the distribution with a step function with M steps, so that it is of the form $p(x) = 1/N\Delta x_i$. Each of this steps has a likelihood of $1/M$ of having a random number out of its bin being chosen as a sample. What is optimised during the process is the width of each of the steps, starting from equal widths (which gives a uniform distribution) towards small step widths for peaks and wide ones for regions of small contribution. The widths are optimised for each dimension separately.

In the one dimensional case, for each iteration, each of the M steps with width Δx_i is subdivided into $m_i + 1$ equal-sized subdivisions. m_i is determined by the ratio of step-width and mean of function value in this interval, such that steps with a big product of both are divided into smaller sections:

$$m_i = K \frac{\bar{f}_i \Delta x_i}{\sum_j \bar{f}_j \Delta x_j}. \tag{1.12}$$

K is, as well as M, a hyperparameter of the simulation. $\bar{f}_i$ can be computed by drawing samples out each step interval:

$$\bar{f}_i = \sum_{x \in x_i - \Delta x_i}^{x_i} |f(x)| \tag{1.13}$$

where x_i is the right endpoint of the step.

Now, not every step was divided into the same amount of subsets, instead those steps with a higher value of f have more substeps and by this a higher contribution. This reflects the necessity of approximating the optimal sampling, $p(x) = |f(x)|/\int f(x)$. In a last step, the original number of steps M is recovered by recombining groups of subsets of a fixed number of subsets. This means for example, that starting from the boundaries, the first three substeps are joined, then the next three, etc. The effect is, as expected, a change of step width. Using the evaluations of the integrand during the optimisation, an approximation of the variance can be calculated and used in order to determine if the desired precision has been reached.

The algorithm can be generalised to arbitrary dimensions. For this, the probability distribution will be factorised, one factor per dimension.

$$p(\mathbf{r}) = \prod_{j=0}^{D-1} p_j(r_j) \tag{1.14}$$

The algorithm is then being applied to each axis independently, and for D dimensions, the only change is that 1.13 is changed to

$$(\bar{f}_i)^2 = \sum_{x_i - \Delta x_i}^{x_i} \sum_{d_1,\ldots,d_{D-1}} \frac{f^2(\mathbf{r})}{\prod_{d_1,\ldots,d_{D-1}} p_d^2(x_d)} \tag{1.15}$$

such that the distributions of the other dimensions are taken into account and the sum goes over the grid of all over dimensions. All other steps stay identical.

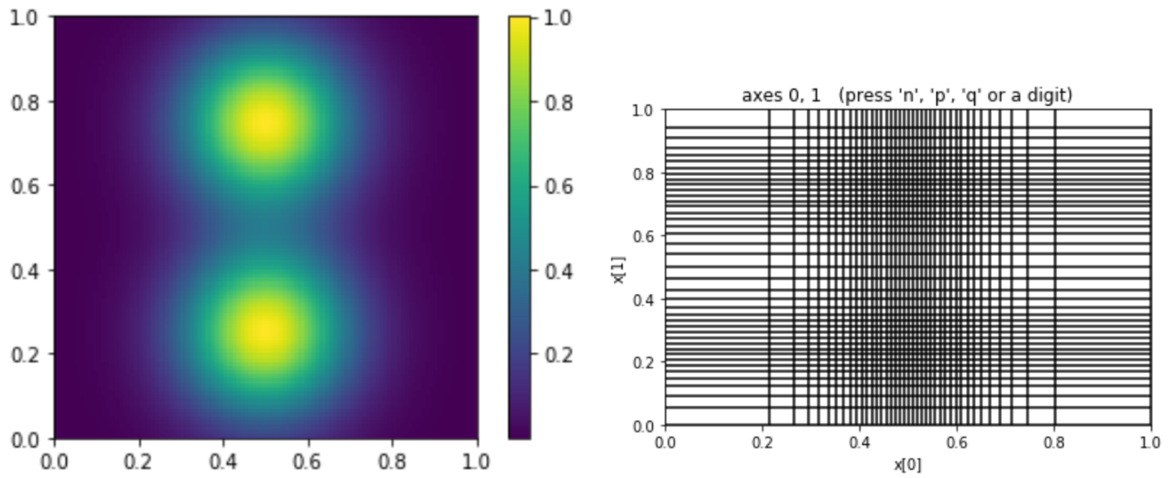


Figure 1.1 – The VEGAS grid for two aligned gaussian peaks.

The factorisation of the distributions is the great downside of the VEGAS algorithm. During the execution of the algorithm, when a fixed axis is optimised, it will see only the sum over all other dimensions, and thus will underperform if the true distribution is not factorizable. In a two-dimensional case with a Camel distribution (two Gaussian peaks), during the optimisation of the x and the y axis, both will see both peaks and decrease the step width at the x (and then y) coordinates of both of the peaks. If the peaks are aligned along an axis, this gives an

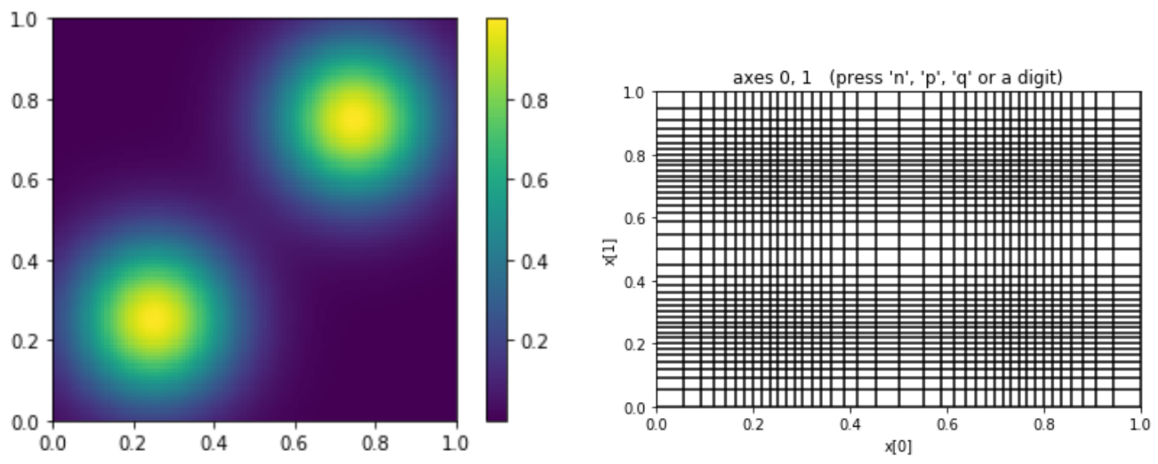


Figure 1.2 – The VEGAS grid for two non-aligned gaussian peaks.

excellent result, as can be seen in Figure 1.1. However, if the peaks are on a diagonal, it gives 4 regions with high resolution instead of 2, as can be seen in figure 1.2.

Also, VEGAS performs better when the structure of peaks itself is aligned to the coordinate axes. However, if one chooses a distribution with a high correlation between the coordinates (which is common in High Energy Physics applications), for example a combination of a peak at $x = y$ and additional structures in the integration domain, the optimisation will fail because the mapping on each of the axis will always show a peak. Thus, VEGAS will not be able to detect the additional structures and its efficiency will be very low, as a quasi-uniform sampling learned.

As a result, although VEGAS is a fast and highly parallelisable algorithm (for a recent implementation in Tensorflow, see [33]), it is not sufficient for many implementations and needs to be combined with other, problem specific algorithms, like multi-channelling. This is the motivation for looking into different approaches.

Chapter 2

Neural importance sampling

Adaptive importance sampling, such as the VEGAS algorithm, is a specific form of machine learning: the algorithm adapts the grid autonomously in order to reflect the peak structure of the integrand, which is the solution of an optimisation problem. Starting from this observation, it is natural to look into more modern ML techniques. Motivated from the success of neural networks in numerous other quantitative sciences, especially image generation and recognition, ML via neural networks became of great interest for the question of phase space integration, leading to multiple approaches trying to use them for this problem ([20, 21, 34–36]). One especially promising approach however is a neural importance sampling algorithm which does not use the NN in order to predict the sampling distribution itself, but instead predicts a set of parameters which determines its form [26]. This will be done with the use of normalising flows. In the following, after a short overview about NN, normalising flows and its core structure, the coupling cells, will be discussed in detail.

2.1 Basics of Neural Networks

Neural networks are a class of functions from $\mathbb{R}^n \to \mathbb{R}^n$ which are differentiable and general approximators. The latter means that the algorithmically learned functions are dense in the space of functions of interest, which follows from a wide range of universal approximation theorems for neural networks [37–39]. For our optimisation problem at hand, this means that integrand can be approximated and that this approximation can be found by gradient descent. The NN achieves this properties by being a composition of layers. Each layer is a mapping of the form

$$\mathbf{x}_{i+1} = \sigma(\mathbf{W}_i \mathbf{x}_i + \mathbf{y}_i) \tag{2.1}$$

where $\mathbf{x}_i$ is the output of the layer before and $\mathbf{x}_0$ is defined as the input data. The $\mathbf{W}_i$ and the $\mathbf{y}_i$ form together and affine transformation, referred to as a linear layer. σ is the a non-linear activation function.

The (artificial) neurons are the combination individual rows of the matrices which form the linear mapping and the activation functions. In the case that the output of all neurons of earlier layers affects the output of later layers and all neurons of the same layer are independent of each other, this is called a fully-connected NN. A counter example would be a NN for which the output of a neuron is set to zero, or equal to the value of another neuron of the same layer. A possible NN, with the activation functions included in the layers, is shown in 2.1.

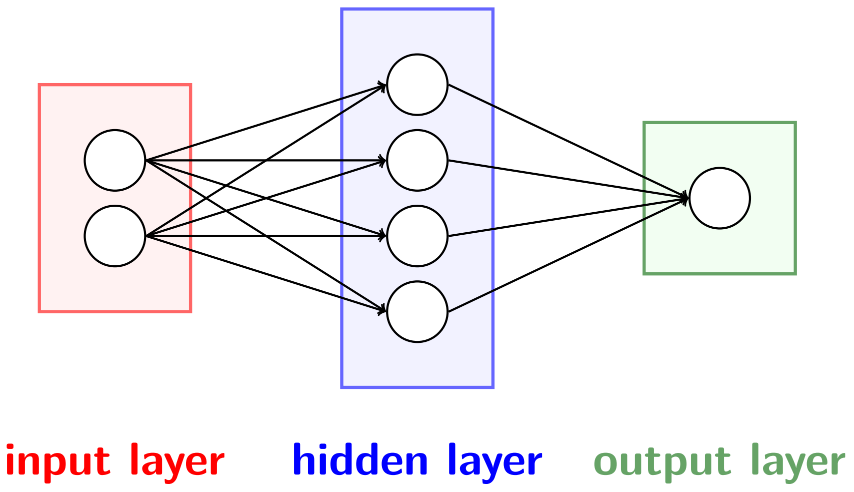


Figure 2.1 – A 2-layer fully connected NN (one hidden layer of 4 neurons and one output layer with 2 neurons), and three inputs. The white circles are the artificial neurons.

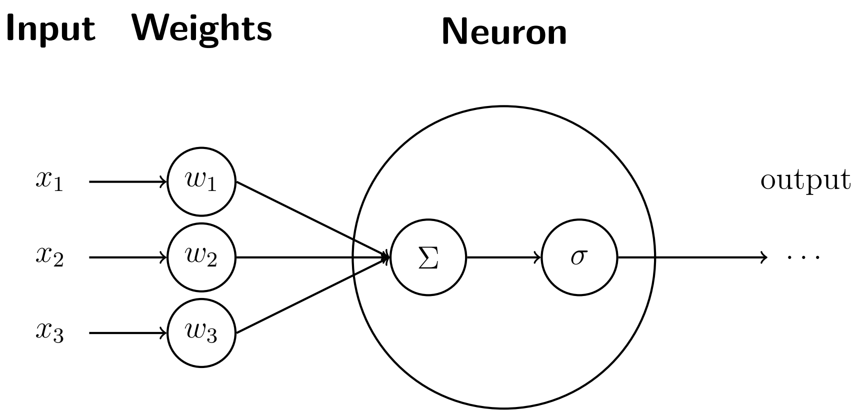


Figure 2.2 – A sketch of the structure of an artificial neuron with three weighted inputs and an activation function σ.

The number and the width of layers are essential for the performance of a NN. The more or the wider layers a NN uses, the more complex features of the relationship between the two data sets can be recognised in the transformation. This turns the choice of the number and size of the layers into an important design choice. Another important point of design is the choice of the activation function, for which many different possibilities exist. One of the most common is the ReLU function, which is zero for negative arguments and the identity mapping for positive ones. With this choice, the first layer of the mentioned NN would have the result $\mathbf{z}' = \max(0, \mathbf{W}_1\mathbf{x_0} + \mathbf{y_1})$, where the maximum-function is understood as being applied elementwise.

The aim is to optimise the different affine transformations $\mathbf{W}$ in order to minimise the "distance" between the approximation $\mathbf{y}'$ and the training set $\mathbf{y}$. In order to achieve this, one defines a loss function which represents this "distance". The loss function usually includes also a regulator proportional to the norm of the transformations, in order to disfavour overfitting [37]. The regulator is an additional additive term proportional to the L_1 or L_2 norm of the $\mathbf{W}_i$. This favours therefore small values for their entries and simpler models. In ML, this regulator prevents overfitting as it hinders the model from attempting to learn the background noise by learning a too complicated model.

This loss function is then differentiated after the transformations, giving a gradient. In order to do this computationally efficient using the chain rule, the gradient for each layer is implemented within the layer. As a result, one computes the gradients by calling the layers in a sequence, which is called backward pass or backpropagation [37] (the difference between backwards and forward comes from the fact that the gradient for the last layer is evaluated first). Suppose the value of the loss is L. The aim is to find the derivative of L after the weights of each linear layer, in order to modify these.

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial o_j}\frac{\partial o_j}{\partial w_{ij}}. \tag{2.2}$$

Here, j refers to the number of the layer and o_j is the output of layer j. The second factor can be derived from the structure of an artificial neuron:

$$\frac{\partial o_j}{\partial w_{ij}} = \frac{1}{\partial w_{ij}} n_j(x_{j-1}) = \frac{1}{\partial w_{ij}} \sigma\left(\sum_k w_{kj}x_{j-1,k} + w_{0k}\right) \tag{2.3}$$

The derivative can be determined using the chain rule, as long as the derivative of the activation function σ is known. The first factor in 2.2 can be only derived directly if j is the output layer and depends on the actual form of the loss function, as o_j is then an entry of the prediction $\mathbf{y}'$. If j is an hidden layer, the loss can be seen as a function of the n_z who all receive o_j:

$$\frac{\partial L(o_j)}{\partial o_j} = \frac{\partial L(n_1, n_2, \dots)}{\partial o_j} = \sum_z \left(\frac{\partial L}{\partial n_z}\frac{\partial n_z}{\partial o_j}\right) \tag{2.4}$$

Now, the derivative with respect to o_j can be calculated if the derivatives after the n_z are known, which is again the case for the output layer only. Therefore, one starts the backpropagation by calculating the derivatives with respect to the parameters of the output layer, and uses them for the gradient descent on these parameters. For the next layer, the gradient is now dependent on the value of the derivative of the loss after the output times the derivative of the output layer after the current layer, and so on. This can be performed efficiently if during the forward propagation, the necessary information for the backward propagation, e.g. the derivatives after input, are stored in order to save computational steps.

The last step is then a gradient descent, for which many different algorithms exist in order to create a stable and fast training. The simplest of these is the stochastic gradient descent (SGD) [40]. For each iteration i, the parameters $\mathbf{w}$ are adjusted as:

$$\Delta\mathbf{w} = \eta\nabla L_i(\mathbf{w}) \tag{2.5}$$

Here, the loss is calculated for a finite set of sampled points, so that the gradient descent step is only an approximation of the correct step. This process is called stochastic because after a finite amount of steps, a new sample is drawn. Refreshing the sample therefore compensates for the finite size of the set. η is the learning rate and determines how strongly the gradient is weighted.

After the training period is finished, the forward pass can be used to map any new vector $\tilde{\mathbf{x}}$ of the right dimensions in order to predict $\mathbf{y}'$. If the correct $\mathbf{y}$ is known, the precision and loss can be determined. In our application at hand, we would naively want to learn the distribution function which is realised by f, in order to improve the integration of f.

In order to be of practical use, two conditions have to be met: Our first requirement is that it should be possible to evaluate the mapping of the NN fast in comparison to the evaluation of f, which is true for sufficiently simple NNs. The second condition is related to the question how to train the NN. Other than for common use cases like image labelling, it is necessary to evaluate the integrand in order to determine the loss. This can be expensive. Therefore, training the model itself ("forward training") by mapping the uniform sampled points through the model and evaluating the function on the output is unfavourable.

Instead, one can precalculate a grid of function values with their respective arguments in the target space. These arguments will be not necessarily uniformly distributed. Now, the training is performed by applying the inverse mapping on the points in target space, providing the Jacobian which is used together with the function value in the loss. This training strategy is equivalent to the forward training, requires however the inversion of the model for sampling, which would be non trivial when the NN is used in order to learn the integrator directly. This training strategy is referred to as "backward training".

This is the reason why trying to learn the integrand directly with the NN is not favourable. We will take another approach instead.

2.2 Normalising Flows

Although normalising flows are of great use in ML due to the common need for modelling probabilistic functions, the concept is older and of much wider use than for what we are interested in. In general, normalizing flows are used to express a transformation on a sample of a fixed distribution which returns a data vector:

$$\mathbf{x} = T(\mathbf{u}) \quad u \sim p_u(\mathbf{u}) \tag{2.6}$$

Normalizing flows [23] are such a transformation which is also a diffeomorphism, i.e. invertible with both the transformation and its inverse being differentiable. Thus, the distribution of $\mathbf{x}$ becomes:

$$p_x(\mathbf{x}) = p_u(\mathbf{u})|\det J_T(\mathbf{u})|^{-1} \quad \mathbf{u} = T^{-1}(\mathbf{x}) \tag{2.7}$$

where J_T is the Jacobian of the transformation. and p_x is a probability density function. This follows from integrating:

$$\int_\Omega \mathrm{d}\mathbf{u} p_u(\mathbf{u}) = \int_{T(\Omega)} \mathrm{d}\mathbf{x} p_u(T^{-1}(\mathbf{x}))|J_T(T^{-1}(\mathbf{x}))|, \tag{2.8}$$

as the integral over the probability density is preserved. The transformation can also carry additional parameters. As a normalizing flow is a diffeomorphism, it is also composable, which will be of crucial importance to us. The composition will lead to a general form of

$$p_x(\mathbf{x}) = p_0(\mathbf{u}_0) \prod_{i=1}^{N} |\det J_{T_i}(\mathbf{u}_{i-1})|^{-1} \tag{2.9}$$

where $\mathbf{u}_i = T_{i-1}(\mathbf{u}_{i-1})$. We will use the normalizing flows in order to build the mapping from the latent to the target space, as they fulfil our requirement of being invertible and preserve the integration volume. We need to find the form of the normalizing flows which will define the variable transformation, as well as their Jacobian, such that the loss is minimised. Complex transformations via normalizing flows can be build as a composition of more simple ones. This is mirroring the layer architecture we met earlier for NNs, but also the sequential structure of the training process.

We saw now that it is desirable to express the transformation via normalizing flows. It is now necessary to determine the normalizing flows in dependency of the integrand. This can be achieved with machine learning.

2.3 Coupling cells

Let us now take a look at the algorithm behind neural importance sampling, following the description from [26]. The central element of this are the coupling cells, which use the NN not to learn the distribution but to learn parameters of the normalizing flows. This element was developed for the NICE (Non-linear independent component analysis) algorithm [24, 25]. A coupling cell takes the incoming vector $\mathbf{x}$ and splits it in two parts, $\mathbf{x}^A$ and $\mathbf{x}^B$. This process is

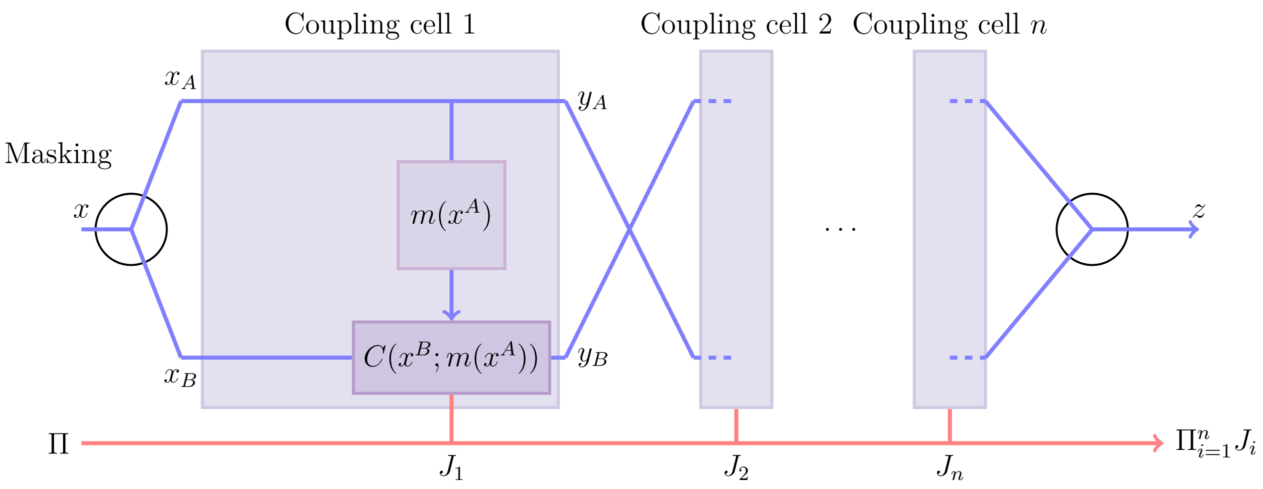


Figure 2.3 – Schematic structure of a coupling layer.

referred to as masking. A and B mark a collection of indices a and b. The coupling cell defines a mapping; for the first set of indices, this is the identity: $\mathbf{x}^A = \mathbf{y}^A$. The vector $\mathbf{y}^B$ will be defined as the output of a set of separable, invertible functions $C_b(\boldsymbol{\nu}_b, x_b)$, one per element of $\mathbf{x}^B$. These $\boldsymbol{\nu}_b$ are crucial: They are determined by the NN m, which takes $\mathbf{x}^A$ as input data. The NN tries to learn the probability density of x_b depending on the values of $\mathbf{x}^A$, but without any knowledge of the other values in $\mathbf{x}^B$. Multiple coupling cells which transform different dimensions form together a coupling layer. This process is also drawn schematically in Figure 2.3. As mentioned, we desire to have an invertible transformation. Therefore, we demand that the coupling transform C is an invertible map. This gives:

$$\begin{aligned} \mathbf{y}^A &= \mathbf{x}^A \\ y^{b_1} &= C^{b_1}(m(\mathbf{x}^A), \mathbf{x}^{b_1}) \\ &\vdots \\ y^{|B|} &= C^{|B|}(m(\mathbf{x}^A), \mathbf{x}^{|B|}) \end{aligned} \tag{2.10}$$

and its inverse

$$\begin{aligned} \mathbf{x}^A &= \mathbf{y}^A \\ \mathbf{x}^B &= C^{B\ -1}(m(\mathbf{y}^A), \mathbf{y}^B). \end{aligned} \tag{2.11}$$

Naively, this can result in a slowdown - as a part of the data is not mapped, it will be necessary to use multiple coupling layers and train multiple instances of the neural network. However, this comes with a great advantage: The Jacobian of the coupling layer has an especially simple structure:

$$\left(\begin{array}{c|c} \begin{matrix}1 & & 0\\ & \ddots & \\ 0 & & 1\end{matrix} & 0 \\ \hline \frac{\partial C^B(\boldsymbol{\nu}^B,\mathbf{x}^B)}{\partial x^A} & \begin{matrix} \frac{\partial C^{b_1}(\boldsymbol{\nu}_{b_1},x_{b_1})}{\partial x_{b_1}} & & 0 \\ & \ddots & \\ 0 & & \frac{\partial C^{b_B}(\boldsymbol{\nu}_{b_B},x_{b_B})}{\partial x_{b_B}}\end{matrix}\end{array}\right) \tag{2.12}$$

This saves us from determining the complex derivatives of the NN, as they do not contribute to the determinant of the Jacobian:

$$\det J = \prod_{b\in B} \frac{\partial C^b(\boldsymbol{\nu}_b(\mathbf{y}^A), x_b)}{\partial x_b} \tag{2.13}$$

Therefore, the determinant of the Jacobian of each of the coupling cells is efficient to evaluate and can be passed on to the next coupling cell, in order to generate a Jacobian of the complete transformation. This reduction of the computation cost for the determinant is the key feature of the coupling cells. Without this, the algorithm would be very expensive for high dimensions. Now, it is also possible to include complex coupling transforms and NN without suffering from difficulties during the computation of the Jacobian. Looking at the definition in the last section, we see that each of the C^b together with id^A form a normalizing flow, as they are diffeomorphisms. They can be composed and define the change in the transformation from an uniform one to the desired approximation of f. As we act on a hypercube, the derivatives of C can be seen as probability distributions, whereas C itself is a cumulative probability distribution. The reason for this is that C, in order to be invertible, has to be monotone, which is the property of a cumulative distribution. The distribution itself, as seen earlier, transforms proportional to the Jacobian.

2.3.1 Number of coupling cells

An important design choice is the number of coupling cells, which is dependent on the size of the input data vector. In general, one wants the different coupling cells to mask the data points such that every coordinate can influence the transformation of all other points. For a two-dimensional vector entering, 2 cells would be therefore sufficient, and for $D = 3$, we will need $n_{CC} = 3$. A reasonable choice for the minimum number of needed coupling cells is $n_{CC} = 2\log_2(D)$. The argument behind this is the following. Each dimension is either transformed or not in each

Dimension	0	1	2	3	4	5	6	7
Transformations	0	0	0	0	1	1	1	1
Transformations	0	0	1	1	0	0	1	1
Transformations	0	1	0	1	0	1	0	1

Table 2.1 – Determination of the transformed dimensions for each coupling cell

coupling cell. Therefore, one expresses all numbers from 0 to $D-1$ in binary representation. In order to do so, $\log_2(D)$ bits are required. Now one goes at the same time through the most significant bit of all numbers; this bit is for each number either valued "1" or "0". By this, two coupling cells are defined, one in which each dimension with a "1" in the most significant bit will be transformed and the "0" valued dimension will be preserved, and the opposite, in order to treat all dimensions equally. This can then be repeated for the second most significant bit, and so on, giving $2\log_2(D)$ cells, as the number of significant bits needed in order to express D is $\log_2(D)$. This guarantees that no set of 2 or more cells is always transformed or preserved always together. A formal theorem stating that all correlations can be expressed with this number of coupling cells can be found in [30].

Table 2.1 can be used to demonstrate this process for $n = 8$. Vertically, the bit representation of each of the dimension labels is written, from the most significant to the least significant bit. Now, the first digit for each dimension is taken, that is 0 until dimension 3, and 1 for the rest. Therefore, the first coupling cell will transform $\{4, 5, 6, 7\}$ in dependency of $\{0, 1, 2, 3\}$. The second will do the inverse, that means, transform the first 4 dimensions. The next set of bits will transform $\{2, 3, 6, 7\}$ in dependency of the other dimensions, and so on. Therefore, six coupling cells are needed.

One needs to define which form the coupling transforms C^B should have. It is sufficient to use relatively simple structures, as the nonlinear correlations between the uniform and the target distributions are handled by the NN. Preferably, the coupling transforms should be easily invertible. There are many options to choose from, with varying complexity and expressivity.

2.3.2 Affine Coupling Transforms

The most simple option is probably to define the transformation as [25]

$$\mathbf{y}^B = C^B(\mathbf{x}^B, \boldsymbol{\nu}^B) = \mathbf{x}^B \odot e^{\mathbf{s}^B} + \mathbf{t}^B. \tag{2.14}$$

The $\odot$ marks elementwise multiplication. Both $\mathbf{s}^B$ and $\mathbf{t}^B$ are generated by $m(\mathbf{x}^A)$. Here, the parameters of the transformation are then:

$$\boldsymbol{\nu}^B = (\mathbf{t}^B, \mathbf{s}^B) \in \mathbb{R}^{2,|B|}. \tag{2.15}$$

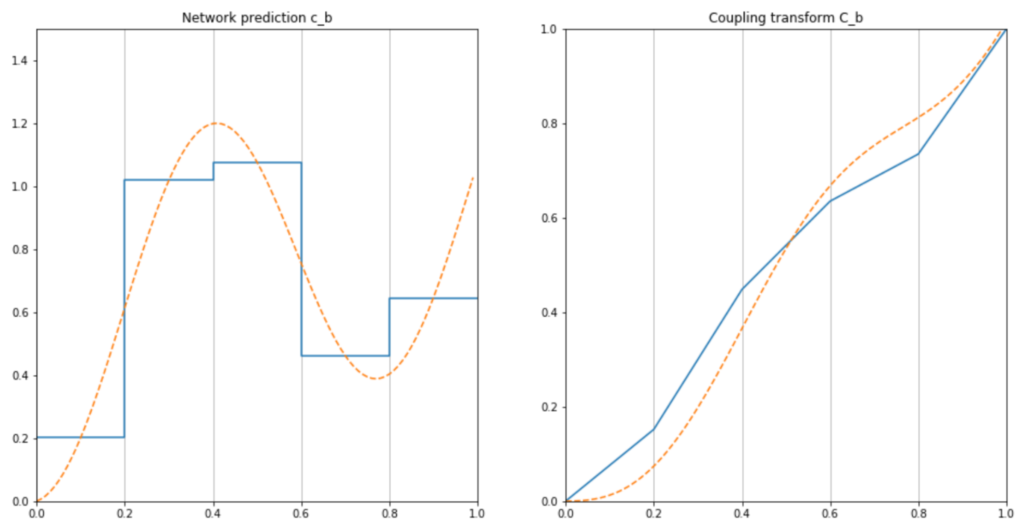


Figure 2.4 – A demonstration of the mapping for piecewise-linear cells in 1D. The orange dotted line is the function to be learned, the blue line is the network prediction/the coupling transform.

2.3.3 Piecewise-Linear Coupling Transforms

An alternative is to use piecewise-linear coupling transformations [26]. Its derivative will be then a piecewise-constant function, as can be seen in figure 2.4. In this setup, the NN determines then $\boldsymbol{\nu}(\mathbf{x}^A)$, which is the vector of the slopes of the piecewise-linear function, or the bin height of its derivative. For this, each of the dimensions in B is divided into K bins. Therefore, the NN m predicts a $|B| \times K$-matrix $\hat{\mathbf{Q}}$, which is then normalised to the matrix $\mathbf{Q}$ by applying the softmax-function

$$s(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{n=1}^{N} e^{z_n}}, \quad j = 1, \dots, N \tag{2.16}$$

to each of the rows. Therefore, the derivative of the coupling transformation in the b-th dimension of $\mathbf{x}^B$ is then defined as $c_b(x_b) = Q_{bz}/w$, where w is the bin width and z is is the number of the bin which contains the value of the argument x_b.

The coupling transform itself becomes then

$$C_b(x_b, Q) = \int_0^{x_b} c_b(t)dt = \alpha Q_{bk} + \sum_{s=1}^{k-1} Q_{bs} \tag{2.17}$$

with α being the relative position of x_b in the bin k, $\alpha = Kx_b - \lfloor Kx_b \rfloor$. The Jacobian of this transformation does not require any additional calculation; it can be directly computed using 2.12:

$$\det\left(\frac{\partial C(\mathbf{x}^B, Q)}{\partial (\mathbf{x}^B)^T}\right) = \prod_{b=1}^{|B|} c_b(x_b) = \prod_{b=1}^{|B|} \frac{Q_{bx}}{w} \tag{2.18}$$

with x again being the bin containing the value in the respective dimension.

More complex choices for the coupling functions will in consequence need more parameters being determined by the NN.

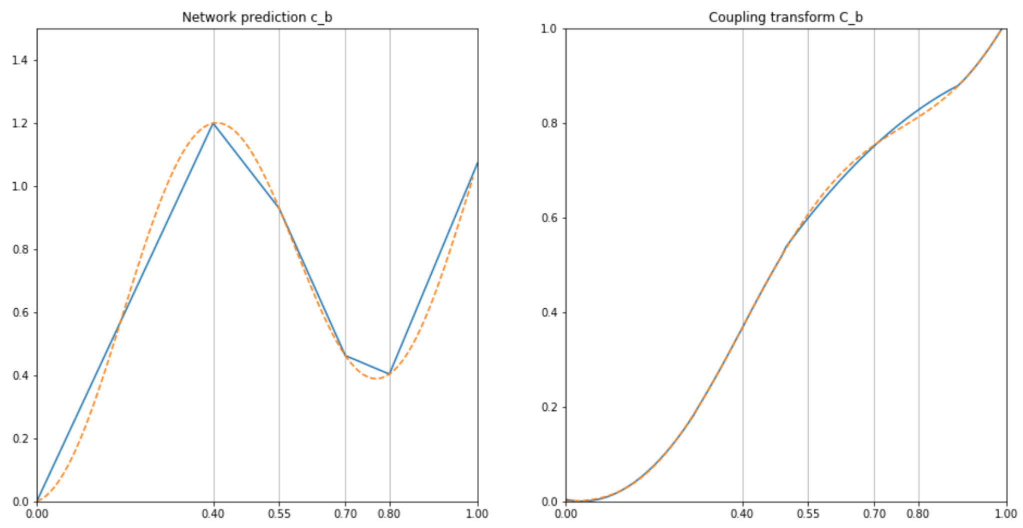


Figure 2.5 – A demonstration of the mapping for piecewise-quadratic cells in 1D. The orange dotted line is the function to be learned, the blue line is the network prediction/the coupling transform.

2.3.4 Piecewise-Quadratic Coupling Transforms

The piecewise-linear coupling transforms suffer from the disadvantage that their bin width is fixed. Each dimension is divided in K bins with equal width. Although it would be possible to have K bins with different widths, this would still be fixed as NN only predicts the height of the bins and not the relative position of the bin boundaries towards each other. As a consequence, a higher number of bins is required in order to have a good resolution for sharp peaks. Including the bin widths besides the bin heights as parameters predicted by the NN leads to quadratic splines, which can greatly improve the adaption for a fixed number of bins, as can be seen by comparing figures 2.4 and 2.5.

Using the same notation as before, the output of the NN are the $|B| \times K$-matrix $\hat{\mathbf{W}}$ containing the bin widths and the $|B| \times (K+1)$-matrix $\hat{\mathbf{V}}$ containing the height of the spline knots. These outputs also have to be normalised. For the bin widths, this can be done by the softmax function. For the vertices, on the other hand, it has to be assured that the transformation represented by the cumulative distribution function predicted by this network is limited to 1; therefore the normalisation has to be modified to

$$V_{i,j} = \frac{\exp\left(\hat{V}_{i,j}\right)}{\sum_{k=1}^{K} \frac{\exp\left(\hat{V}_{i,k}\right)+\exp\left(\hat{V}_{i,k+1}\right)}{2} W_{i,k}} \tag{2.19}$$

Now, the derivative of the coupling transformation, which gives rise to the Jacobian, is not piecewise constant but the piecewise linear interpolation between the $(W_{i,j}, V_{i,j})$-tuples. This linear interpolation exists as efficient implementations in many programming languages. The values of these interpolations at x^b have to be then multiplied for each transformed dimension.

The coupling transform itself is then a quadratic polynomial of the normalised output, dependent on the position of transformed coordinate in the predicted bin with number z, $\alpha = (x_b - \sum_{k=1}^{z-1} W_{bk})/W_{bz}$:

$$C_b(x_b, W, V) = \frac{\alpha^2}{2}(V_{bz+1} - V_{bz})W_{bz} + \alpha V_{bz} W_{bz} + \sum_{k=1}^{z-1} \frac{V_{bk} - V_{bk+1}}{2} W_{bk} \tag{2.20}$$

With this theoretical basis at hand, one can implement the coupling cells and gain some insight on their behaviour.

Chapter 3

Neural importance sampling in two dimensions

As a first step, the piecewise-linear and piecewise-quadratic coupling cells are implemented for the two-dimensional case, in order to evaluate their capabilities. The two-dimensional case has the advantage of a simple visualisation, fast simulation and reduced effects of stacking of coupling cells.

3.1 Implementation

Neural importance sampling is implemented in PyTorch [41], a modern Python package for machine learning. This has the advantage of providing pre-implemented layers and optimizers as well as automatised back-propagation. PyTorch code can also be executed on GPUs with only minor adjustments, which allows a considerable speedup for vectorized computations, such as NN evaluation.

The structure of the neural importance sampling package, sketched in figure 3.1, is the following: the Manager class provides the basic initialisation and training functions. Both the implementation for piecewise-linear and quadratic coupling cells share the same BasicManager, but have different implementations of the *create_model* function. During the execution of this function, the model of the manager (inheriting of PyTorch's Module class) is constructed as a sequential model, consisting of coupling cell and masking layers. The number and form of these layers are dependent on the arguments of the dimension of the model as well as the number of coupling cells. Another argument is the structure of the rectangular neural network (see figure 3.2), of which each of the coupling cells will create an independent instance. Unless stated otherwise, the rectangular NN has the following structure: it consists of sequences of a linear layer, a ReLU layer and a batch normalisation layer, where all hidden linear layers have the same number of nodes. Batch normalisation layers rescale and shift the output of the previous layers,

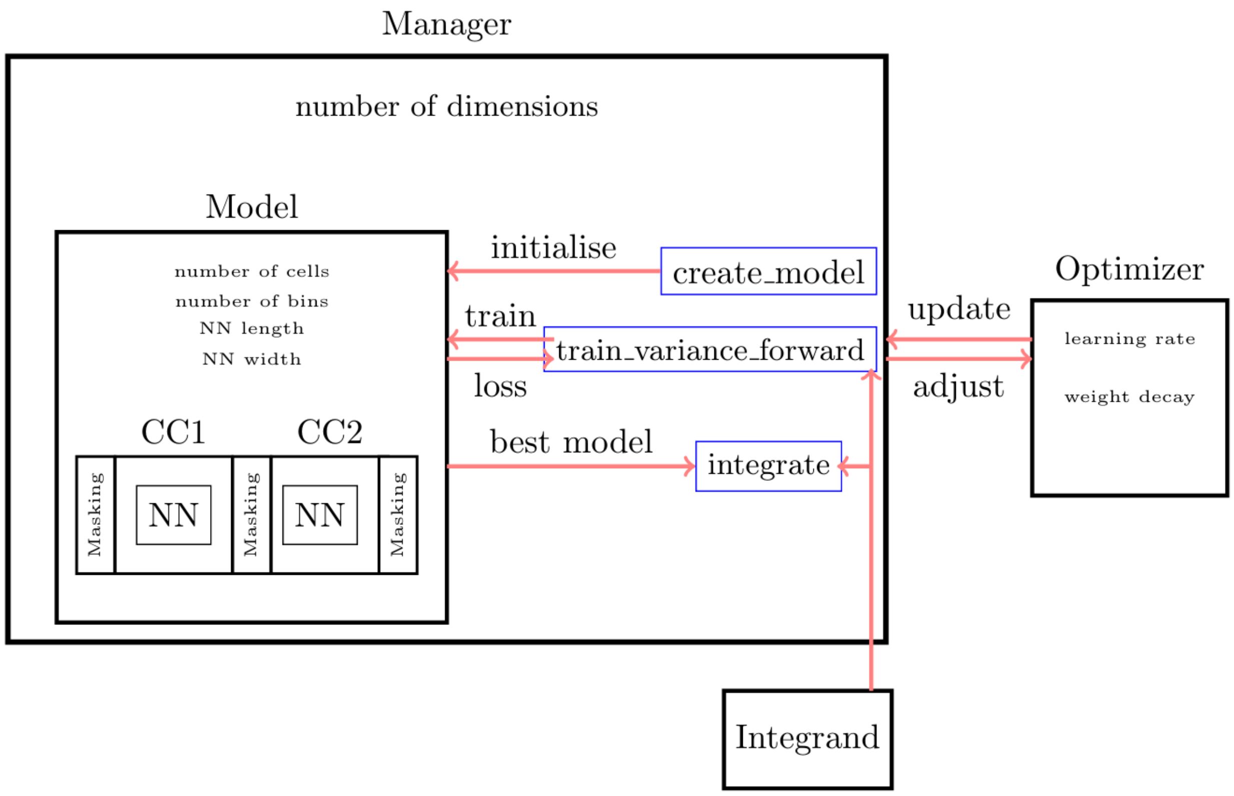


Figure 3.1 – A sketch of the structure of the code for the case of 2 coupling cells. Black are classes with their most important properties, blue are methods and the red arrows indicate their relations.

which has been shown to greatly improve the stability and quality of NNs and is crucial for good results with this algorithm [42]. A more detailed overview of the most important classes and functions can be found in the appendix.

During the forward pass, the masking layers reshuffle the dimensions of the input based on the strategy of section 2.3.1. This is done for a whole batch (a collection of sampled n-dimensional random points) at once. The coupling cell then evaluates the NN on x^A and performs the transformation described earlier, whilst calculating the Jacobian at the same time. During the training, the forward pass is performed in a loop. The function to be learned is executed with the transformed batch as argument. From this, an estimate of the variance when using the model as a sampler is calculated. Following 1.11, we want to calculate the variance of the product of the function value and the value of the Jacobian at this point and define this as the loss. This loss is however only available as an estimate. Ideally, we would calculate the variance over the sample as

$$L' = \int \mathrm{d}x g(x) \left(\frac{f(x)}{g(x)}\right)^2 - \left(\int \mathrm{d}x g(x) \frac{f(x)}{g(x)}\right)^2 = \int \mathrm{d}x \frac{f(x)^2}{g(x)} - I(f)^2 \tag{3.1}$$

However, during runtime, the integral value is only known in approximation, as the batch size is limited. The integral is however independent of the parameters of the normalizing flows.

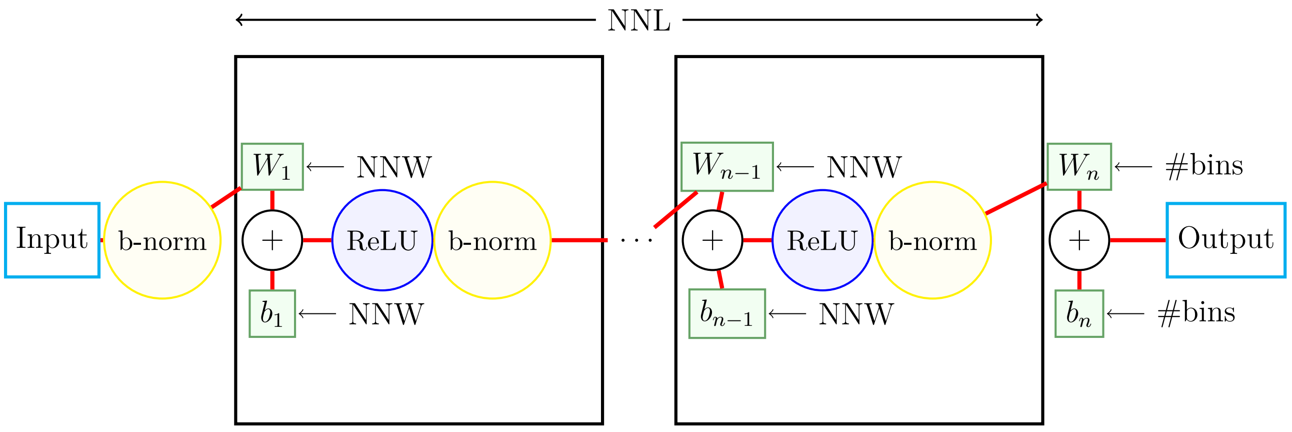


Figure 3.2 – The structure of the rectangular neural network used for the coupling cells. Each block refers to one of the repeating units. The number of these units is meant with the length of the NN. The width refers to the shape of the linear transformation. The shape of the last linear transformation is set by the number of parameters required by the coupling cell. The red line symbolises the flow of calculation.

Therefore, it is a reasonable choice to take for the loss instead

$$L = \int \mathrm{d}x \left(\frac{f(x)^2}{g(x)} \right) \tag{3.2}$$

Now, changing again the perspective back to the variable transformation picture by defining $g(x) = \left(\frac{\partial h}{\partial x}(x,\theta)\right)^{-1} = J^{-1}$ and integrating over $y = h(x,\theta)$ gives the definition of the loss function:

$$L = \mathbb{E}_{\mathbf{Y}\sim\mathcal{N}(\{\mathbf{x}\})|x_i\in\Omega}(f(\mathbf{Y})J)^2, \tag{3.3}$$

where $\mathcal{N}$ denotes the effect of the coupling cell transformations. This is the loss of the forward training, which is used in the following. The motivation behind this is to examine the trained model directly, whereas runtime is not the main focus of this investigation as it largely depends on the used hardware and the implementation and form of the integrand.

After the loss has been calculated, PyTorch's AUTOGRAD implementation can automatically calculate the gradient and by this optimise the parameters of the NNs. This is done, if not otherwise stated, using PyTorch's implementation of the Adamax gradient descent [43]. For every step (epoch), a new batch of random points in the hypercube is sampled. Every time the loss reaches a new absolute minimum, the corresponding model is preserved. This is continued until a breaking condition is met.

Figure 3.3 shows two typical behaviours of the loss during the training. Ideally, the loss falls evenly for most of the training time, until it saturates and stays virtually constant. If on the other hand the optimizer is not well calibrated on the training architecture and the integrand, the loss shows a greater variance. It reduces faster but does not necessarily reach the same optimum. However, after reaching a minimum it might increase again. Due to this behaviour

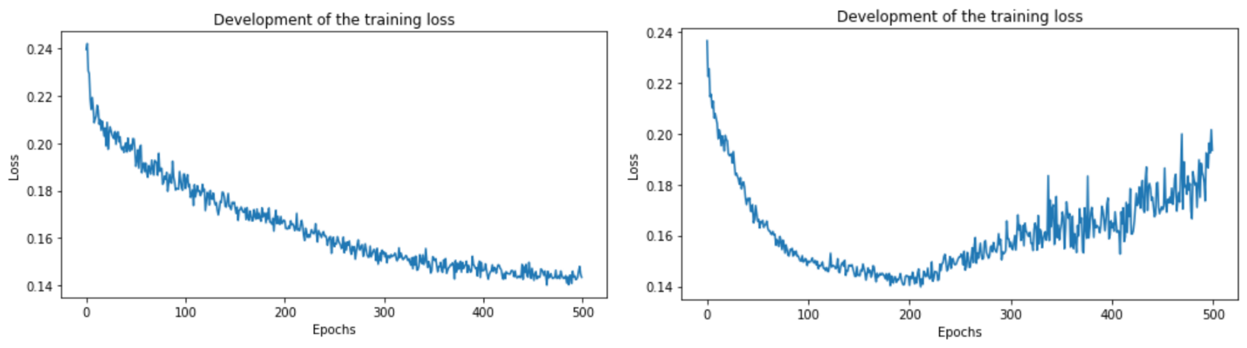


Figure 3.3 – Stable (left) and instable (right) behaviour of the loss during the training

and due to the computational cost of updating the model, it is not desirable to train during the whole integration but only for a limited time in the beginning, until an optimal model is reached. In order to account for this, early stopping regularization [44] is used. This means that the training is stopped when the loss increases for multiple subsequent epochs or exceeds a certain threshold. The best model during the training can be then used. Additional criteria for stopping include a test if the minimum loss has not or not significantly been reduced for too many epochs and eventually ending the training if it exceeded a certain runtime.

With this setup, the next step is to determine the right values of the hyperparamters. These include the length and width of the rectangular NN. On the side of the optimizer, there are the learning rate which determines how strong the gradient descent is, as well the weight decay, which is a penalty for the norm of the linear layers and prevents overfitting. As it limits the norm of the linear transforms, it gives also additional stability to the training. Figure 3.3 shows the effect of too small values for the weight decay on the right side. Apart from this, there are the batch size and the number of bins of both coupling cells. These hyperparameters are in general dependent on the function which one aims to learn; they can be however be also limited by hardware, as it is with the batch size, which is limited by the memory of the used device.

These hyperparameters are costly to find. In general, there is no analytic way to predict the optimal value, and it has to be determined experimentally. For this, a range for each of the hyperparameters is defined and fixed combinations of random parameters are tested. The optimal configuration is then used.

In the following, the performance for the optimal configuration after hyperparameter optimisation is compared for both the piecewise-linear and the piecewise-quadratic coupling cells. The motivation behind this is to see which of both can achieve in its optimal configuration the greater improvement of the variance.

The reason that affine coupling cells are not investigated deeper lies in their specific behaviour. From the definition of the affine coupling cells, it becomes clear that they map into the full $\mathbb{R}^n$, other than the piecewise-linear and piecewise-quadratic coupling cells. This requires mapping their output back into the hypercube, which causes a bad initialisation and in return makes

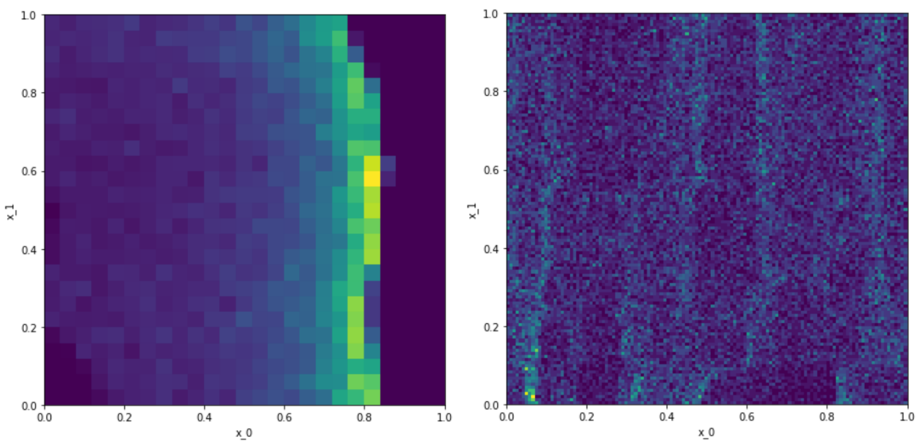


Figure 3.4 – The untrained mapping of the affine (left) and piecewise-linear (right) coupling cell, for a single cell in 2 dimensions. The peaked behaviour of the untrained affine coupling cells reduces the ability to correctly train features.

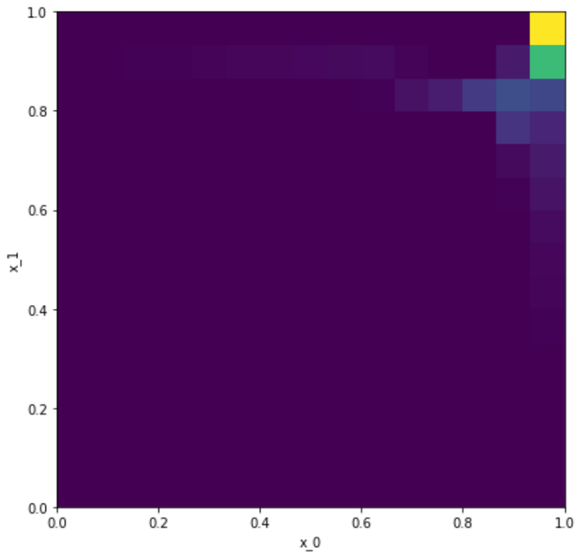


Figure 3.5 – The mapping of the affine coupling cell after training on a gaussian. A wrong local minimum of the loss was found, referring to a vanishing Jacobian.

training harder. An example of disadvantageous initialisation can be seen in 3.4, which also shows that other coupling transforms do suffer less from this. For this figure, the transformed dimension x_1 was, after the transformation via the coupling cell happened, mapped back to [0,1] using the arctangent. The arctangent is very steep around 0, resulting in a low likelihood of sampling into small values of the initialised model. On the other hand, it converges to 1 for $x \to \infty$, mapping many sampled values in a small area close to 1. Additionally, wrong convergence can also be a result of the necessity of including the Jacobian of the mapping via the arctangent, of which the derivative converges to zero for $x \to \infty$. Here again, the training could lead to an increasing affine factor which minimises the Jacobian and by this the loss, but does not improve the integral as the mapping for all dimensions will be ~ 1. An example for such a training result is shown in figure 3.5. The other two presented coupling cells do not suffer from this issues and are therefore more promising.

3.2 Comparison of piecewise-linear and piecewise quadratic coupling cells

Both coupling cells were optimised by two independent random hyperparameter searches narrowing down the optimal set of hyperparameters. The training was limited to 1000 epochs or 15 min of runtime, which was however never reached during the hyperparameter optimisation. As it was observed that during the training, both cells usually reached a minimal loss before diverging in loss again, the training was stopped when the loss exceeded 175% of the minimal loss observed in the training. The function which was learned was

$$f(\mathbf{x}) = e^{((\mathbf{x}-0.75)^2/0.2^2)} + e^{((\mathbf{x}-0.25)^2/0.2^2)} \tag{3.4}$$

This function has two gaussian peaks in the the two-dimensional plane at different positions, and is nicknamed the "camel"-function. This function is of interest as it can test whether the coupling cells are able to learn non-central peaks along a diagonal.

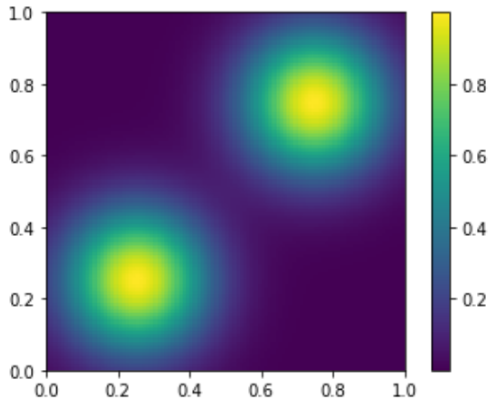


Figure 3.6 – Plot of the "camel" function

A second test function is used to determine how the training deals with regions of low value which are enclosed in regions of high values.

$$f(\mathbf{x}) = \min(1, e^{-\frac{(|\mathbf{x}-0.5|-0.3)^2}{0.1^2}} + e^{-\frac{(x_0-x_1)^2}{0.1^2}}) \tag{3.5}$$

The ceiling value of one is not a necessity for the success of the training and is used for cosmetic reasons of the the plot.

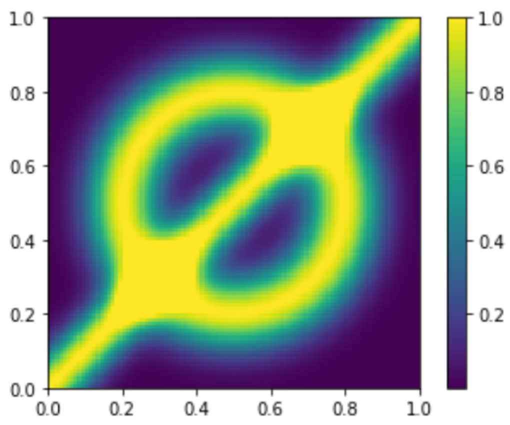


Figure 3.7 – Plot of the slashed circle function

It is important to note that the results given below are subject to statistical variations. The training with NN is not deterministic but stochastic; if one trained twice with the same hyperparameters and the same data, it would not necessarily lead to the same parameters. The mapping by the NN after the training is however deterministic as it can be represented by a function.

3.2.1 Performance of the piecewise-linear coupling cell

The optimal configuration for the piecewise-linear case was reached with 2 coupling cells with 19 bins each. Each coupling cell had a rectangular NN with width 13 and length 3, as well as a learning rate of 9.4×10^{-3} and weight decay of 7.8×10^{-4}. The batch size was 76000. The data reported in table 3.1 is the average over 10 runs. The minimal variance is the best variance reached during the training, which is usually reached shortly before the training gets terminated due to early stopping regularisation. The relative minimal variance is the ratio between the minimal and the initial variance. The speedup refers to the reduction of function evaluations necessary to reach a certain standard deviation of integration result. Following chapter 1, if one would use the trained model for sampling instead of uniform sampling, the number of required function evaluations would reduce by the inverse of the relative minimal variance. The reported time depends on the function which is evaluated and refers to the time needed to reach the minimal variance.

	Minimal variance	**Rel. minimal variance**	**Speedup**	**# fct evaluations**	**Execution time/s**
Double Gaussian, best	$4.07 \pm 0.38 \times 10^{-3}$	$4.84 \pm 0.48 \times 10^{-2}$	$2.06 \pm 0.21 \times 10^{1}$	$9.53 \pm 0.57 \times 10^{6}$	178 ± 36
Double Gaussian, 2nd best	$5.01 \pm 0.57 \times 10^{-3}$	$6.26 \pm 0.72 \times 10^{-2}$	$1.60 \pm 0.18 \times 10^{1}$	$4.80 \pm 0.72 \times 10^{6}$	77 ± 21
Slashed circle, best	$6.42 \pm 0.36 \times 10^{-3}$	$1.312 \pm 0.076 \times 10^{-1}$	7.69 ± 0.45	$9.09 \pm 0.95 \times 10^{6}$	105 ± 15

Table 3.1 – Performance of the piecewise-linear coupling cell setups.

A similar result was also reached with a different configuration (NN width 14, NN length 7, 19 bins, learning rate 7.0×10^{-3}, weight decay 7.2×10^{-4}, batch size 20000), but reduced training time due to the smaller batch size and higher learning rate.

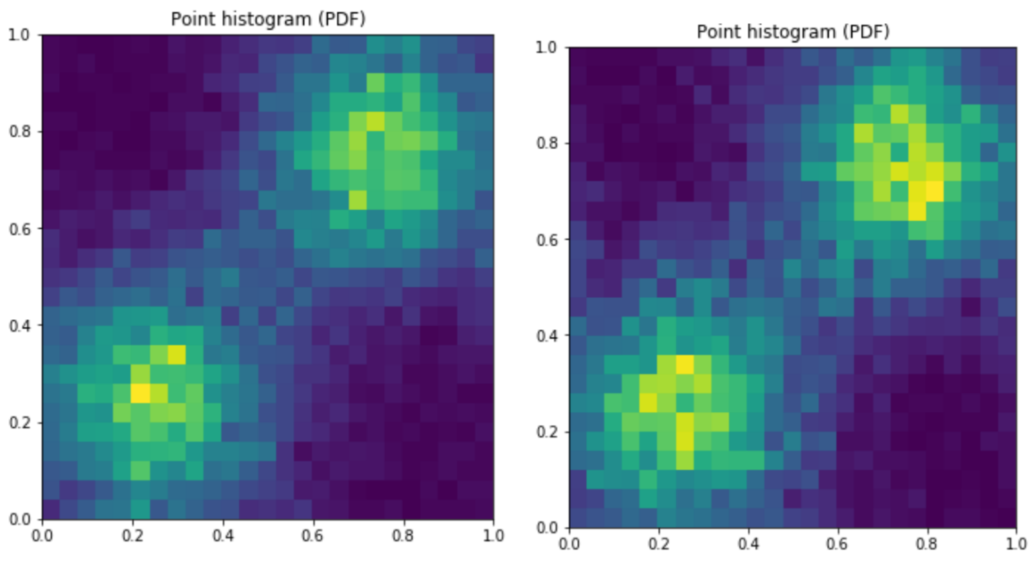


Figure 3.8 – The output of the piecewise-linear coupling cell after hyperparameter optimisation in the best (left) and second best (right) configuration

This second best configuration had a much higher learning rate and reached the minimal loss earlier and evaluating the integrand thus less often, at the cost of not reaching the full variance reduction.

Although the level of loss reduction strongly varies between the configurations and runs, even when doing a wide scan of different configurations, 35% of the runs reduce the variance down to 10% of the initial value and 90% reach reduction to 50% of the initial variance before diverging again. Less than 2% of the configurations did not succeed in reducing the loss. The worst runs usually suffered from a small number of bins (<7), and were unable to reflect the structure of the integrand.

For the slashed circle, a considerable improvement can be reached as well as can be seen in table 3.1. However, in the regions where the area between the boundaries and non-zero values is small, the regions where the function is zero is not clearly recognised. The success of the training is highly dependent of the number of bins.

3.2.2 Performance of the piecewise-quadratic coupling cell

The optimal configuration observed in the hyperparameter optimisation had 3 NN layers of width 16, 10 bins in each coupling cell a learning rate of 0.0149, a weight decay of 3.2×10^{-4} and a batch size of 58,000.

	Minimal variance	**Rel. minimal variance**	**Speedup**	**# fct evaluations**	**Execution time/s**
Double Gaussian, best	$2.62 \pm 0.32 \times 10^{-3}$	$3.24 \pm 0.92 \times 10^{-2}$	$3.2 \pm 1.0 \times 10^{1}$	$6.6 \pm 1.2 \times 10^{6}$	180 ± 99
Double Gaussian, 2nd best	$4.6 \pm 2.1 \times 10^{-3}$	$5.6 \pm 2.8 \times 10^{-2}$	$1.78 \pm 0.89 \times 10^{1}$	$5.1 \pm 2.1 \times 10^{6}$	51 ± 34
Slashed circle, best	$3.20 \pm 0.73 \times 10^{-3}$	$6.52 \pm 0.73 \times 10^{-1}$	$1.53 \pm 0.17 \times 10^{1}$	$1.03 \pm 0.28 \times 10^{6}$	210 ± 55

Table 3.2 – Performance of the piecewise-quadratic coupling cell setups.

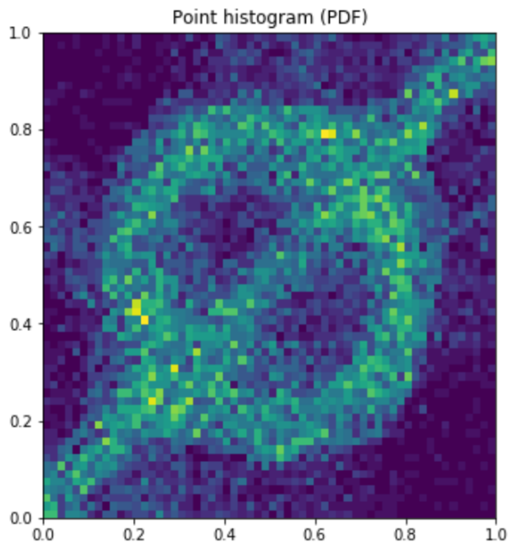


Figure 3.9 – The output of the piecewise-linear coupling cell for the slashed circle

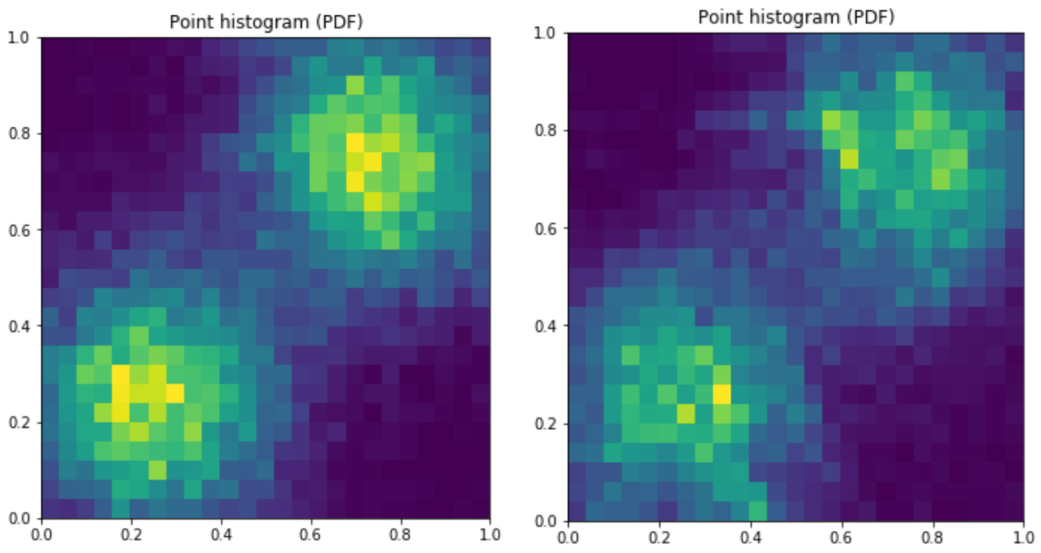


Figure 3.10 – The output of the piecewise-quadratic coupling cell in the optimal configuration (left) and the second best (right)

An alternative, slightly smaller improvement was reached again with a model (NN width 11, NN length 3, 4 bins, learning rate 8.9×10^{-3}, weight decay 4.3×10^{-4}, batch size 26000) which trains considerably faster and reached its optimum after shorter time due to the simpler structure and smaller batches. However, this configuration suffers due to this from a higher variance of the quality of the results, as can be seen in table 3.2

Although again the level of loss reduction strongly varies between the configurations and runs, even when doing a wide scan of different configurations, half of the runs reduce the variance down to 10% of the initial value and 95% reach reduction to 50% of the initial variance before diverging again. Less than 2% of the configurations did not succeed in reducing the loss.

For the slashed circle, a great improvement is noted. Here, the zero-valued regions around the boundaries is well recognised. The region inside the circle is however still not very crisp.

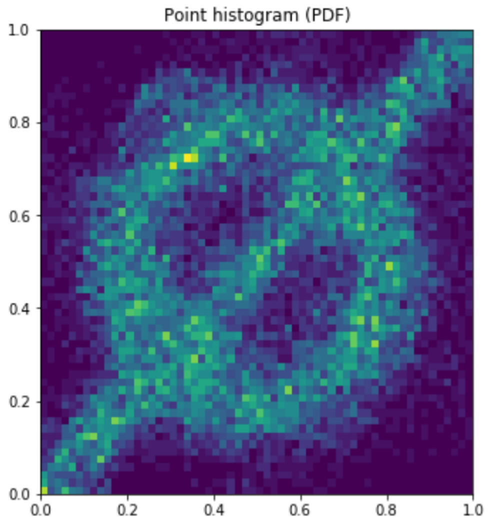


Figure 3.11 – The output of the piecewise-quadratic coupling cell for the slashed circle

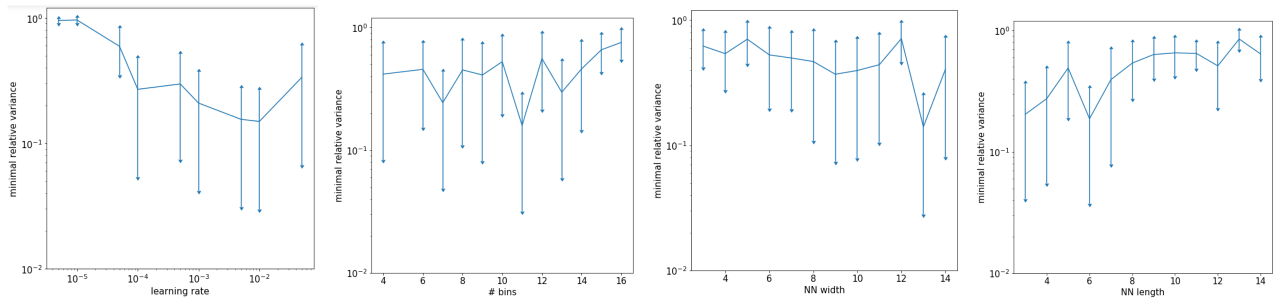


Figure 3.12 – Overview of the minimal relative variance achieved of the different setups of the piecewise-quadratic coupling cells versus the hyperparameters of the setup. From left to right: learning rate, number of bins, width of the NN, length of the NN.

The examples and the overview of the performance versus the hyperparamters in figure 3.12 make clear that the number of bins is not an decisive hyperparameter. It is only required to be high enough to reflect the structure of the integrand for the piecewise-linear cells, but beyond this constraint, the number of bins does not affect the training outcome in general. The value of the learning rate is the single most important hyperparameter. Indeed, values much greater than 3×10^{-2} lead to a very instable training process during which no great improvement can happen, whereas values much smaller than 1×10^{-4} do not succeed in improving the model sufficiently in order to see improvement, as the statistical error between the batches becomes greater than the influence of the training. As figure 3.13 shows, the range of values of the learning rate for which an improvement is achieved increases with the batch size, and bigger batches achieve a better training. Whereas figure 3.12 shows apart from this no clear correlation between the performance and the number of bins or the width of the NN, a certain preference for short NN appears. Very deep NN can lead to unstable training due the high number of

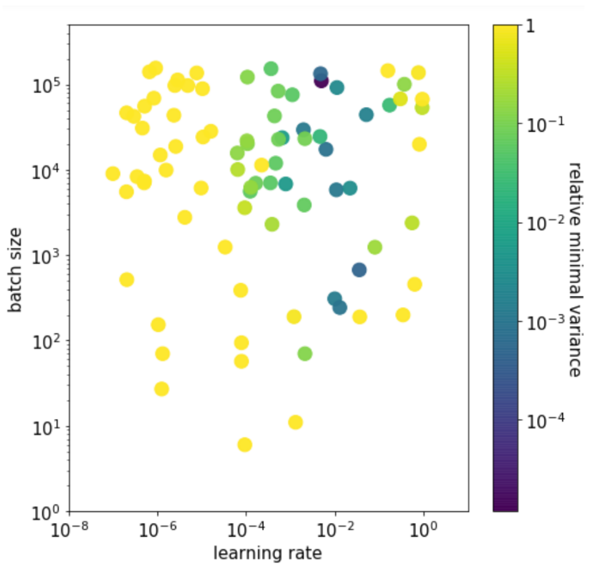


Figure 3.13 – Overview of the minimal relative variance achieved in relation to the batch size and the learning rate.

activation functions involved. In general, however, the variance between the runs is of the same order of magnitude as the influence of the hyperparamters.

3.2.3 Analysis of the behaviour of piecewise-quadratic coupling cells

The piecewise-quadratic coupling cells reach in general a lower minimal loss before diverging in comparison to the piecewise-linear coupling cells, resulting in a greatly increased speedup. They are also more tolerant of disadvantageous choices of the hyperparameters, in the sense that it is less likely that the training immediately diverges. They are also much more tolerant to the number of bins due to the increased expressivity. The lower number of bins needed also compensates the increased computational cost of the quadratic bins. In this example, half the number of bins were needed, which means that both the quadratic and the linear bins tried to determine the same number of parameters.

	Rel. minimal variance PWLin	**# fct. eval. PWLin**	**Rel. minimal variance PWQuad**	**# fct. eval. PWQuad**
Double Gaussian	$4.84 \pm 0.48 \times 10^{-2}$	$9.53 \pm 0.57 \times 10^{6}$	$2.62 \pm 0.32 \times 10^{-3}$	$6.6 \pm 1.2 \times 10^{6}$
Slashed circle	$1.312 \pm 0.076 \times 10^{-1}$	$9.09 \pm 0.95 \times 10^{6}$	$6.52 \pm 0.73 \times 10^{-2}$	$1.03 \pm 0.38 \times 10^{7}$

Table 3.3 – Comparison of the performance of the piecewise-linear and piecewise quadratic coupling cell.

The difference between the training of both coupling cells becomes more clear looking at the evolution of the mapping during the training process. Figure 3.14 and 3.15 show that the quadratic coupling cells reach earlier an already reasonable adaption and suffer less from linear artefacts originating from the bin boundaries. This is especially relevant when the cost of

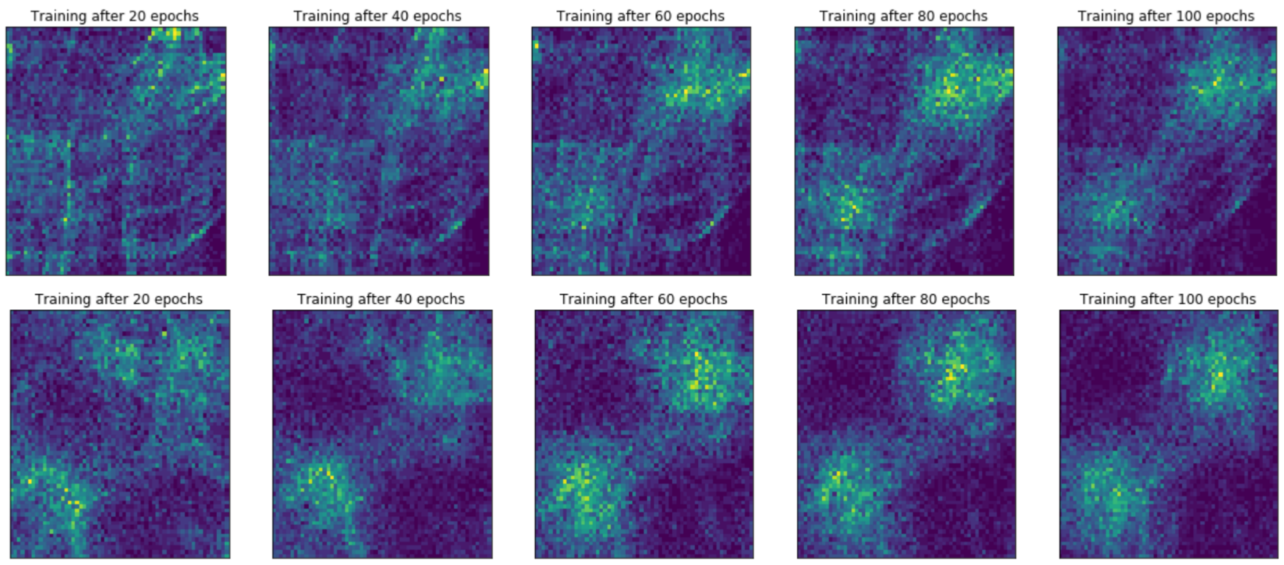


Figure 3.14 – Time evolution of the mapping for the gaussian double peak for piecewise-linear (top) and piecewise-quadratic (bottom) coupling cells.

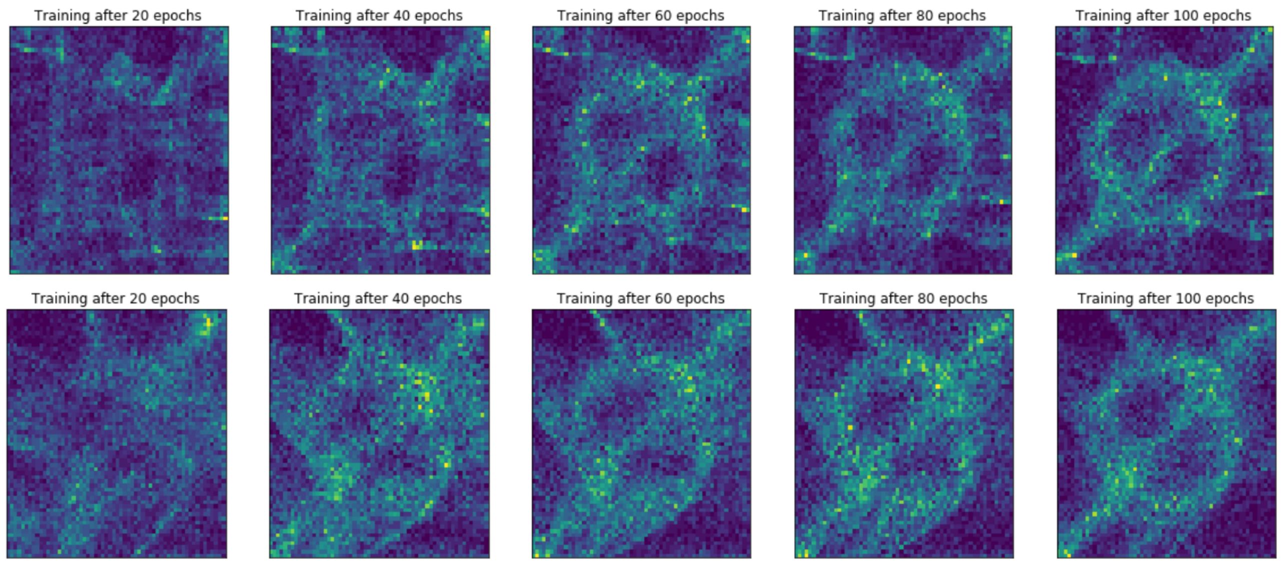


Figure 3.15 – Time evolution of the mapping for the slashed circle for piecewise-linear (top) and piecewise-quadratic (bottom) coupling cells.

evaluating functions rises and the maximal improvement is less important than the improvement over function evaluations. From the evolution, it becomes also clear that both coupling cells learn all features of the functions at the same time, instead of first learning for example the diagonal of the slashed circle and then improving upon including the circle.

This makes clear that the piecewise-quadratic coupling cells are an improvement over the linear ones, which is why we will focus on their application from now on.

An output of the bin boundaries of the trained model gives insight into the way the piecewise-quadratic coupling cells adjust to the structure of the function learned. A first observation is the continuity of the boundary borders, which is necessity in order to preserve differentiability.

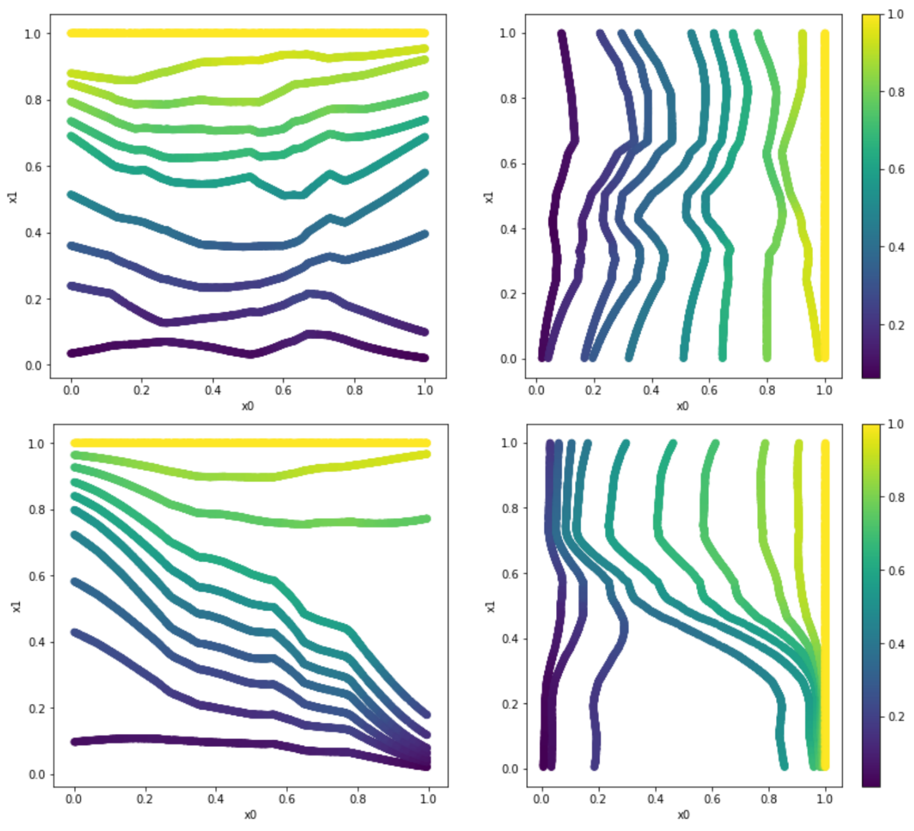
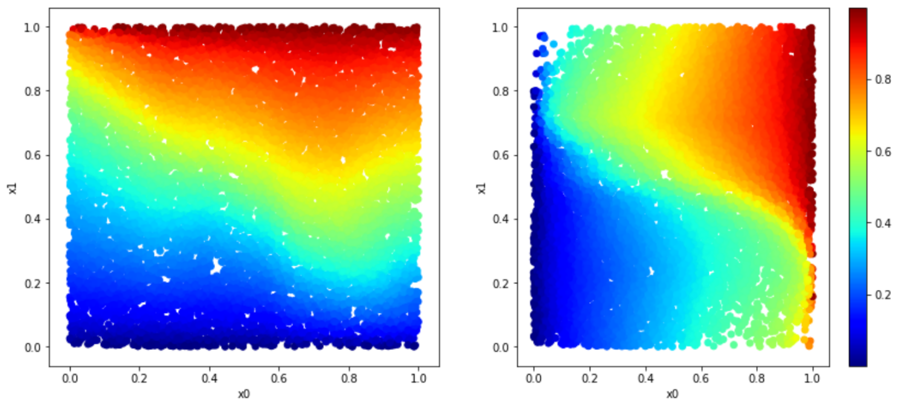


Figure 3.16 – Bins of the first (left) and second (right) coupling cell of the optimal configuration. Top: untrained, bottom: trained model. The axis parallel to the lines refers to the non-transformed dimension (in the right image, this dimension was transformed by the first coupling cell already), the position of the lines to the right bin boundary. The colour encodes the value of the cumulative distribution function at the right bin boundary, e.g. the value to which the input dimension will be transformed at the bin boundary.

In a next step, one can focus on the second coupling layer. For both peaks, the bin size is increased, whereas the values of the CDF are only slowly increasing over the boundaries close to the peak. However, for other examples one would see only small changes of bin sizes and a stronger adjustment via the values of the vertices. This becomes more clear when taking into account how the coupling cell maps the input into the output. Two neighbouring lines with similar values of the CDF means that input which lies in this bin is mapped to the same output value; in this example for the first cell, all input vectors with $0.1 < x_0 < 0.3$ and $0.1 < x_1 < 0.5$ are mapped to $y_1 \sim 0.25$, which increases the likelihood of such configurations and causes a peak. The opposite effect comes into play for small bins with a rapid change of the cumulative value, as for the second coupling cell. There, for the upper peak, the neighbouring minimum of the distribution function is realised by small, rapidly increasing bins.

This is indeed also what happens for the piecewise-linear coupling cell, however, here the adjustment is only done via the bin heights. This enforces a constant resolution and reduces by this how easily the model can adopt to more complex structures.



Cumulative distribution function of the first (left) and second (right) coupling cell of the optimal configuration. The axes refer to the dimensions of the input. The colour encodes the value of the cumulative distribution function.

Looking at the plot of the whole CDF without the bins, one can see the shift of mapping probabilities along the transformed dimension. For the second coupling cell, the both peaks are especially clearly realised, whereas for the first cell, the are visible in a less clear resolution. It is to be expected that the first cell gives a less clear resolution as its output is alternated by the second cell. In some cases, a splitting of peaks between the coupling cells can be observed, such that each of the cells has a clear resolution of one of the peaks.

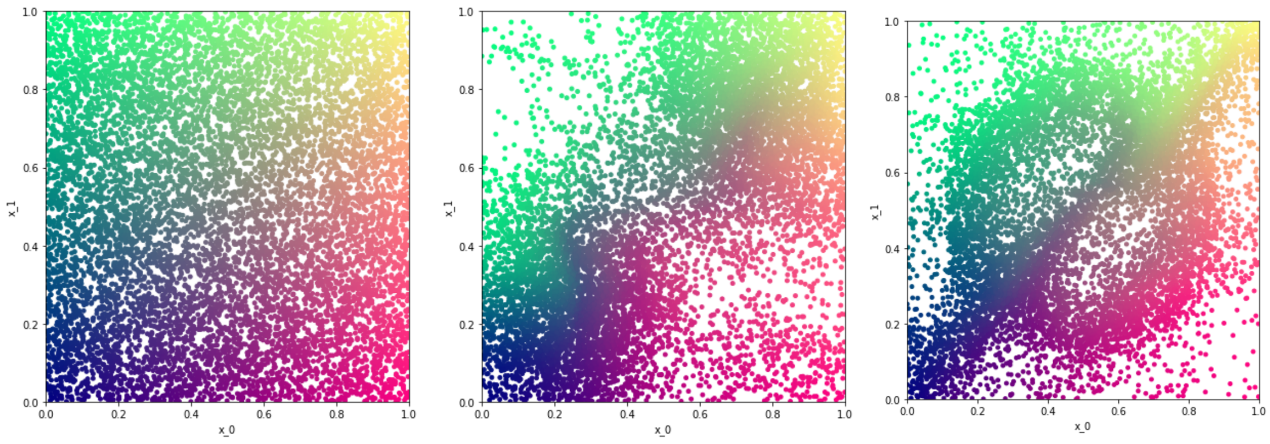


Figure 3.17 – Left: the colour-encoded 2D plane. Each point got encoded in the RGB-scheme as $(x_0, x_1, 0.5)$. Middle: the same set of points after transformation with a model trained on the double gaussian, but with the colour-encoding of their original position. Right: the same, but mapped with a model trained on the slashed circle.

Similar to this, figure 3.17 shows how the mapping of each individual point is performed. In accordance to what has been shown before, the transformation is continuous and mainly shifts points in their immediate surroundings instead of shifting them by large distances. For the double peak, the main contributions originates in density shifts along x_1, as expected from the CDF. For the slashed circle, the translation patterns of the individual points is richer. The circular structure is created translating points out of the top left and bottom right corner towards the other two corners, whereas the minima inside the circle are generated by moving the points there on the slash and the circle maxima. This shows the high expressiveness of this approach.

3.3 Comparison to the VEGAS algorithm

The VEGAS Python-package [45] is an implementation of original VEGAS algorithm described earlier. In the following, its performance in comparison to the piecewise quadratic coupling cells is demonstrated. The adaptive grid of this module was trained on the respective function, without the usage of stratified sampling. The training was done with 10000 evaluations per epoch until the variance reduction per epoch was minimal.

For the first example, a central gaussian is trained using the best configuration of the piecewise-quadratic coupling cells before. The comparison of the performance is presented in table 3.4.

	Minimal variance	**Rel. minimal variance**	**Speedup**	**# fct evaluations**	**Execution time/s**
VEGAS	$2.26 \pm 0.28 \times 10^{-5}$	$3.71 \pm 0.28 \times 10^{-4}$	$2.69 \pm 0.29 \times 10^{3}$	$4.31 \pm 0.15 \times 10^{6}$	52.4 ± 5.3
NIS	$1.72 \pm 0.23 \times 10^{-3}$	$2.82 \pm 0.13 \times 10^{-2}$	$1.12 \pm 0.15 \times 10^{1}$	$3.32 \pm 0.25 \times 10^{6}$	45.2 ± 9.3

Table 3.4 – Performance for the two integration strategies for the single gaussian peak.

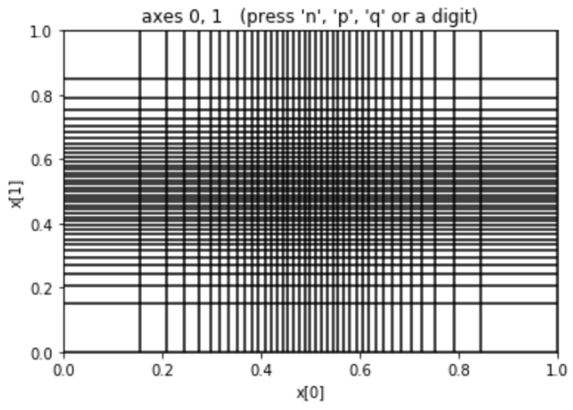


Figure 3.18 – The adaptive grid of VEGAS at the end of the training for the gaussian peak.

As expected, the VEGAS algorithm is very successful at learning this structure. It reached a considerably higher accuracy. Although the execution time was higher, this was mainly caused by it being executed on the CPU instead of the GPU, as the number of function evaluations used is comparable. This shows that for very simple, low-dimensional integrals, this algorithm is superior due to its higher stability, simpler structure and high optimisation.

The second example is considerable more challenging. Here, we use again the gaussian double peak, e.g. the camel function. Other than before, correlations exist now between the integration axes, which greatly reduces the performance of VEGAS in comparison to neural importance sampling, as can be seen in table 3.5.

	Minimal variance	Rel. minimal variance	Speedup	# fct evaluations	Execution time/s
VEGAS	$6.38 \pm 0.25 \times 10^{-3}$	$7.51 \pm 0.30 \times 10^{-1}$	1.332 ± 0.041	$2.11 \pm 0.42 \times 10^{6}$	37.4 ± 3.4
NIS	$2.62 \pm 0.32 \times 10^{-3}$	$3.24 \pm 0.92 \times 10^{-2}$	$3.2 \pm 1.0 \times 10^{1}$	$6.6 \pm 1.2 \times 10^{6}$	180 ± 99

Table 3.5 – Performance for the two integration strategies for the double gaussian peak.

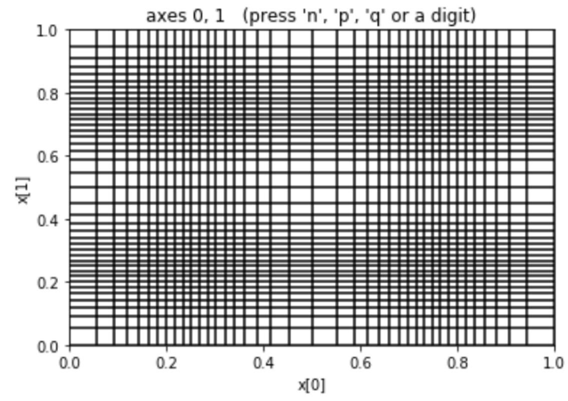


Figure 3.19 – The adaptive grid of the VEGAS algorithm at the end of the training for the gaussian double peak. Note that instead of two peaks, four were learned.

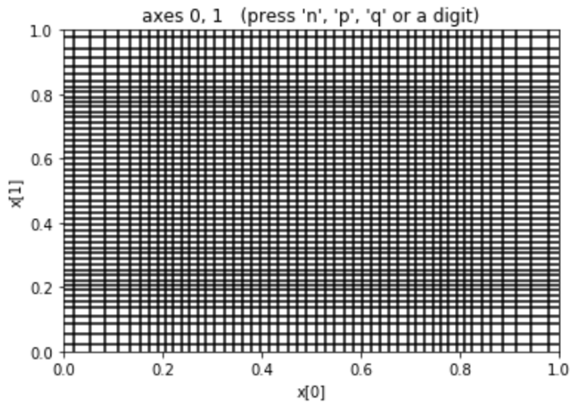


Figure 3.20 – The output of the VEGAS grid for the slashed circle

As expected, VEGAS detects four instead of two peaks, as the adaption of the grid is done for both axes independently. As a result, the relative loss is considerably higher than for the piecewise-quadratic coupling cells seen in the last section. Furthermore, VEGAS had a higher runtime and consumed more function evaluations.

The situation becomes even more clear for the slashed circle function in figure 3.20.

VEGAS is unable to represent the zero-valued areas inside the circle as it sums up the function values along the non-transformed axis. The only adaption possible is therefore a finer grid in the area of the circle than outside of it, reaching only a minimal improvement down to about 80% of the original variance.

This shows that neural importance sampling can indeed outperform the widely adopted VEGAS algorithm both in variance reduction and runtime. This first success motivates a more in depth exploration of neural importance sampling.

Chapter 4

Investigation of the quadratic-coupling cells in higher dimensions

After having investigated the behaviour in 2 dimensions, the next natural question is how neural importance sampling behaves in higher dimensions, which are of course of great interest for high energy physics. In order to be able to give a clear statement about the performance of neural importance sampling, a wide range of dimensions and gaussian multi-peak functions is investigated. As the variance reduction depends on the value of the hyperparameters and as it is not clear whether there is a optimal setup for all integrands, a hyperparameter optimisation has to be performed for each different integrand.

4.1 Setup

Contrary to earlier setups, several modifications have been performed on the algorithm. In order to reach high batch sizes also for high-dimensional problems, minibatches of size 20,000 are used, which helps dealing with limited memory. The gradients are agglomerated during each epoch for the required number of minibatches to reach the full batch size. Additionally, the breaking condition is now that every 25 epochs, it is tested if the minimum loss has decreased by at least a factor of 0.999 (this deals with situations where only minimal improvement is reached). Apart from this, if the loss is increasing for 5 epochs in a row, the training is stopped as well (early stopping regularization). As a last breaking condition is the requirement that the training is not allowed to last for more than 8 minutes, a time after which the point of minimal loss was nearly always reached in earlier experimental runs.

Additional modifications include the modified masking described earlier, following the I-FLOW library [30] in order to be able to use the minimum number of coupling cells. Another modification stabilises the initialisation of the coupling cells. For higher dimensions the volume of the gaussian peaks in comparison to the volume which is virtually zero becomes very small. For

2 dimensions and the single gaussian peak with width 0.25, the volume inside the hypercube with a function value greater than 0.01 is ~ 0.85. For 8 dimensions, this is reduced to ~ 0.029. Due to this, the chance that a point in the batch lies in this region is largely reduced and the quasi-uniform initialisation does not guarantee a successful training. Therefore, a survey phase (the so called "preburner") is used to compensate this. This is done by training in the beginning with a modified loss of the function evaluated on the points of the latent space (which is uniformly distributed) times the Jacobian of the transformation. This loss is then, similiar to equation 3.3

$$L = \mathbb{E}_{\mathbf{x} \in U(\Omega)} (f(\mathbf{x}) J)^2, \tag{4.1}$$

This ensures that the Jacobian is proportional to the inverse of the training function. When this modified loss is decreased considerably (here, the boundary was set for the modified loss to be 15% of the initial loss. If this can be not achieved, the survey phase is stopped after 100 iterations), the normal training started. The training then starts with a mapping which has a much higher likelihood to recognise the to-be-learned features.

The advantage of this approach is that uniform sampling in the target space is guaranteed for the start of the training, making it less likely that disadvantageous gradient descent steps lead to a model which only samples scarcely into the important regions of the target space. The downside is that during the preburn time, the wrong loss is optimised, which is the reason why it is desirable that it is kept as short as possible. In preliminary runs, this modification was crucial to ensure a successful training.

The performance of the algorithm is tested in 2, 4, 8, 16 and 32 dimensions. For each of this dimensions, runs are performed with NN of length 3 and 8 (minimum one hidden layer is required to be able to learn more complex structures; lengths far beyond 10 tend to lead to instable training) and width 6, 11 and 16.

The training functions are a single, centred gaussian peak; two gaussian peaks on a hyperdiagonal; four gaussian peaks (at $[0.25 \dots 0.25]$,$[0.75 \dots 0.25]$,$[0.25 \dots 0.75]$ and $[0.75 \dots 0.75]$) and similarly eight gaussian peaks on positions $[a, \dots, a, b, \dots, b, c \dots, c]$. The runs are performed with peak widths of 0.15, 0.25 and 0.35. This is intended to test how well the algorithms can deal with small peaks on the one hand and wide, touching peaks on the other hand. These peaks are motivated by the enhancement of propagators in particle physics. However, they allow us to investigate the behaviour of neural importance sampling without having to apply a mapping into the phase space.

The remaining hyperparameters are determined via hyperparameter optimisation by comparing 80 runs per configuration. The boundaries of the hyperparameters are determined by hardware restrictions (especially the batchsize) on the one hand and the usual positions of optimal values in earlier investigations on the other hand. The batch size will be chosen in multiples of the minibatch size in the range of 1 to 4, the number of bins in the range of 4 (which is the minimum needed to spline two gaussian peaks) to 16, the learning rate in a range from 5e-5 to 2e-2 and the weight decay in a range from 1e-6 to 5e-4. The number of coupling cells is also used as a hyperparameter, giving the choice between the minimum amount of coupling cells needed to learn the distribution in all dimensions, $2\lceil \log_2 D \rceil$, and additionally for the same amount of coupling cells added which will each transform a single dimension. The reasoning behind this is to test if additional transformations on certain dimensions after the adjustment of all dimensions can lead to an additional improvement.

After hyperparameter optimisation, the ideal configuration is executed 6 times. The documented values are cumulants over these runs.

The same functions are also used to perform importance sampling with the VEGAS algorithm using the VEGAS python package, for each configuration in 12 runs. The VEGAS algorithm is executed with 25,000 evaluations per iteration until its variance improvement after 5 iterations becomes less than 1%. On each axis, 40 grid points are given, such that multiple peaks can be detected easily.

4.2 Results

In the tables in the appendix, the original variance of the test function is given, as well as for both the present algorithm as well as the VEGAS algorithm the variance reduction factor (VRF, the variance using importance sampling divided by the variance using flat sampling) and the number of function evaluations used to achieve this (this does not include the number of evaluations during hyperparameter optimisation for neural importance sampling or evaluations after the optimum was reached). Additionally, the number of standard deviations between the results of both algorithms are given in order to assure their agreement, as well as the ratios of both VRFs, where a ratio greater than 1 means that the VEGAS algorithm reached a superior result, and a ratio smaller than 1 is a sign of a better improvement with neural importance sampling. The last column contains the estimated relative speedup between neural importance sampling and the VEGAS algorithm, derived from the mean relative variance improvement.

In Table 6.7, one can see the outcome for the run in 2 dimensions. The 8-peak case has been omitted as with the placement outlined above, only 4 peaks are possible in two dimensions. In 2 dimensions, the quadratic coupling cells in general perform worse than the VEGAS algorithm for the single peak and the 4-peak case. An improvement can be only seen for 2-peak case, where the improvement is greater for wider peaks. VEGAS can perform well in the single and 4-peak case as here the peaks are distributed fully symmetrical, which explains its good performance in this case. This shows that for lower dimensions, neural importance sampling is not in general superior, but can compensate the difficulties of the VEGAS algorithm.

For the run in 4 dimensions in Table 6.8, VEGAS is superior for the single peak. However, for multiple peaks the quadratic coupling cells manage to achieve results of similar quality, however with a higher variance of the outcome. This could be connected to the strategy of hyperparameter optimisation, which only tries to improve the loss and not the consistency of the achieved improvement. Runs with many or wide peaks tend to give less improvement. This is a sign that neural importance sampling does perform better for sharper features.

Again in Table 6.9 for 8 dimensions, neural importance sampling gives equal or superior results to the VEGAS approach for test functions with more than one gaussian peak, additionally now with a more comparable variance of the quality. The superiority of neural importance sampling is not caused by a lack of improvements from the VEGAS sampling, but they can instead reach better results as it can deal with the correlations between the axes. The improvements are however smaller than for the 4 dimensional case. It is important to mention that the required number of function evaluations is in general of the same order of magnitude for the VEGAS algorithm and neural importance sampling. This shows that the improvement comes only with low additional computational cost if used for slow integrands, which is the case for the desired application on cross sections.

In the case of 16 dimensions in Table 6.10, multiple cases occur where the VEGAS algorithm can reach no improvement at all. This is related to the fact that the phantom peaks which occur for every additional dimension introduce additional sources of variance, until no improvement can be achieved. As further investigation showed, this can be not compensated by an increase in evaluations during each iteration. Only a considerable higher number of iterations can improve the performance of VEGAS slightly, yet not beyond the performance of neural importance sampling and only at the cost of many additional function evaluations. The quadratic coupling cell can reach great improvements especially for small peak widths and many-peak cases. However, the VEGAS algorithm is still superior for flat single-peaks. With increasing dimensionality, the variance of the improvement of neural importance sampling increases. The reason for this is the ever increasing empty volume, which is not the case for physical integrands. This increases the possible gain of the variance reduction, but also reduces the stability of the training as the sampling density of the batch decreases and statistical errors become more and more important. Especially for higher dimensions, the batch size is limited due to memory issues, increasing this problem.

This is the reason why for 32 dimension, narrow peaks were excluded. The problems of VEGAS increase whereas neural importance sampling can now also reach superior results for the single peak case.

As a result, it seems like the VEGAS algorithm is preferable for flat, low-dimensional problems without correlations along the axis, whereas neural importance sampling performs better for highdimensional problems with multiple strong peaks. A problem that remains is the high variance of the quality of the result of neural importance sampling. Especially for higher dimen-

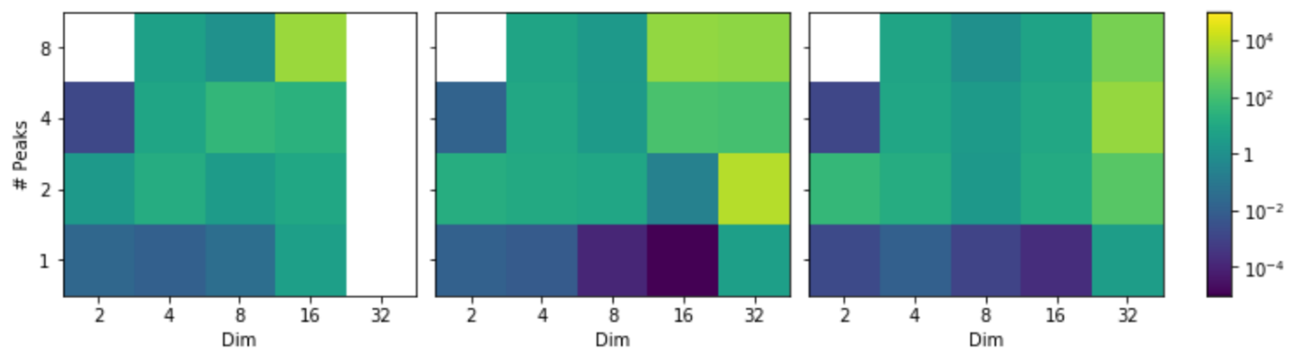


Figure 4.1 – The absolute speedup of neural importance sampling in dependency of the number of peaks and dimensions. The colour encodes the logarithm of the speedup. Left: peak width 0.15, in the middle 0.25 and to the right 0.35. White fields represent non-existing data points.

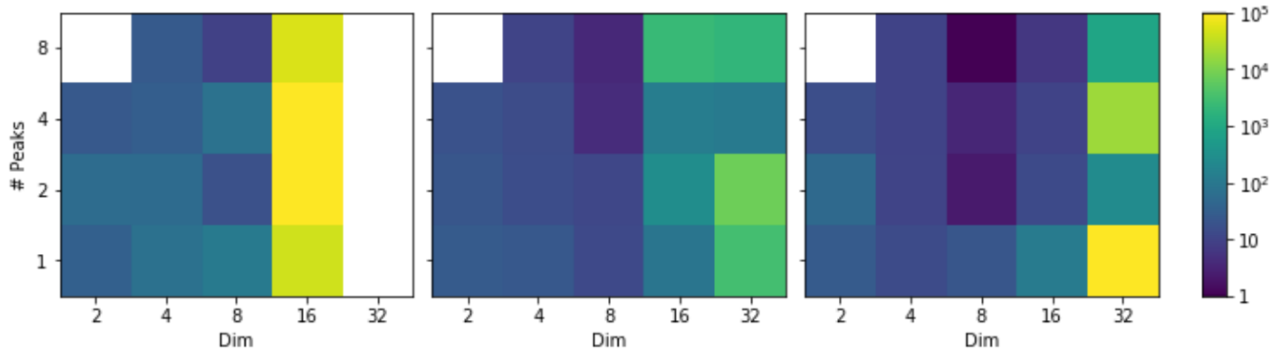


Figure 4.2 – The relative speedup between neural importance sampling and VEGAS in dependency of the number of peaks and dimensions. The colour encodes the logarithm of the relative speedup. Left: peak width 0.15, in the middle 0.25 and to the right 0.35. White fields represent non-existing data points.

sional cases, the uncertainty is often of the order of the improvement or one magnitude smaller. Next to the zero-volume, this might be related to the strategy of hyperparameter optimisation, which chooses configurations with good maximum improvement without knowing the average improvement reached by the respective set of hyperparameters.

An overview over the relative performance can be gained by visualising the absolute speedup of neural importance sampling and the relative speedup in comparison to VEGAS in dependency of the dimension and number of peaks in figure 4.2 and 4.1. Again, for cases where the VEGAS algorithm could reach no improvement, the absolute speedup was taken. In the figures, one can see clearly that VEGAS performs comparatively well for single peaks, as expected. Additionally, for four peaks in 2 dimensions, neural importance sampling has no advantage too, as this configuration is symmetrical in both dimensions. However, for increasing dimensions and number of peaks, neural importance sampling is eventually superior. The absolute speedup is most of the time of the same order, except for smaller improvements for wide peaks in dimension 8 and greater improvements for very high dimensional cases and small peaks, which can be also a result of the scarce distribution of points.

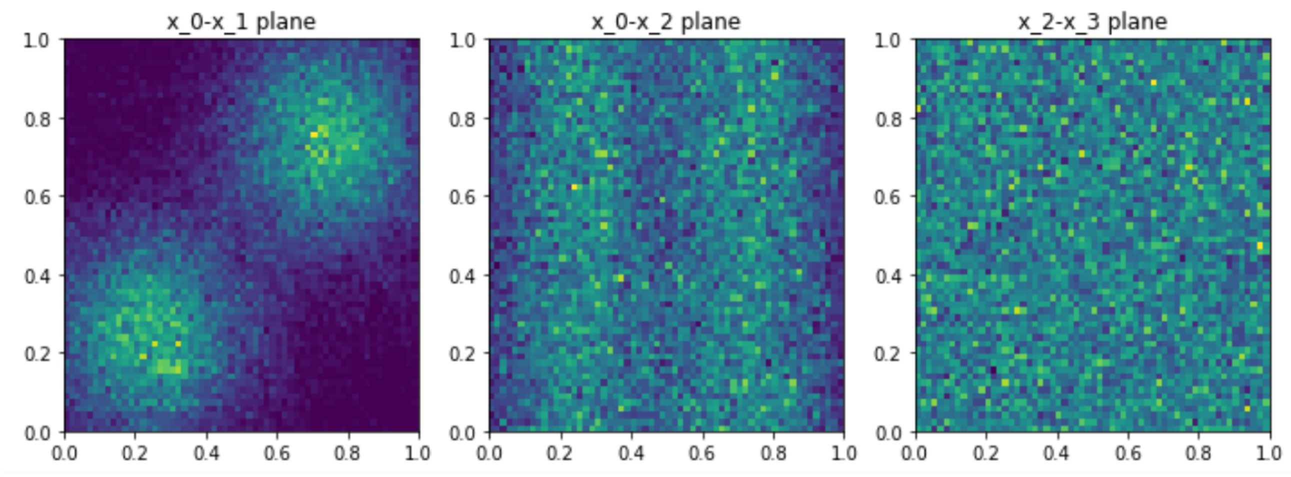


Figure 4.3 – The trained model for the 4D double gaussian peak in multiple projections.

One major reason for the fact that the improvement is approximately always of the same order is that the most important adaption is performed by the first coupling cell only. Looking at the 4D gaussian double peak in figure 4.3, one can see that the projection in the $x_0 - x_1$-plane is of the same quality as seen before in 2 dimensions. Looking instead into the $x_0 - x_2$-plane, the adaption is much worse; indeed, along the x_2 axis, only a small change of likelihood of sampling is observed, whereas the peaks are still visible along the x_0 axis. On the $x_2 - x_3$-plane, almost no adaption happens. The reason for this is that, following the masking outlined earlier, the last two coupling cells learn (x_0, x_2) for fixed (x_1, x_3) and vice versa. The last coupling cells are easier to learn as their effect is not altered by later transformations. However, the preference of learning the correlation between x_0 and x_1 over the others seems to be an effect arising in the training only, as all dimensions are transformed, but x_0 and x_1, considerably stronger. The optimizer seems to favour learning one correlation between the variables instead of all at once, leading to a behaviour similiar to 2 dimensions.

This is however not a general behaviour of neural importance sampling. The training does not try to improve all correlations but tries to reduce the variance, which can be in this case be achieved the easiest by focusing on the double peak in the $x_0 - x_1$ plane. Another example shows that in general all features can be learned. Taking the function

$$\begin{aligned} f(\mathbf{x}) = &\exp\left(\frac{(x_0-0.5)^2+(x_1-0.5)^2}{0.25^2}\right) \\ &+\exp\left(\frac{(x_2-0.25)^2+(x_3-0.25)^2}{0.25^2}\right)+\exp\left(\frac{(x_2-0.75)^2+(x_3-0.75)^2}{0.25^2}\right), \end{aligned} \tag{4.2}$$

in words, a simple gaussian peak for the first two and a double peak for the second two dimensions. Now, the training cannot achieve the same improvement for all correlations along the axes and therefore does indeed succeed at learning different structures along different dimensions, as can be seen in figure 4.4.

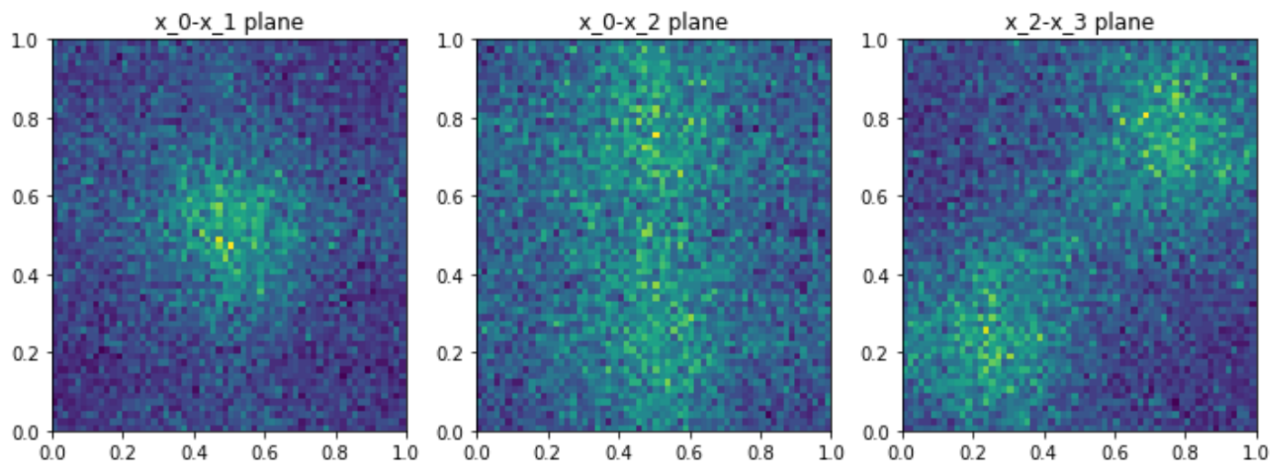


Figure 4.4 – The trained model for the 4D mixed gaussian peaks in multiple projections.

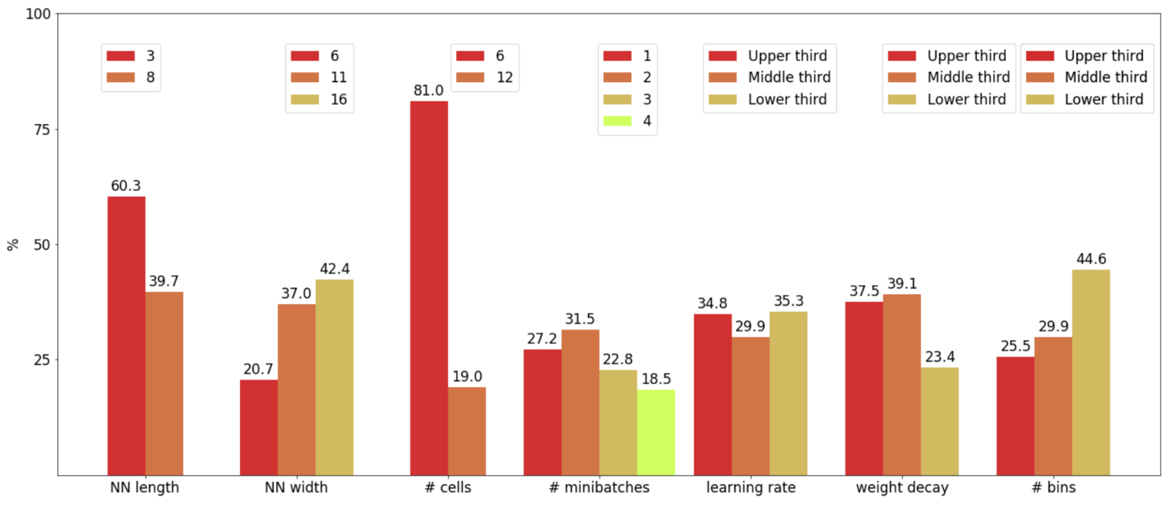


Figure 4.5 – Distribution of the hyperparameters for the best runs in 8 dimensions of the gaussian double peak with peak width 0.25.

4.3 Analysis of the impact of hyperparameters

For the case of the double-peak gaussian with a peak width of 0.25 in 8 dimensions, one can take a look on which choice of hyperparameters lead to a good performance. For this, all runs with randomly assigned hyperparameters during the optimisation step are taken as the sample. From this, one can take now a look on all runs which were within two standard deviations from the best run (N=184). The distribution of their hyperparameters can be seen in figure 4.5. It shows that, whereas the number of minibatches of 20000 points and the learning rate are more or less uniformly present, there is a clear preference of the minimal number of coupling cells. Shorter and wider NNs as well as higher weight decay and smaller number of bins are slightly preferred.

The aim is now to gain a deeper understanding of the rate of convergence of the training. For this, the above training set is split and the distribution of the hyperparameters and the architecture parameters is compared for the quartile of models which converged after the shortest and largest number of epochs. This can be seen in figure 4.6. In comparison, the learning rate is very different for both sets, as expected. A high learning rate leads to less gradient descent steps necessary in order to reach convergence. For the other parameters, the results are less clear. Amongst the runs with less gradient descent steps, wide NNs with a higher number of bins seem to be slightly more common. There are also more configurations with additional coupling cells. However, the runs requiring more steps have in average more layers in their NN. This is surprising, as in general, more complex architectures are more expressive and should train faster. This could be a hint that for this function to be trained, less non-linear behaviour/activation layers are needed. This would mean that introducing additional layer nodes is to be preferred over introducing additional layers.

In light of this, it is worth to also look at the fastest and slowest quartile with respect to time, which takes into account the numerical cost of more complex architectures, yet is prone to poor generalizability due to technical factors. In figure 4.7, one can see again the direct comparison. Again, the expected result that shorter runs were such which used a smaller number of minibatches is realised. The learning rate tends to be higher for the fast runs, although the correlation is much less clear than before. However, in light of the last result, it is interesting to note that again a smaller number of layers with more nodes seems to have lead to a faster training, as they also needed less epochs to finish the training.

The combination of wide NNs with only few layers seems therefore to be ideal with respect to training time, reducing both the number of epochs needed as well as the numerical cost. The expressiveness of the model is then given by the width of the NN. Indeed, many of the runs with the best training result were of this combination of parameters, proving that it reaches not only good but optimal results.

Now that it has been shown that neural importance sampling can be - if the right hyperparameters have been found - used to train functions also in higher dimensions, the next step is to relate this to phase space integrals.

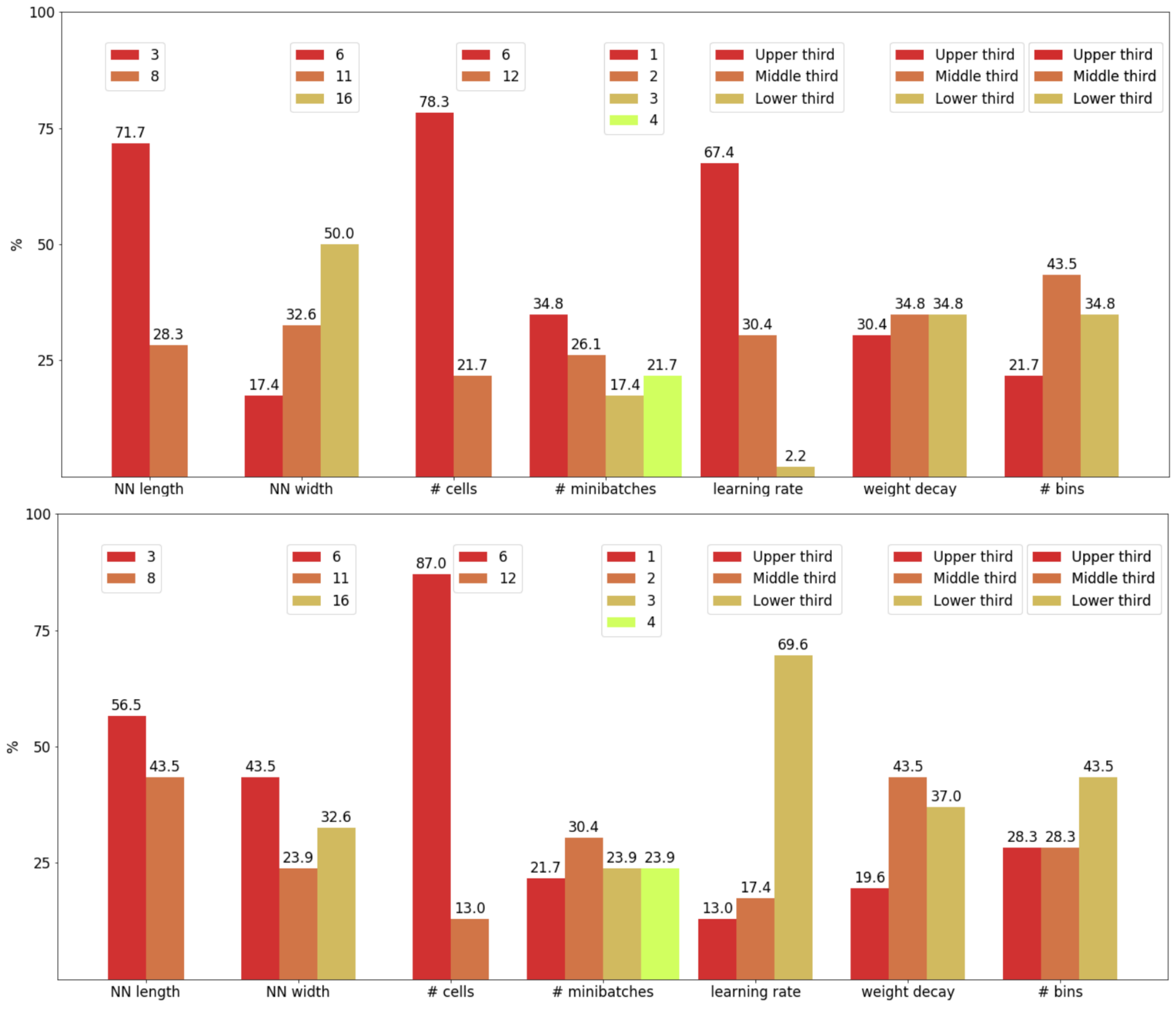


Figure 4.6 – Distribution of the hyperparameters for the best runs in 8 dimensions of the gaussian double peak with peak width 0.25. Top: data for the quarter which converged after the smallest number of epochs. Lower: data for the quarter which converged after the biggest number of epochs.

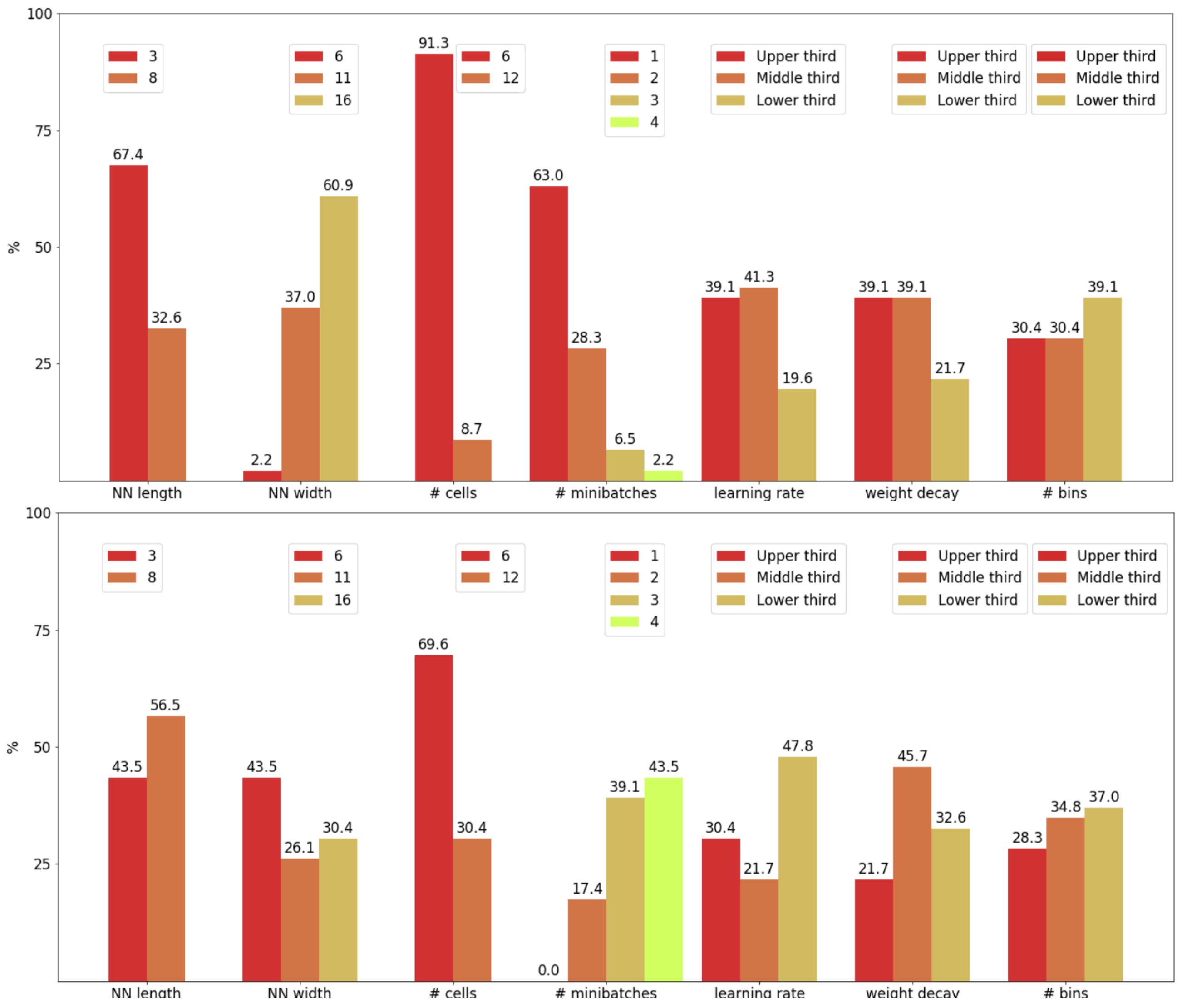


Figure 4.7 – Distribution of the hyperparameters for the best runs in 8 dimensions of the gaussian double peak with peak width 0.25. Top: data for the quarter which converged after the shortest time. Lower: data for the quarter which converged after the longest time.

Chapter **5**

Phase Space Generation

In the following, the aim is to show that the neural importance sampling algorithm can be used in event generators in order to improve the precision of the predicted cross section for cases in which other importance sampling algorithms either fail or need detailed knowledge of the integrand. In order to achieve this, the neural importance sampling is interfaced with modern event and matrix element generators like MadGraph [46]. As importance sampling does not require any information on the integrand, it can be applied to any matrix element. In order to be used for phase space integration, the importance sampled target space has to be mapped into the phase space. For the massless $2 \to 2$-case, this could be naively done by mapping the hypercube of the importance sampling output into $\mathbb{R}^n$, using for example the tangent, and enforcing the kinematics during this mapping. This however does not evenly distribute the phase space points and thus introduces additional variance. It is therefore of great interest to have an (almost) flat phase space generator which at the same time only requires the minimal amount of input variables for a process with multiplicity n, $3n - 4$.

Although implementations for such a phase space generator, including in Python, are already available, the "RAMBO on diet" [47] phase space generator was implemented during this thesis. The motivation behind this is to make use of the GPU processing tools of PyTorch. This allows to not only perform the importance sampling, but also the phase space generation on GPU. For the case that a GPU-implementation of the matrix element would exist, the integration could be completely performed on GPU, which would greatly increase the speed.

5.1 RAMBO

The first example of a phase space generator given here is RAMBO [48], which serves as an illustrative example of flat phase space sampling. Its advantage in comparison to older approaches (see for example [49]) lies in the fact that instead of a hierarchical chain of subsequent decays into two body states, the final states are generated democratically. For massless many-particle final states, the weights associated with each phase space point are then identical. In other words, in such a case the distribution in the phase space is uniform. As at the high energies of colliders like the LHC the mass of particles can be often neglected and in the light of the results of the last chapter, this is the case which is especially relevant for us.

This original algorithm performs for a n-particle final state the mapping $[0,1]^{4n}$ into n 4-momenta, depending on the centre of mass energy $E_{cm} = \sqrt{P^2}$. The phase space volume for such a case is

$$V_n(E_{cm}) = \int \prod_{i=1}^{n} \frac{\mathrm{d}^4 p_i}{(2\pi)^3} \theta(p_i^0)\delta(p_i^2)(2\pi)^4 \delta^4\left(P - \sum_{i=0}^{n} p_i\right) \tag{5.1}$$

The starting point is to relax now the kinematical restrictions of momentum conservation. In order to maintain a final integral, a weight function is included:

$$R_n = \int \prod_{i=1}^{n} \frac{\mathrm{d}^4 q_i}{(2\pi)^3} \theta(q_i^0)\delta(q_i^2)(2\pi)^4 f(q_i^0) = (2\pi)^{4-2n} \left(\int_0^\infty x f(x)\,\mathrm{d}x\right)^n \tag{5.2}$$

Here, the integrals were performed using the distributions. Now, the task is to relate the p_i and q_i using Lorentz and scaling transformations:

$$p_i^0 = x(\gamma q_i^0 + \mathbf{b}\cdot\mathbf{q_i}) \quad \mathbf{p}_i = x(\mathbf{q}_i + \mathbf{b}q_i^0 + a(\mathbf{b}\cdot\mathbf{q}_i)\mathbf{b}) \tag{5.3}$$

with $Q^\mu = \sum_{i=1}^n q_i^\mu$, $M = \sqrt{Q^2}$, $\mathbf{b} = \mathbf{Q}/M$, $\gamma = \sqrt{1-b^2}$, $a = \frac{1}{1+\gamma}$ and $x = E_{cm}/M$. This transformation basically maps the q_i into a configuration which respects the momentum conservation. We will call this transformation

$$p_i^\mu = x H_{\mathbf{b}}^\mu(q_i) \tag{5.4}$$

One can use this transformation in order to expand the left side of 5.2:

$$\begin{aligned} R_n = \int \prod_{i=1}^{n} \left[\frac{\mathrm{d}^4 q_i}{(2\pi)^3} \theta(p_i^0)\delta(p_i^2)(2\pi)^4 f(q_i^0)\,\mathrm{d}^4 p_i \delta^4(p_i - xH_{\mathbf{b}}(q_i))\right] \mathrm{d}^3\mathbf{b} \\ \delta^3\left(\mathbf{b} + \sum_{i=1}^{n} \frac{\mathbf{q_i}}{\sqrt{\left(\sum_{i=0}^n q_i\right)^2}}\right) \mathrm{d}x\, \delta\left(x - \frac{E_{cm}}{\sqrt{\left(\sum_{i=0}^n q_i\right)^2}}\right) \end{aligned} \tag{5.5}$$

This integral can be solved by substitution, providing:

$$R_n = \int \prod_{i=1}^{n} \left(\frac{\mathrm{d}^4 p_i}{(2\pi)^3} \delta(p_i^2)\theta(p_i^0)\right) (2\pi)^4 \delta^4\left(E_{cm} - \sum_{i=1}^{n} p_i\right) \left(f\left(\frac{1}{x} H^0_{-\mathbf{b}}(p_i)\right)\right) \frac{E_{cm}^4}{x^{2n+1}\gamma}\,\mathrm{d}^3 b\,\mathrm{d}x \tag{5.6}$$

Now, choosing an actual function which fulfils the condition of turning 5.2 into a finite integral, like $f(x) = e^{-x}$, gives for 5.2 the result:

$$R_n = (2\pi)^{2-n} \tag{5.7}$$

As we assume the incoming momenta to sum up to $P = (E_{cm}, 0, 0, 0)$, we also have

$$f\left(\frac{1}{x} H^0_{-\mathbf{b}}(p_i)\right) = e^{-\frac{\gamma E_{cm}}{x}} \tag{5.8}$$

This enables us to perform the integration over $\mathbf{b}$ and x. One can write

$$R_n = V_n \cdot S_n \tag{5.9}$$

with

$$S_n = 2\pi(E_{cm}^2)^2 \frac{\Gamma(3/2)\Gamma(n-1)\Gamma(2n)}{\Gamma(n+\frac{1}{2})} \tag{5.10}$$

This gives the correct expression for the phase space volume

$$V_n(E_{cm}) = (2\pi)^{(4-3n)} \left(\frac{\pi}{2}\right)^{n-1} \frac{(E_{cm}^2)^{(n-2)}}{\Gamma(n)\Gamma(n-1)} \tag{5.11}$$

The form of the sampled momenta is determined by the following algorithm:

In the first step, the $4n$ random numbers $u_{ji} \in [0,1]$ are used to generate n isotropic momenta with energies distributed according to the density $q_i^0 e^{-q_i} dq_i^0$ (this represents the weight function we saw earlier). This is achieved by:

$$\begin{gathered} c_i = 2u_{1i} - 1 \quad \phi_i = 2\pi u_{2i} \quad q_i^0 = \log(u_{3i}u_{4i}) \\ q_i^x = q_i^0\sqrt{1-c_i^2}\cos\phi_i \quad q_i^y = q_i^0\sqrt{1-c_i^2}\sin\phi_i \quad q_i^z = q_i^0 c_i \end{gathered} \tag{5.12}$$

The isotropy is clear from this from, whereas the form of the energy distribution is a result of induction (see Appendix C of [48]).

In the second step, it is only necessary to do the mapping from the q_i's to the p_i's as seen before. This algorithm is indeed democratic in the sense that all final state momenta are chosen at the same time and with the same likelihood.

Hierarchical algorithms can easily include masses in the final state. This is not the case for this democratic approach, as a scaling is not possible without changing their masses. Therefore, after having generated a set of massless uniformly distributed momenta, a second mapping to massive momenta has to be performed.

One can implement this by solving the equation

$$E_{cm} = \sum_{i=1}^{n} \sqrt{m_i^2 + \chi^2 (p_i^0)^2} \tag{5.13}$$

for χ and then generating the new momenta as

$$\mathbf{k}_i = \chi \mathbf{p}_i, \quad k_i^0 = \sqrt{m_i^2 + \chi^2 (p_i^0)^2}. \tag{5.14}$$

This transformation however does not leave the weights unchanged and, as demonstrated in [48], introduces a Jacobian

$$W_M = \left[\frac{1}{E_{cm}} \sum_{i=0}^{n} |\mathbf{k_i}|\right]^{2n-3} \left[\prod_{i=0}^{n} \frac{|\mathbf{k_i}|}{k_i^0}\right] \left[\sum_{i=1}^{n} \frac{|\mathbf{k_i}|^2}{k_i^0}\right]^{-1} \tag{5.15}$$

which breaks the uniformity of the distribution. However, for small masses, $W_M \sim 1$ and also for greater masses, this algorithm has been proven to be superior to earlier approaches.

5.2 RAMBO on diet

Although RAMBO fulfils the purpose of flat phase space generation, the necessity of $4n$ random numbers is a major flaw, as this strongly increases the computational cost for neural importance sampling, especially in the case of many-particle final states. This is the motivation behind the "RAMBO on diet" algorithm [47], which reduces the required random numbers as input down to the number of degrees of freedom $3n - 4$.

The derivation of this algorithm is different. The starting point is the decomposition of the multiparticle phase space with final states $\{p_i, m_i\}$ into subsequent $1 \to 2$ decays with intermediate states $\{Q_i, M_i\}$

$$\begin{aligned} \mathrm{d}\phi_n(\{p_1, m_1\}, \ldots, \{p_n, m_n\}|P) = &\left(\prod_{i=2}^{n} \mathrm{d}\phi_2(\{p_{i-1}, m_{i-1}\}, \{Q_i, M_i\}|Q_{i-1})\right) \\ &\times \left(\prod_{i=2}^{n} \theta(M_{i-1} - m_{i-1} - M_i)\theta\left(M_i - \sum_{k=i}^{n} m_k\right) \mathrm{d}M_i^2\right) \end{aligned} \tag{5.16}$$

with $Q_1 = P$ and $\{Q_n, M_n\} = \{p_n, m_n\}$ and without prefactors. Now, each of the two particle phase spaces can be expressed in the rest frame of its mother particle:

$$\mathrm{d}\phi_2(\{p_{i-1}, m_{i-1}\}, \{Q_i, M_i\}|Q_{i-1}) = \rho(M_{i-1}, M_i, m_{i-1})\,\mathrm{d}\cos\theta_{i-1}\,\mathrm{d}\phi_{i-1}. \tag{5.17}$$

Here, ρ is a weight that comes from the requirement of being on-shell:

$$\begin{aligned} \mathbf{p}_{i-1} &= -\mathbf{Q_i} = 4M_{i-1}\rho(M_{i-1}, M_i, m_{i-1})(\cos\phi_{i-1}\sin\theta_{i-1}, \sin\phi_{i-1}\sin\theta_{i-1}, \cos\theta_{i-1})^T \\ \rho &= \frac{1}{8M_{i-1}^2}\sqrt{(M_{i-1}^2 - (M_i - m_{i-1})^2) + (M_{i-1}^2 - (M_i + m_{i-1})^2)} \end{aligned} \tag{5.18}$$

The zero-component can be then deduced from the on-shellness condition. The resulting final-state momenta can be brought into the centre-of-mass frame by a Lorentz boost, which preserves flatness. The flatness of the phase space generation is, when the angles are sampled in such a way that isotropy is guaranteed, only dependent on the sampling of the intermediate masses. This can be achieved by sampling them proportional to the measure of the two body phase spaces:

$$\begin{aligned} &\mathrm{d}M_n(M_2, \ldots, M_{n-1}|M_1; m_1, \ldots, m_n) = \\ &\rho(M_{n-1}, m_n, m_{n-1}) \left(\prod_{i=2}^{n} \rho(M_{i-1}, M_i, m_{i-1})\theta(M_{i-1} - m_{i-1} - M_i)\theta\left(M_i - \sum_{k=i}^{n} m_k\right) \mathrm{d}M_i^2\right) \end{aligned} \tag{5.19}$$

where M_1 is the mass of the initial state.

In the case of massless final states, this can be written as

$$\mathrm{d}M_n(M_2, \ldots, M_{n-1}|M_1; 0, \ldots, 0) = \frac{1}{8^{n-1}} \prod_{i=1}^{n-1} \frac{M_{i-1}^2 - M_i^2}{M_{i-1}^2}\theta(M_{i-1}^2 - M_i^2)\theta(M_i^2)\,\mathrm{d}M_i^2 \tag{5.20}$$

It is always possible to write $M_i = u_2 \dots u_i M_1$. The u_i can be seen as the ratios of centre-of-mass energy being carried forward to the next decay at each step. Thus one gets

$$\mathrm{d}M_n(M_2, \dots, M_{n-1}|M_1; 0, \dots, 0) = \frac{1}{8^{n-1}} M_1^{2n-4} \prod_{i=1}^{n-1} u_i^{n-1-i}(1-u_i)\theta(1-u_i)\theta(u_i)\,\mathrm{d}u_i \quad (5.21)$$

Using $v_i = (n+1-i)u_i^{n-i} - (n-i)u_i^{n+1-i}$, this can be expressed as

$$\mathrm{d}M_n(M_2, \dots, M_{n-1}|M_1; 0, \dots, 0) = \frac{1}{8^{n-1}} M_1^{2n-4} \frac{1}{(n-1)!(n-2)!} \prod_{i=1}^{n-1} \theta(1-v_i)\theta(v_i)\,\mathrm{d}v_i \quad (5.22)$$

Substituting this into 5.16 and performing the integrations over the angels and the mass restores the earlier result for the phase space volume/weight of 5.11.

Starting from this description, it is possible to define an algorithm which performs a mapping from $[0,1]^{3n-4}$ into the flat massless n-particle phase space, which is the desired result. Other than the standard RAMBO algorithm, the structure of the generation is hierarchical due to the usage of the intermediate masses, which leads to problems for our application which are dealt with in the next section. For now, the following is the resulting flat phase space generation algorithm:

Algorithm 1: RAMBO on diet in the massless case

```
Data: random numbers $r_0, \dots, r_{3n-4}, Q_1 = E_{cm}, M_1 = E_{cm}, M_n = 0$
Result: Flat sampled momenta $\{p_1, \dots, p_n\}$ with weight $V_n$
for $i = 2, \dots, n-1$ do
    solve $r_{i-1} = v_i = (n+1-i)u_i^{n-i} - (n-i)u_i^{n+1-i}$ for $u_i$;
    $M_i \leftarrow u_2 \dots u_i M_1$;
    $\cos\theta_i \leftarrow 2r_{n-5+2i} - 1, \phi_i \leftarrow 2\pi r_{n-4+2i}$;
    $q_{i-1} \leftarrow 4M_{i-1}\rho(M_{i-1}, M_i, 0)$;
    $\sin\theta_{i-1} \leftarrow \sqrt{1-\cos\theta_{i-1}}$;
    $\mathbf{p}_{i-1} = 4M_{i-1}\rho(M_{i-1}, M_i, m_{i-1})(\cos\phi_{i-1}\sin\theta_{i-1}, \sin\phi_{i-1}\sin\theta_{i-1}, \cos\theta_{i-1})^T$;
    $p_{i-1} \leftarrow (q_{i-1}, \mathbf{p}_{i-1})$,$Q_i \leftarrow (\sqrt{q_{i-1}^2 + M_i^2}, -\mathbf{p}_i)$;
    boost $p_i$ and $Q_i$ by $\mathbf{Q}_{i-1}/Q_{i-1}^0$;
end
$p_n \leftarrow Q_n$;
```

This algorithm is also straightforwardly invertible, just as it is the case with neural importance sampling. For the massive case, two options exist. On the one hand, the boost described in the last section can be used again. On the other hand, it is possible to avoid the computationally expensive solving of 5.13 by relating the measure for the massive and the massless case. One may define $K_i = M_i - \sum_{k=1}^{n} m_k$ for $i = 1, \dots, n-1$ and $K_n = 0$. Now, it is possible to express the phase space measure of the massive final states by the phase space measure of the massless final states with modified intermediate masses, by taking the quotient of their form in 5.19:

$$\begin{aligned} &\mathrm{d}M_n(M_2, \dots, M_{n-1}|M_1; m_1, \dots, m_n) \\ &= \prod_{i=2}^{n} \frac{\rho(M_{i-1}, M_i, m_{i-1})}{\rho(K_{i-1}, K_i, 0)} \prod_{i=2}^{n-1} \frac{M_i}{K_i} \, \mathrm{d}M_n(K_2, \dots, K_{n-1}|K_1; 0, \dots, 0) \end{aligned} \tag{5.23}$$

Thus, we only gain a prefactor for the weight, depending on the masses of the final state particles.

5.3 GPU compatible implementation

The presented algorithm suffers from the problem that the intermediate masses are generated sequentially. Additionally, solving the equation for u_i is a sequential operation, too. This equation is usually solved using the bisection method, which searches for the root of the equation by testing the sign of $v-(n+2)u^{n+1}-(n+1)u^{n+2}$ at the midpoint of the interval of interest and choosing the interval in which the root is expected to lie. Due to this, the inspected interval has to be updated during each iteration, and each following iteration is dependent on the earlier one. This would strongly limit the advantages of GPU calculations when single phase space points are generated, up to the point that calculation on CPU would actually be more efficient. However, it is important to realise that all calculations during training are performed on batches. Due to this, no single phase space points but sets of phase space points are to be generated. One can use the vectorization feature of Python and rewrite the algorithm in terms matrices, such that all operations are done for all phase space points at once. For sufficiently sized batches, the calculation speed can be increased on GPU thanks to this.

The bisection however is still problematic, as not all phase space points will converge at the same speed, or are guaranteed to converge beyond a defined threshold in a reasonable time. If performed on all phase space points at the same time, this can greatly increase computational cost. For this implementation, a choice was made which tries to balance speed and precision, by stopping the bisection when the error is smaller than 10^{-16} for all phase space points or when at least 600 steps were performed.

The comparison of the runtime in figure 5.1 underlines the great advantage of the vectorized approach, not just for the GPU implementation but also for phase space generation on CPU. It is important to note that the runtime for the phase space generation on GPU is virtually only the overhead of initialising the calculation on the GPU; the runtime is, other than for the CPU generation, almost independent on the batch size. Training on the GPU is therefore highly attractive for increasing runtime and sanctions less the usage of a bigger batches, which is necessary for stability for high-dimensional target spaces.

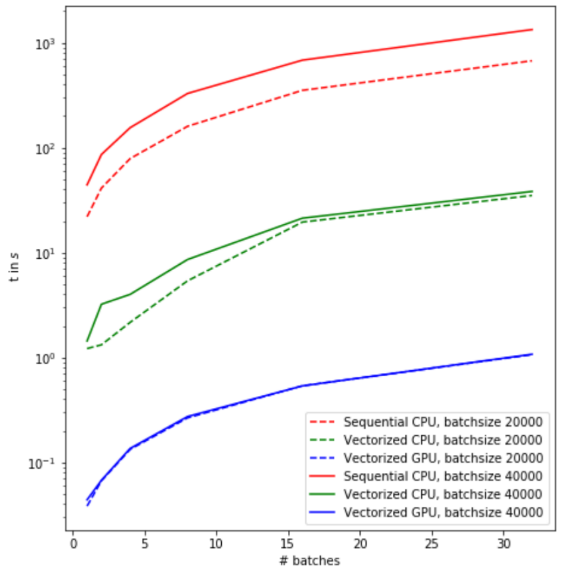


Figure 5.1 – Comparison of the runtime of different implementations for the generation for 4-particle massive final states. The batches were generated sequentially.

	Sequential PS generator	**Vectorized PS generator**
$\Delta_{max}E$	3.41E-13	9.10E-13
$\overline{\Delta E}$	8.32E-14	6.92E-14

Table 5.1 – Comparison of the maximal and average difference of 40000 runs between initial and final state energies in the phase space generator for the generation of 4-particle massive final states.

	m=0	$m = 100\,\text{GeV}^2$
Unweighting efficiency	1.0	0.061

Table 5.2 – Unweighting efficiency of 40000 runs with massless massive final and initial states at $E_{cm} = 1000\text{GeV}^2$.

The data in table 5.1 prooves that the great speedup observed earlier does not come at an unacceptable loss of precision, which is for both at the same order of magnitude and does not suffer from considerable outliers.

The present implementation supports central $2 \to n, \quad n > 1$ processes with massive or massless initial and final states. In order to provide the same functionality as MadGraph, additional features need to be included.

5.4 Phase Space Cuts

Many processes suffer from divergent matrix elements. Examples of such divergences are the Coulomb divergence for $e^+e^- \to e^+e^-$, for which divergences occur in the t-channel, or the matrix elements of three-jet processes, which diverge if one of the outgoing particles has low energy or is collinear to another outgoing particle (see for example [50]). In order to regularise the integral and achieve a finite result, MadGraph includes multiple phase space cuts in the lab frame which are applied inside the phase space generator. Therefore, to be able to come to similar results, it is necessary to include these cuts in this generator too.

The first of this is a cut in the transverse momentum for all of the outgoing momenta, which reads

$$(p_i^1)^2 + (p_i^2)^2 = (p_{i,0})^2 \sin^2\theta_i > p_{T,min}^2 \tag{5.24}$$

This enforces that the Mandelstam variable t between one of the ingoing and one of the outgoing variables is bounded from above:

$$t = (p_{in} - p_{out})^2 = -2p_{in}^0 p_{out}^0 (1 - \cos\theta_{out}) \tag{5.25}$$

for the massless case.

The next important cut is the rapidity cut. This enforces that the pseudo-rapidity

$$\eta_i = \frac{1}{2}\frac{|\mathbf{p_i}| + p_i^3}{|\mathbf{p_i}| - p_i^3} \tag{5.26}$$

is limited from above for each particle. The pseudo-rapidity is a measure describing the angle of the particle relative to the beam axis, but other than for θ differences in η are Lorentz invariant under longitudinal boost. High values of η imply a movement quasi along the beam axis. This cut is in a certain way similar to the p_T cut. However, events are also cut out which have a high enough transverse momentum but are still strongly dominated by their longitudinal momentum due to their high energy. The motivation for this cut is that events with a small angle to the beam cannot be detected by the detector, and need therefore to be filtered out by the event generator in order to produce comparable predictions.

The third cut is a cut in the angular distance

$$\sqrt{\Delta(\eta_i, \eta_j)^2 + \Delta(\phi_i, \phi_j)^2} > \Delta_{R,min} \tag{5.27}$$

between all final state particles. This ensures that none of the final state particles is collinear. Together with the cut in transverse momenta, this prevents for example the soft and collinear divergences of the three-jet cross section.

If one of the cuts becomes active for one particle or pair of particles, the corresponding event is weighted zero and does therefore not contribute to the cross section.

The comparison in table 5.3 shows that the cross sections by MadGraph, calculated with its included phase space generation, reaches identical results to the calculation using the here presented phase space generator and the matrix element standalone Fortran output of MadGraph, thus validating the correct implementation of the phase space generator.

	MadGraph 5 PS generator (pb)	**PyTorch PS generator (pb)**
LO $e^+e^- > e^+e^-$	32.710 ± 0.033	32.680 ± 0.066
LO $e^+e^- > t\bar{t}h$	$2.005 \pm 0.042 \times 10^{-3}$	$1.988 \pm 0.050 \times 10^{-3}$
LO $e^+e^- > u\bar{u}j$	$0.216\,10 \pm 0.000\,61$	$0.213\,40 \pm 0.000\,62$

Table 5.3 – Comparison of the cross section calculated from the MadGraph matrix element, either with the integrated integration tool or using the developed PyTorch PS generator and importance sampling with VEGAS. $E_{cm} = 1000\text{GeV}^2$, $p_{T,min} = 10\text{GeV}, \eta_{max} = 2.4, \Delta_{R,min} = 0.4$.

.

5.5 Hadronic Processes

Apart from the electron-positron collisions, which were investigated in table 5.3, proton-proton collisions are of great interest in particle physics. The underlying hard processes are interactions between partons (quarks of different flavors and gluons). Depending on the energy of the interaction, the different partons have different densities and carry a fraction x of the proton momentum. Due to the centre-of-mass energy of the hard process being now not fixed anymore, enhancements can happen along different channels, for example if the interactions happen via a virtual Z-boson and the centre of mass energy $(x_1 + x_2)\frac{E_{collider}}{2}$ approaches the Z-boson mass. If this is possible via the s and the t channel, for example for $d\bar{d} \to d\bar{d}$, this can lead to a case for which the VEGAS integrator has difficulties.

In order to include this behaviour in the phase space generator, it is necessary to sample to additional random numbers, one for each of the momentum fractions of the partons, which are called the Bjorken x . In order to accommodate more easily the physical integration boundaries, they are not sampled directly, but instead

$$\tau = x_1 x_2 \quad \eta = \frac{1}{2}\log\frac{x_1}{x_2} \tag{5.28}$$

which can be rewritten to

$$x_1 = \sqrt{\tau}\exp\eta \quad x_2 = \sqrt{\tau}\exp -\eta \tag{5.29}$$

The Jacobian of this transformation

$$J = \begin{pmatrix} \frac{\partial\tau}{\partial x_1} & \frac{\partial\tau}{\partial x_2} \\ \frac{\partial\eta}{\partial x_1} & \frac{\partial\eta}{\partial x_2} \end{pmatrix} = \begin{pmatrix} x_2 & x_1 \\ \frac{1}{2}\frac{1}{x_1} & -\frac{1}{2}\frac{1}{x_2} \end{pmatrix} \tag{5.30}$$

has a determinant of 1, such that no additional variance is induced. The boundaries of the sampling are

$$\begin{aligned} \frac{\max(E_{cm,min}, \sum_{i=0}^{n} m_i)}{E_{cm}} \leq& \tau \leq 1 \\ \frac{1}{2}\log\tau \leq& \eta \leq -\frac{1}{2}\log\tau \end{aligned} \tag{5.31}$$

	MadGraph 5 PS generator (pb)	**PyTorch PS generator (pb)**
QCD LO $uu > uu$	$3.508 \pm 0.012 \times 10^5$	$3.56 \pm 0.18 \times 10^5$
QCD LO $ug > ug$	$4.236 \pm 0.014 \times 10^6$	$4.21 \pm 0.11 \times 10^6$
QCD LO $ug > ugg$	$2.209 \pm 0.068 \times 10^5$	$2.27 \pm 0.12 \times 10^5$
QCD LO $u\bar{c} > u\bar{c}g$	$3.425 \pm 0.010 \times 10^3$	$3.425 \pm 0.073 \times 10^3$

Table 5.4 – Comparison of the cross section calculated from the MadGraph matrix element, either with the integrated integration tool or using the developed PyTorch PS generator and importance sampling with VEGAS $E_{cm} = 1000\text{GeV}^2$, $p_{T,min} = 10\text{GeV}, \eta_{max} = 2.4, \Delta_{R,min} = 0.4$, PDF set NNPDF23_nlo_as_0119.

.

which induce flat volume factors to the weights. $E_{cm,min}$ is the lower energy boundary until which a process is seen as a hard process. We fix it to 1GeV^2 for the rest of this work. Taking into account the Bjorken x, the interaction is now not anymore between two particles with momenta p_1 and p_2, $\mathbf{p_1} = -\mathbf{p_2}$, but instead with $p_1' = x_1 p_1$ and $p_2 = x_2 p_2$. Although the incoming momenta are now not anymore in the centre-of-mass frame, an additional Lorentz boost is performed in the end so that the output is indeed in the centre-of-mass frame.

With the information of the Bjorken x and the nature of the incoming particles, it is now possible to include the value of the PDFs in the weights. This will be done by interfacing LHAPDF [51], with which we will in the following access the NLO PDF set NNPDF23_nlo_as_0119 [52]. LHAPDF extrapolates the value of the PDF for the incoming particles based on their flavor, momentum fraction and the renormalization scale (chosen in the following to be the Z-mass, which introduces an additional considerable computational workload done exclusively on the CPU.

We would like to validate again that this implementation leads to the same cross-sections in comparison to using the phase space generation and integration included in MadGraph. Therefore, table 5.4 contains a comparison of the results.

Chapter 6

Neural importance sampling for cross section integrations

After investigating different methods of neural importance sampling, the importance of hyper-parameters, the behaviour for complex structures in higher dimensions and efficient methods of mapping between the output of the importance sampling and the momenta, which are the input of matrix elements, it is now time to investigate which advantages neural importance sampling has in the current implementation for the integration of matrix elements.

6.1 Modification of the loss function

In a first step, an important change has to be made to the loss function in order to be able to safely train any matrix element. Starting with our investigation with LO process $e^+e^- \to \mu^+\mu^-$, it became clear that the variance proxy used for the training so far, although less prone to statistical errors, is insufficient to guarantee successful training.

The differential cross section of this process is proportional to $(1 + \cos^2\theta)$. Naively, one would expect the same training behaviour for $\cos^2\theta$ and $(1 + \cos^2\theta)$ as the peak size and position are identical. Other than the test functions used before, this integrand does however not reach a value close to zero, neither does the loss function approach zero but a constant. A constant offset leads to a term in the variance proxy proportional to the mean of the Jacobian squared, which favours small values for the Jacobian, especially if the non-constant part of the function is of a small value. This is the reason why the training with the variance proxy fails for $(1+\cos^2\theta)$. Calculating instead the sample variance on the batch does eliminate the effect of the constant offset, as now a small variance of the Jacobian is favoured. With this loss function, indeed an identical training behaviour is observed for the case with and without the offset.

A second observed issue is related to the scaling of the integrand. For small scales of the integrand, the training suffers from strong instabilities. Although $\cos^2\theta$ and $1 \times 10^{-20}\cos^2\theta$

have the same peak structure, the latter is less likely to train successful. This is related to numerical errors and instabilities in the calculation of the loss. In order to compensate this, the integrand is rescaled by its maximal observed value during the evaluation of the function. This can be done as the rescaling does not change the position and ratio of peaks.

With these modifications in place, it is possible train any matrix element which is passed to the neural importance sampling. However, they introduce additional instability to the algorithm as neither the correct rescaling factor nor the correct mean are known. The question of interest is now if an improvement over other importance sampling methods can be reached also for this integrands.

6.2 Comparison of the neural importance sampling and VEGAS

In the following, the effect of the neural and VEGAS importance sampling on the variance of MadGraph matrix elements in the Fortran output are investigated for different standard model processes at $E_{cm} = 1\,\text{TeV}$. The PyTorch phase space generator will be used, including the PDFs when necessary. The cuts are fixed as $p_{T,min} = 10\,\text{GeV}$, $\eta_{max} = 2.4$ and $\Delta_{R,min} = 0.4$. For the comparison with VEGAS, the VEGAS algorithm is optimised until it gives virtually no variance improvement. The setup of the neural importance sampling is gained by doing a hyperparameter optimisation similar to before. As we could show that the piecewise-quadratic coupling cells are superior, only they will be investigated in the following. The hyperparameters of the batch size, the length and width of the neural network, the number of bins, the learning rate, weight decay and the preburn time are chosen from a wide spectrum to find the optimal setup for the variance reduction. The setup is found by taking the setup with the smallest variance amongst 75 generated configurations. The optimal configuration is then trained 10 times, in order to see how stable its performance is.

6.2.1 $e^+e^- \to \mu^+\mu^-$

As mentioned before, the LO matrix element of this process can be expressed in an especially simple way which is free of any peaks due to the fixed centre-of-mass energy in interactions of electron-positron beams. The number of degrees of freedom necessary to generate a phase space point are 2. Comparing therefore the properties of this matrix element with the behaviour observed in Chapter 4, no improvement over VEGAS is to be expected.

Indeed, table 6.1 and figure 6.1 show that no improvement could be achieved over VEGAS, which can handle very well uncorrelated low dimensional situations. Neural importance sampling achieves a stable improvement, with minor dependency on the hyperparameters, but is not able to outmatch VEGAS. It is therefore more interesting to look into processes which have a peaked structure.

Original variance	**NIS**		**VEGAS**		**# SD**	**VRF ratio**
	VRF	**# Eval**	**VRF**	**# Eval**		
2.63×10^{-20}	$5.7 \pm 1.6 \times 10^{-2}$	$1.45 \pm 0.57 \times 10^{7}$	$1.49 \pm 0.44 \times 10^{-2}$	$2.30 \pm 0.12 \times 10^{6}$	0.12	3.840

Table 6.1 – Comparison of the performance of the best NIS setup and VEGAS for LO $e^+e^- \to \mu^+\mu^-$. VRF stands for the mean reduction factor of the variance.

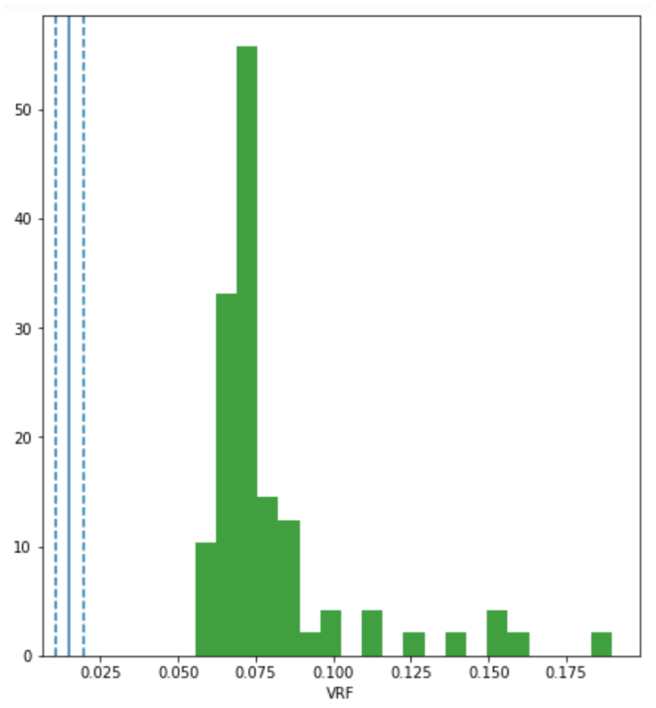


Figure 6.1 – Histogram of the performance of the different setups during hyperparameter optimisation for $e^+e^- \to \mu^+\mu^-$. The blue line is the variance reduction achieved by VEGAS with its standard deviation.

6.2.2 $gu \to gu$

Investigating proton-proton collision introduces the PDFs and by this peaked behaviour in the dimension of the Bjorken x. The parton distribution functions have a characteristic shape, increasing sharply at low Bjorken fractions and could benefit from applying suitable importance sampling. The first process investigated here is the LO in QCD of $gu \to gu$. As the light quarks are assumed massless at this energy scale, no propagator peaks are present. It is nevertheless interesting to see how neural importance sampling behaves in this slightly more complex situation.

Table 6.2 and figure 6.2 show that here indeed neural importance sampling can gain a small advantage, although great improvement is rare and suffers from instabilities. It shows however

Original variance	**NIS**		**VEGAS**		**# SD**	**VRF ratio**
	VRF	**# Eval**	**VRF**	**# Eval**		
0.048	$3.8 \pm 3.3 \times 10^{-3}$	$1.93 \pm 0.52 \times 10^{7}$	$1.24 \pm 0.22 \times 10^{-2}$	$1.26 \pm 0.38 \times 10^{6}$	0.73	0.27

Table 6.2 – Comparison of the performance of the best NIS setup and VEGAS for QCD LO $gu \to gu$. VRF stands for the mean reduction factor of the variance.

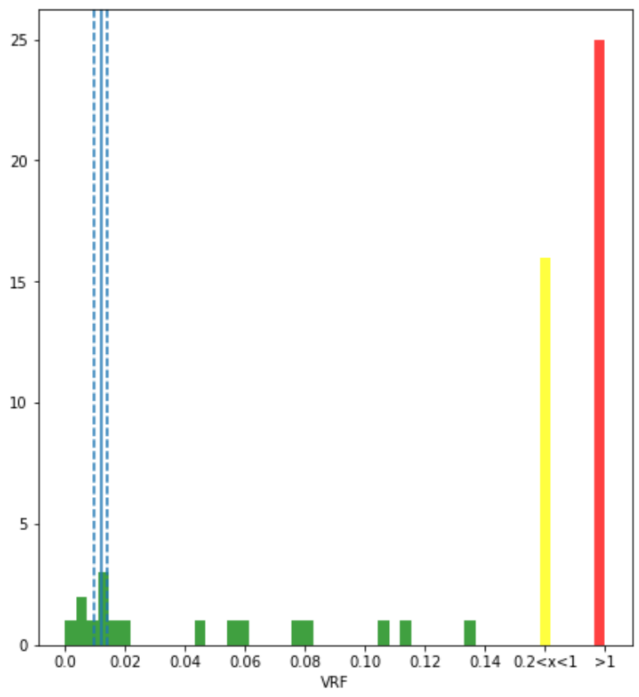


Figure 6.2 – Histogram of the performance of the different setups during hyperparameter optimisation for $gu \to gu$. The blue line is the variance reduction achieved by VEGAS with its standard deviation. The yellow bin refers to setups which could achieve no variance better than 20% of the original value, the red bin refers to setups which achieved no improvement.

that neural importance sampling, under the the proper choice of the hyperparameters, can reach competitive results to VEGAS.

6.2.3 $uc \to ucg$

Similar results are expected for a three-jet cross-section in LO QCD. The additional degrees of freedom are however a potential issue for the VEGAS algorithm, as it has been shown before that it performs worse in higher dimensions. Apart from this, the phase space cuts gain increased importance when three final state particles are present, as for example the angular distance between two jets can reach zero and lead to divergences. The phase space cuts introduce discontinuous behaviour, which is potentially a challenge both VEGAS and for the neural importance sampling.

Original variance	**NIS**		**VEGAS**		**# SD**	**VRF ratio**
	VRF	**# Eval**	**VRF**	**# Eval**		
5.87×10^{-8}	$9.4 \pm 4.3 \times 10^{-3}$	$1.6 \pm 1.0 \times 10^{7}$	$7.1 \pm 2.2 \times 10^{-2}$	$4.50 \pm 0.63 \times 10^{6}$	0.98	0.13

Table 6.3 – Comparison of the performance of the NIS and VEGAS for QCD LO $uc \to ucg$. VRF stands for the mean reduction factor of the variance.

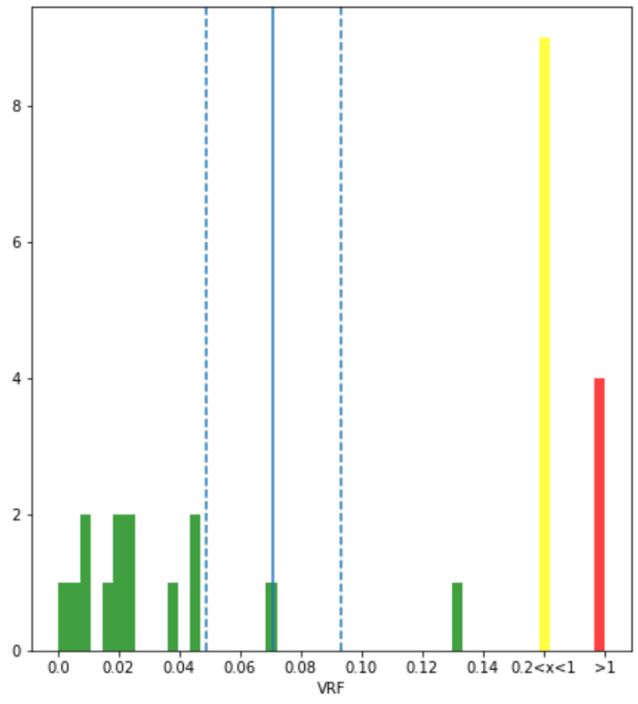


Figure 6.3 – Histogram of the performance of the different setups during hyperparameter optimisation for $uc \to ucg$. The blue line is the variance reduction achieved by VEGAS with its standard deviation. The yellow bin refers to setups which could achieve no variance better than 20% of the original value, the red bin refers to setups which achieved no improvement.

Table 6.3 and figure 6.3 show that neural importance sampling indeed can outperform VEGAS also for this case, and with a greater improvement than for the 2 jet example. However, the price is an increased variance of the quality of the results.

6.2.4 $d\bar{d} \to d\bar{d}$ via Z

For this process, two diagrams exist: one s-channel and one t-channel diagram. As the Z boson is massive, the s channel has a peak near the Z mass whereas the t channel has a peak near the vanishing transverse momentum. Other than before, we use here $p_T = 0.1$ in order to test the full peak. When the Bjorken x are sampled directly and not via the τ and η, these peaks are not aligned to a single coordinate axis. Looking at the variance reduction in table 6.4 only would lead to the conclusion that VEGAS is superior.

It is important to mention however that the improvement VEGAS reaches over the neural importance sampling comes at the cost of a greatly increased number of function evaluations.

Original variance	NIS		VEGAS		# SD	VRF ratio
	VRF	# Eval	VRF	# Eval		
7.37×10^{-12}	$2.5 \pm 1.2 \times 10^{-1}$	$1.14 \pm 0.21 \times 10^{6}$	$9.8 \pm 2.8 \times 10^{-2}$	$2.75 \pm 0.57 \times 10^{7}$	0.79	2.61

Table 6.4 – Comparison of the performance of the NIS and VEGAS for $d\bar{d} \rightarrow d\bar{d}$ via Z. VRF stands for the mean reduction factor of the variance.

If a too small number of evaluations during each iteration is chosen, VEGAS does not properly recognise the peak structure and introduces systematic errors. This does greatly increase the computational cost in comparison to neural importance sampling, for which systematic errors are less likely to occur. Indeed, comparing the value of the cross section for the mentioned configuration after integrating only over 3.6×10^6 points in table 6.5 and 6.6 shows that the VEGAS precision is misleading, as high systematic errors prevail.

	MadGraph 5	MC w/o importance sampling	NIS
σ in pb	40.740 ± 0.054	39.1 ± 5.7	40.52 ± 0.35

Table 6.5 – Overview of the result for $d\bar{d} \rightarrow d\bar{d}$ via Z for different integrators.

	VEGAS with 3×10^6 eval.	VEGAS with 6×10^6 eval.	VEGAS with 9×10^6 eval.
σ in pb	42.24 ± 0.25	41.03 ± 0.17	40.43 ± 0.14

Table 6.6 – Overview of the result for $d\bar{d} \rightarrow d\bar{d}$ via Z for VEGAS at different values for the number of function evaluations.

Although neural importance sampling does require expensive training in advance, including often hyperparameter optimisation, it allows importance sampling with a lower risk to loose the information about the peaks and by this about the correct value.

The output of the projection of the trained model in two dimensions in comparison to the interpolated projection of the value of the integrand helps to understand how neural importance trains these complex structures. It is important to realise that this integrand suffers from multiple orders of magnitude difference in value, originating from the PDFs as well from the propagator peaks. As a consequence, it is considerably harder to train than the examples studied in the last chapter. It is therefore interesting to look at the structures which were recognised by the training.

Figure 6.4 shows that there is, as expected, a strong improvement of the integrand for small Bjorken x, as the PDFs reach very high values here, and a preference for a high value of x_0 as this minimises t. Looking at the trained mapping, although it did not train perfectly, especially the peak for the intersection of both improvements was detected as well as a generally increased likelihood for small t. However, the histogram shows also the limitation the difference of three orders of magnitude could not be reflected by the mapping, as in almost all regions of the $(x_0 - x_2)$-plane there is still a considerable chance to be sampled in. High differences in the integrand value seem to be a limiting factor of the efficiency of this integration strategy.

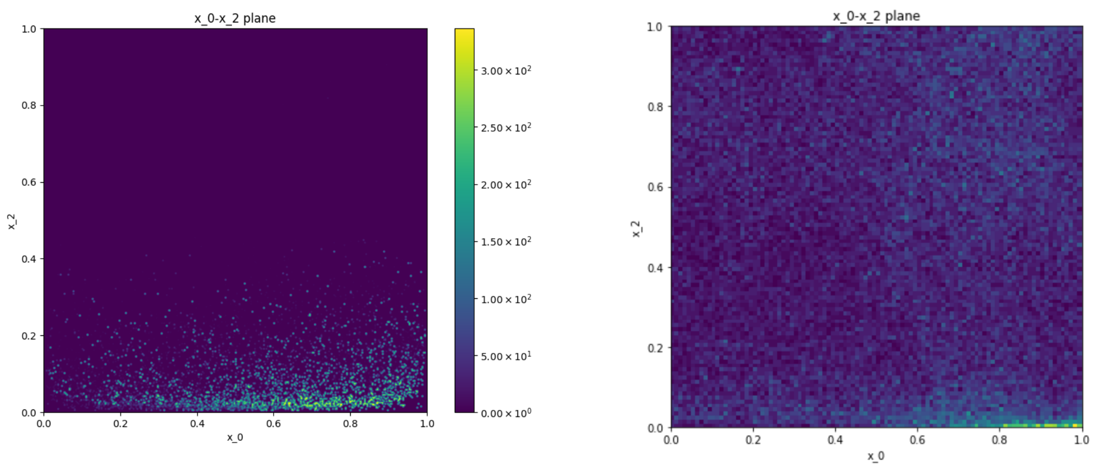


Figure 6.4 – The $(x_0 - x_2)$ plane, referring to θ and the Bjorken x of the d, of the integrand (left) and the trained model (right).

The most interesting structure can be observed in the plane of both Bjorken x, as here the solution of $x_d x_{\bar{d}} s = m_z^2$ is a peak, which has both a part parallel to the x_2 and the x_3 axis.

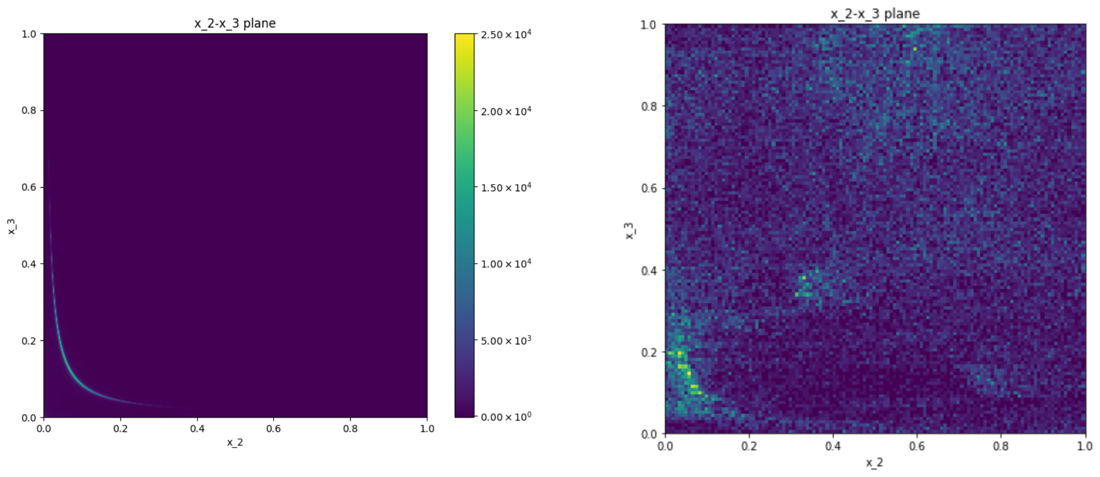


Figure 6.5 – The plane of the Bjorken x of the integrand (left) and the trained model (right).

Figure 6.5 shows that similiar to before, the main peak is recognise, but residual noise exists all over the plane. It is an important observation that the position and form of the peak is very well recognised.

In a last step, these observations are compared to the VEGAS grid.

Figure 6.6 shows that both discussed peaks are recognised by VEGAS. For the s-peak, the form of the peak can not be represented and is thus only optimised closely around its highest value. The peak in t, improved by the effect of the PDF, shows that the peak along the x_2-axis is at the same height as in the (x_2, x_3)-plane as each axis is improved separately and thus taking not into account the different position of the peak in x_2 in both planes . In this case, this leads to a small deviation of the optimal position which is, as seen in comparison to 6.4. This is something

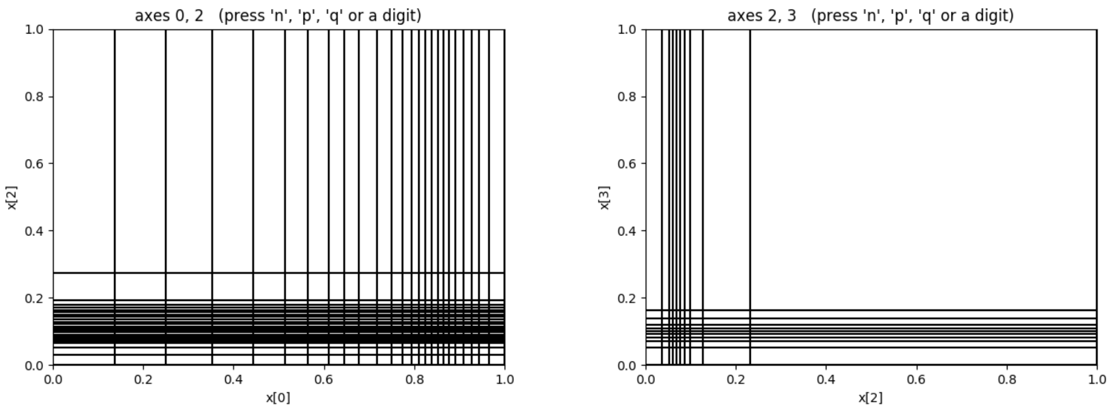


Figure 6.6 – Both planes in the VEGAS grid.

which neural importance sampling can accommodate more easily. Thus it becomes clear that, if the peaks can be represented more clearly, that neural importance sampling can adapt to this process better.

Conclusions and outlook

The goal of this thesis was the introduction of machine learning techniques and their application to the calculation of cross sections. Machine learning was used in order to perform an importance sampling which induces a change of variables such that the efficiency of Monte Carlo integration is increased. This is possible as neural networks are non-linear function approximators, and are therefore very well suited for learning the integrand during integration in order to improve the result.

This method of choice for this neural importance sampling was one using normalizing flows, which allows invertability as well a simple calculation of the Jacobian of the transformation. Different implementations of this normalizing flows were discussed: the affine, the piecewise-linear and the piecewise-quadratic coupling cells. Starting from two dimensional problems, their training process was discussed. It became clear that the piecewise-quadratic coupling cells are superior due to their higher flexibility in representing the integrand. As a benchmark for the quality of the results of this algorithm, the VEGAS importance sampling was used. This approach does not require prior knowledge of the integrand as long as no correlations along the coordinate axes are present.

In a subsequent step, the behaviour of the piecewise-quadratic coupling cells was studied more in detail. In a wide range comparison run, optimal configurations were searched for the multidimensional multi-peak gaussians up to dimension 32. Hyperparameter optimisation is necessary in order to achieve results of high quality, as it is strongly dependent on finding one of the optimal sets of parameters. It was shown that wide and short neural networks are preferred, and that enforcing high learning rates gives rise to models which train faster. The main result of this explorative run was that neural importance sampling is inferior to VEGAS in in the presence of non-correlated functions, as VEGAS performs well there. This makes clear that neural importance sampling is at the present not suited as a general replacement of other methods of importance sampling, but is rather an alternative in cases which are otherwise hard to handle. It became also evident that the stability of the training decreases if the points in the sampled batch are scarce in the integration volume.

After this general work on neural importance sampling, the next aim was to apply it to cross sections. A phase space generator was needed, which should introduce as little additional variance as possible. The flat phase space generator RAMBO on diet was implemented in such a that a batch of flatly distributed phase space points can be generated on GPU much faster than on CPU. With this in place, it was now possible to interface the MadGraph 5 output. Special attention was given to the $d\bar{d} \to d\bar{d}$ via Z process as it suffers from peak structures which are hard to deal with using regular VEGAS importance sampling. Although VEGAS could yet not be outperformed in terms of pure variance reduction, a considerably lower number of function evaluations was needed in order to achieve the improvement.

In the light of these results, there are still multiple areas of improvement for this approach. Due to the great dependency on hyperparameter optimisation, this is a point where both speed and quality of the result can be substantially improved. Using more sophisticated approaches of hyperparameter optimisation than random search, for example Bayesian or gradient-based optimisation, can effect in a better ratio between tested setups and reached optimisation. The same can also be achieved when investigating which are typical values (such as values of 5×10^{-2} to 5×10^{-4} for the learning rate) for the hyperparameters, in order to reduce the size of the parameter space. For the experiments here, a wide range was used as the typical values for optimal setups are yet unknown. Apart from this, it is maybe also necessary to define a better way of rating setups. Using the optimal achieved variance only is very sensitive to statistical errors and instable models.

One important improvement can be achieved when implementing the inverse model. The normalizing flows are straightforwardly invertible, as well as the RAMBO on diet algorithm. Defining a training set of momenta and the corresponding value of the matrix element allows, after mapping the momenta into the unit hypercube, to train on the inverse, which saves the evaluation of the function during optimisation and reduces statistical errors. Both would greatly increase the usability of this approach.

An interesting point of investigation would be also to use a different loss function. The presented implementation uses the variance or a variance proxy for training. Although this is the most natural choice when aiming at improving the efficiency of a Monte Carlo integration, there are other natural choices which could give different results, for example the Kullback-Leibler divergence between the uniform distribution and the product of the function value and the Jacobian. Both should provide the same minimum, but do not necessarily have the same training behaviour.

The deciding factor for the success of neural importance sampling is however if indeed a gain in the integration speed at a fixed precision can be achieved. In comparison to VEGAS, the computational cost is due to the rich structure of the quadratic coupling cells higher. In general, the analysis of Chapter 4 shows also that neural importance sampling needs a comparable number of function evaluations to reach the optimal variance. However the numerical cost of the hyperparameter optimisation is still a great flaw. Whereas possible solutions for this had been already discussed, the runtime of a single setup has especially for the cross section still a great potential of optimisation. As pointed out in the first chapter, this can indeed be easily compensated by a small decrease in variance, as doubling the precision requires 4 times the amount of evaluations.

A question which stays is if these possible improvements can help to outperform VEGAS or even the more sophisticated methods of multi-channelling in MadGraph, or if these can be combined with the neural importance sampling in order to improve the overall performance and to provide a general integrator for cross sections. It would be a great advantage if the performance of VEGAS could be reached also for integrals where VEGAS is performing well, as neural importance sampling could then possibly replace VEGAS as a general process-independent integrator. However, in order to achieve this, neural importance sampling has to be studied more in detail.

Appendix

Implementation

In this Appendix, the most important classes and functions of the implementation presented in the earlier chapters are outlined. The tools were written in Python using the PyTorch framework, and are compatible both with Python 2 and Python 3.

Neural importance sampling

We focus here on the the class of the piecewise-quadratic coupling cells, as these were in the focus of this thesis due to their superior performance.

class PWQuad

A daughter class of torch.nn.Module, the basic Pytorch class for neural network models. This is the central class for the piecewise-quadratic coupling cells. It represents a single piecewise-quadratic coupling cell. The constructor takes the following arguments:

flow_size: The number of transformed dimensions.
pass_through_size: The number of preserved dimensions.
n_bins: The number of bins for the quadratic interpolation.
NN_layers: A list giving the number of nodes for each layer of the neural network.

The neural network of the coupling cell is an instance of the following class:
class RectNN

This is itself also daughter class of torch.nn.Module. It describes a rectangular neural network consisting out of a batch normalisation layer in the beginning, followed by multiple sets of a linear, a batch normalisation and a ReLU layer. The final layers are a linear layer (in order to be able to reach the whole $\mathbb{R}$) and a reshaping layer. This class is used to construct the neural network used by the coupling cell. The initialiser takes the following arguments:

> ***pass_through_size***: The number of preserved dimensions.
> ***sizes***: A list of the number of output nodes of the linear layers. The length of the list determines the length of the rectangular neural network.
> ***reshape***: A tuple containing the information of the required shape of the output.

class PWQuadManager

A daughter class of BasicManager, which contains the training logic. It is initialised with n_flow , the number of dimensions of the training input. A similiar class exists for the piecewise-linear coupling cells. After initialising the manager, the following methods perform the training:

create_model*(n_cells, n_bins, NN, dev)*:

> ***n_cells***: The number of coupling cells. If the number is smaller than what is required for the training dimension, it will be automatically adjusted to the minimum required number. If more than the minimum number of coupling cells are requested, additional coupling cells are added which transform one dimension each.
> ***n_bins***: The number of bins for the quadratic interpolation.
> ***NN***: A list giving the number of nodes for each layer of the neural network.
> ***dev***: An integer specifying which GPU should be used. Default to 0. If no GPU is available, the argument is not necessary and the CPU will be used.

This method creates the coupling cells and connects them in a sequential model such that each dimension is transformed never twice at the same time as any other dimension

_train_variance_forward*(f, optimizer_object,log, logdir, batch_size, epochs, epoch_start, pretty_progressbar, save_best, run,dev,mini_batch_size, integrate, preburn_time, kill_counter, impr_ratio, loss_mode)*:

> ***f***: The integrand which is to be trained. The only requirements are that it takes arguments of shape $a\times$ n_flow, where a is a variable length (the batch_size) and returns a 1D-tensor of length a containing the results
> ***optimizer_object***: This is a PyTorch optimizer which is used for the gradient descent, initialised with the chosen learning rate.
> ***log***: If this is true, the initial and best model are saved as a pickle on the disk, as well as additional information about the run (see ***save_best***). Default is True.
> ***logdir***: Relative path for saving the model. Default is None.

batch_size: The number of function evaluations per epoch. Default is 10000.
epoch: The maximal number of epochs used for training. Default is 1000.
epoch_start: The starting point of counting the epochs, useful if an existing model is to be refined. Default is 0.
pretty_progressbar: If enabled, displays progressbars with the current progress of the training. Default is True.
save_best: If enabled, the best model (defined as the state of the model when the loss was minimal) is saved under self.best_model. Additional information, like the number of function evaluations, the initial and best loss etc. are saved in their respective fields and can be called from the fields of the manager after the training is completed. Default is True.
run: The run ID if hyperparameter optimisation is performed with a experiment observer of the Python SACRED package. Default is None.
dev: An integer specifying which GPU should be used. Default to 0. If no GPU is available, the argument is not necessary and the CPU will be used.
mini_batch_size: Has to be smaller then the batch size, otherwise it is set identical to the batch size. If smaller than the batch size, the batch is divided in equal sized mini-batches. In each epoch, ⌊batch_size/mini_batch_size⌋ evaluations are performed. The advantage of the usage of minibatches is the reduced memory consumption at the price of increased runtime.
integrate: If True , integrates during each step with the current model, and after the training stopped for each leftover epoch. The result is the weighted average using the inverse variance as weight. The usage is discouraged for small batch sizes as fluctuations of the model during the training can lead to systematic errors. Default is False. If it is active, a tuple of the integration result and its error are returned, otherwise the tuple is set to (0,0).
preburn_time: The number of epochs for which at the beginning of the training, not the mapped input variables are used but the original ones, together with the Jacobian of the mapping. By this, the Jacobian is trained on a uniform target space. The preburner is stopped early if the loss has fallen under a fourth of the initial loss. The preburn time is also used as a characteristic epoch time for the early stopping regularization. Default is 75.
kill_count: The number of epochs during which the loss is allowed to increase in a row before the training is stopped. If the preburner is still active, normal training starts and the counter is reset. Default is 7.
impr_ratio: $1 - Q$, with Q the ratio between the current best loss and the best loss n epochs before. n is the characteristic time of the training, which is either ***preburn_time*** or, if this is smaller than 10, set to 50. Default is 1e-2.
loss_mode: If equal to "var", the batch variance is used as a loss function. If equal

to "est", the loss proxy (without the estimate of the integral value) is used. Default is "var".

integrate*(f, nitn, neval, dev)*:

f: The integrand which is to be trained. The only requirements are that it takes arguments of shape $a\times$ n_flow, where a is a variable length (the batch_size) and returns a 1D-tensor of length a containing the results.
nitn: The number of subsequent integration steps.
neval: The number of function evaluations per step.
dev: The id of the GPU to use. If no GPU is available, CPU is used. Default is 0.

Performs a Monte Carlo integrations sampled by the best model, and returns the weighted (inverse variance weight) average of the ***nitn*** steps with the integration error.

Phase space generation

class FlatInvertiblePhaseSpace

A daughter class of VirtualPhaseSpaceGenerator and shares the same initialiser:

initial_masses: A list containing the initial masses.
final_masses:A list containing the final masses. Its length determines the size of the phase space.
pdf: A LHAPDF object, constructed from the LHAPDF Python bindings. Default is None.
pdf_active: Determines if PDFs should be used. Default is False.
tau: Determines how the integration over the Bjorken is parametrized. Default is True, meaning that not the Bjorken x directly but τ and η are sampled.

This class provides the logic of creating sets of final momenta. The most important methods are:

nDimPhaseSpace*()*:
Returns the dimensionality of the phase space.

bisect_vec_batch*(v_t, target, maxLevel)*:

v_t: The vector of v values. See 5.21.
target: The target accuracy. Default is 1e-16.
maxLevel: The maximal number of iteration steps. Default is 600.

Uses a vectorized bisection in order to solve 5.21. Returns *u*.

get_pdfQ2*(pdf, pdg, x, scale2)*:

> ***pdf***: The LHAPDF object.
> ***target***: The PDG code of the ingoing particle. If it is an unkonwn code, the return value is 1.
> ***x***: A 1D-tensor containing the Bjorken x.
> ***scale2***: The squared renormalization scale.

Returns the value of the PDF for the ingoing particle without the factor of x.

generateIntermediatesMassless_batch*(M, E_cm, random_variables)*:

> ***M***: A tensor of size batchsize$\times(n-1)$ with n the number of final state particles, initialised with the centre-of-mass energy at the first entry.
> ***E_cm***: The 1D-tensor of the centre-of-mass energy.
> ***random_variables***: The random variables to-be-mapped to the phase space.

Generate intermediate masses for a massless final state, following the algorithm described in chapter 5. Returns a weight-tensor with the flat weights.

generateIntermediatesMassive_batch*(M, E_cm, random_variables)*:

> ***M***: A tensor of size batchsize$\times(n-1)$ with n the number of final state particles, initialised with the centre-of-mass energy at the first entry.
> ***E_cm***: The 1D-tensor of the centre-of-mass energy.
> ***random_variables***: The random variables to-be-mapped to the phase space.

First calls the generation of the intermediate masses, then modifies them for the massive case. Returns a weight-tensor.

setInitialStateMomenta_batch*(output_momenta, E_cm)*:

> ***output_momenta***: A tensor of size batchsize$\times(2+n)\times 4$, with n the number of final state particles.
> ***E_cm***: The 1D-tensor of the centre-of-mass energy.

Sets the initial state momenta according to the centre-of-mass energy of the system.

get_flatWeights*(E_cm, n)*:

> ***E_cm***: The 1D-tensor of the centre-of-mass energy.
> ***n***: The number of final particles.

Initialises the weights to the weights of the flat, massless case. Returns the weight.

generateKinematics_batch(*E_cm, random_variables_full, pT_mincut, delR_mincut, rap_maxcut, pdgs)*:

> ***E_cm***: The 1D-tensor of the centre-of-mass energy.
> ***random_variables_full***: The tensor of random numbers. If the PDF mode is active, the last two entries for every batch element contain the information for the Bjorken x.
> ***pT_mincut***: The minimal value for pT (transverse momentum) for any of the final state particles. If pT is lower, the weight of this point is set to zero. Default value is -1 (no cut).
> ***delR_mincut***: The minimal value for Δ_R between any two particles. If Δ_R is lower, the weight of this point is set to zero. Default value is -1 (no cut).
> ***rap_maxcut***: The maximal value for η of any particles. If η is lower, the weight of this point is set to zero. Default value is -1 (no cut).
> ***pdgs***: A two element list with the PDG codes of initial state particles. Default is $[0, 0]$ which is equivalent of a non-parton particle.

After checking for the validity of the entered data, the centre-of-mass energy is adjusted if the PDFs are active. After this, the phase space is generated according to the "RAMBO on diet"-algorithm (see Chapter 5). The weights are agglomerated, and the PDFs are included in the weight tensor. The cuts are checked in the lab frame. If a phase-space point fails the cut criteria, its weight is set to zero. Afterwards, the momenta are boosted into the centre-of-mass frame. Returns a tuple of (*output_momenta*, *weight*), where *output_momenta* is a batchsize$\times(2+n)\times 4$-Tensor and *weight* is a 1D-Tensor of the length of the batchsize. A single phase space point can be generated by setting the batchsize equal to 1.

Results of the hyperparameter optimisation

The following tables contain an overview of the results of the hyperparameter optimisation in chapter 4.

# peaks	Peak width	Original variance	NIS		VEGAS		# SD	VRF ratio	rel. speedup
			VRF	# Eval	VRF	# Eval			
1	**0.15**	3.033E-02	2.952E-02± 4.86E-03	1.810E+06 ± 4.33E+05	6.497E-04 ± 1.31E-05	3.54E+06 ± 4.32E+05	1.07E-01	4.543E+01	2.22E-02
	0.25	6.103E-02	2.838E-02 ± 4.84E-03	4.320E+06 ± 6.54E+05	3.705E-04 ± 1.02E-04	4.00E+06 ± 1.32E+06	9.43E-02	7.512E+01	1.33-02
	0.35	6.694E-02	3.591E-02 ± 1.34E-03	1.8100E+07 ± 8.16E+05	6.1832E-04 ± 3.97E-05	3.51E+06 ± 1.76E+06	2.23E-01	5.80E+01	1.72E-02
2	**0.15**	5.132E-02	1.726E-01 ± 4.27E-02	2.590E+06 ± 2.97E+05	3.7519E-01 ± 1.13E-03	2.57E+06 ± 2.38E+05	2.63E-01	4.397E-01	2.32
	0.25	8.401E-02	4.517E-02 ± 7.54E-03	1.440E+06 ± 2.48E+05	7.5937E-01 ± 3.11E-03	2.140E+06 ± 1.39E+05	5.08E-02	5.948E-02	1.68E+01
	0.35	7.947E-02	1.947E-02 ± 3.99E-03	3.845E+06 ± 7.49E+05	8.4351E-01 ± 4.75E-03	1.634E+06 ± 1.23E+05	3.84E-01	2.261E-02	4.83E+01
4	**0.15**	6.348E-02	4.085E-02 ± 2.50E-03	1.433E+07 ± 1.30E+06	5.8347E-04 ± 2.14E-05	4.519E+06 ± 3.21E+05	1.03E+01	6.85E+02	2.13E-04
	0.25	4.242E-02	5.509E-02 ± 2.06E-02	2.8800E+06 ± 9.27E+04	8.59E-04 ± 1.01E-03	3.674E+06± 4.45E+05	1.10E-03	6.415E+01	2-43E-04
	0.35	6.530E-02	6.331E-02 ± 2.73E-03	2.910E+06 ± 1.30E+05	8.705E-04 ± 1.04E-03	4.12E+06± 1.02E+06	3.51E-01	7.313E+02	1.36E-02

Table 6.7 – Comparison of the performance of the PWQuad cells with VEGAS for 2 dimensions. VRF stands for the mean reduction factor of the variance.

# peaks	Peak width	Original variance	NIS		VEGAS		# SD	VRF ratio	rel. speedup
			VRF	# Eval	VRF	# Eval			
1	**0.15**	1.242E-03	1.343E-02 ± 5.41E-03	3.915E+06 ± 9.44E+05	1.5616E-04 ± 2.30E-06	3.412+06±2.46E+05	1.10E-01	8.571E+01	1.17E-02
	0.25	8.163E-02	4.108E-02 ± 4.67E-03	6.140E+06 ± 7.04E+05	3.1260E-04 ± 1.05E-05	4.43E+06±1.03E+06	4.99E-02	1.32E+02	7.69E-02
	0.35	2.098E-02	7.337E-02 ± 9.31E-03	6.08E+06 ± 1.06E+06	8.659E-04 ± 1.95E-05	4.1275+06±1.38E+05	1.50E-01	8.473E+01	1.19E-02
2	**0.15**	2.524E-03	1.809E-02 ± 1.69E-03	4.310E+06 ± 7.50E+05	2.6971E-01 ± 1.44E-03	3.254E+06±2.46E+05	5.41E-01	6.706E-02	1.49E+01
	0.25	2.617E-02	6.70E-02 ± 1.11E-02	1.540E+06 ± 2.04E+05	7.6272E-01 ± 4.14E-03	2.135E+06±3.34E+05	7.34E-02	8.790E-02	1.15E+01
	0.35	3.193E-02	9.286E-02 ± 7.06E-03	4.67E+06 ± 1.06E+06	9.1758E-01 ± 3.84E-03	6.23E+05±1.34E+05	3.32E-01	1.012E-01	9.90
4	**0.15**	4.395E-03	3.262E-02 ± 4.41E-03	2.810E+06 ± 2.70E+05	2.39650E-01 ± 6.52E-04	3.274E+06±4.53E+05	1.23E-01	1.361E-01	7.35
	0.25	2.187E-02	7.021E-02 ± 6.18E-03	4.72E+06 ± 1.12E+06	7.0582E-01 ± 2.23E-03	2.503E+06±2.348E+05	6.04E-01	9.948E-02	1.01E+01
	0.35	5.792E-02	9.9057E-02 ± 7.34E-03	2.810E+06 ± 4.79E+05	8.2541E-01 ± 4.68E-03	1.534E+06±5.32E+05	6.34E-01	1.200E-01	8.33
8	**0.15**	8.724E-03	3.862E-02 ± 1.82E-03	3.590E+06 ± 5.72E+05	1.80050E-01 ± 5.93E-04	2.560E+05±1.43E+05	3.49E-02	2.145E-01	4.66
	0.25	4.962E-03	9.306E-02 ± 3.80E-03	2.560E+06 ± 2.89E+05	6.1981E-01 ± 2.05E-03	2.203E+06±3.318E+05	3.51E-02	1.501E-01	6.66
	0.35	1.233E-01	9.9829E-02 ± 4.55E-03	1.165E+06 ± 1.25E+05	6.6157E-01 ± 2.41E-03	2.570E+06±2.32E+05	1.80E-01	1.509E-01	6.67

Table 6.8 – Comparison of the performance of the PWQuad cells with VEGAS for 4 dimensions. VRF stands for the mean reduction factor of the variance.

# peaks	Peak width	Original variance	NIS		VEGAS		# SD	VRF ratio	rel. speedup
			VRF	# Eval	VRF	# Eval			
1	**0.15**	1.703E-06	9.29E-03 ± 2.59E-03	1.540E+06 ± 2.89E+05	5.7519E-06 ± 1.35E-07	4.340E+06±3.45E+05	8.54E-01	2.477E+01	4.05E-02
	0.25	9.325E-05	7.998E-02 ± 6.64E-03	4.700E+06 ± 8.72E+05	9.4945E-06 ± 2.04E-07	2.463E+06±1.92E+05	1.34	8.206E+03	1.22E-04
	0.35	1.118E-03	4.520E-02 ± 2.32E-03	3.614E+07 ± 8.27E+06	4.8832E-05 ± 9.17E-06	1.134E+07±2.94E+06	7.76E-01	9.257E+02	1.08E-03
2	**0.15**	2.563E-06	5.65E-02 ± 1.25E-02	1.790E+06 ± 3.23E+05	1.774E-01 ± 1.78E-02	2.793E+06± 4.36E+05	6.97E-02	3.185E-01	3.14
	0.25	2.932E-04	8.767E-02 ± 4.53E-03	3.560E+06 ± 2.89E+05	7.065E-01 ± 1.66E-02	5.532E+06±3.23E+05	5.71E-01	1.241E-01	8.06
	0.35	2.593E-03	4.325E-01 ± 8.43E-02	1.790E+06 ± 4.33E+05	9.436E-01 ± 1.22E-02	1.745E+06±4.23E+05	7.10E-01	4.583E-01	2.18
4	**0.15**	8.041E-06	1.278E-02 ± 8.06E-03	2.165E+06 ± 5.15E+05	4.0158E-01 ± 1.71E-03	7.789E+06±4.38E+05	1.02E+00	2.358E-02	42.6E+02
	0.25	4.232E-04	2.291E-01 ± 1.48E-02	5.040E+06 ± 5.40E+05	6.7335E-01 ± 7.33E-03	1.246E+07±2.93E+06	9.99E-01	3.403E-01	2.93
	0.35	5.632E-03	3.89E-01 ± 1.15E-01	7.060E+06 ± 7.07E+05	9.1624E-01 ± 6.75E-03	1.233E+06±3.32E+05	2.07E-01	3.516E-01	2.85
8	**0.15**	1.370E-05	1.0729E-01 ± 2.68E-03	6.23E+06 ± 1.82E+06	1.2738E-01 ± 9.36E-03	1.012E+07±4.32E+06	7.78E-01	8.440E-01	1.18
	0.25	1.654E-03	2.699E-01 ± 2.46E-02	5.70E+06 ± 1.12E+06	6.4977E-01 ± 5.13E-03	1.324E+07±1.34E+06	3.67E-02	4.154E-01	2.41
	0.35	5.923E-03	7.9629E-01 ± 3.63E-03	4.560E+06 ± 5.00E+05	8.6094E-01 ± 5.29E-03	2.362E+06±1.34E+05	2.05E-01	9.186E-01	1.09

Table 6.9 – Comparison of the performance of the PWQuad cells with VEGAS for 8 dimensions. VRF stands for the mean reduction factor of the variance.

# peaks	Peak width	Original variance	NIS		VEGAS		# SD	VRF ratio	rel. speedup
			VRF	# Eval	VRF	# Eval			
1	**0.15**	8.901E-16	2.274E-05 ± 1.78E-06	6.78E+06 ± 1.46E+06	9.72823E-04 ± 1.31E-07	1.034E+06±3.34E+05	7.73E-01	2.337E-01	1.37E+01
	0.25	3.380E-09	1.134E-02 ± 3.06E-03	2.364E+06 ± 1.76E+05	1.17022E-07 ± 3.65E-08	1.204E+07±2.23E+06	3.31E-01	9.692E+04	1.03E-5
	0.35	1.495E-06	8.5E-03 ± 2.90E-02	9.60E+06 ± 2.06E+06	1.7355E-06 ± 1.14E-05	1.723E+07±3.34E+06	1.82E+00	4.878E+03	2.05E-04
2	**0.15**	8.947E-16	8.826251E-06 ± 3.74E-09	2.235E+06 ± 5.73E+05	3.5454E-05 ± 7.87E-07	2.230E+06±3.32E+05	4.23E-01	1.12E-01	2.36
	0.25	3.172E-09	3.48E-03 ± 1.77E-03	1.665E+06 ± 4.73E+05	8.451E-04 ± 1.58E-05	1.304E+07±4.23E+06	3.90E-03	4.123E+00	2.24E-01
	0.35	1.495E-06	7.49E-02 ± 3.28E-02	1.540E+06 ± 5.00E+05	1.858E+00 ± 4.12E-01	1.323E+07±3.43E+06	6.54	7.49E-02 *	1.33E+01 *
4	**0.15**	1.421E-14	3.1843E-07 ± 1.78E-08	8.76E+06 ± 2.53E+06	3.236E-05 ± 3.04E-06	2.343E+06±4.23E+05	4.33E-01	4.14E-02	2.41E+01
	0.25	1.996E-08	7.69E-03 ± 2.57E-03	1.005E+06 ± 1.03E+05	2.03E+01 ± 1.99E+01	1.2940E+07±2.43E+06	7.58E-01	7.69E-03 *	1.30E+02*
	0.35	1.634E-06	9.82E-02 ± 9.17E-02	4.060E+06 ± 9.13E+05	2.849E+00 ± 5.30E-01	1.324E+07±5.34E+06	8.87E-01	9.82E-02 *	1.01E+01
8	**0.15**	2.287E-14	1.91E-05 ± 1.11E-05	1.290E+06 ± 1.44E+05	6.24E-02 ± 1.97E-02	2.3071E+06±9.3E+044	5.11E-01	3.066E-04	3.2E+03
	0.25	2.348E-08	4.18E-04 ± 3.98E-04	1.165E+06 ± 1.25E+05	2.377E+00 ± 7.82E-01	1.430E+07±5.33E+06	1.62	4.18E-04 *	2.39E+03 *
	0.35	3.788E-06	1.57E-01 ± 1.73E-01	7.96E+06 ± 1.55E+06	1.261E+00 ± 1.04E-01	1.34E+07±4.23E+08	1.30E+00	1.58E-01 *	6.32

Table 6.10 – Comparison of the performance of the PWQuad cells with VEGAS for 16 dimensions. VRF stands for the mean reduction factor of the variance. For starred values, the ratio was taken between the neural importance sampling variance and the variance without importance sampling, as VEGAS did not converge here

# peaks	Peak width	Original variance	NIS		VEGAS		# SD	VRF ratio	rel speedup
			VRF	# Eval	VRF	# Eval			
1	**0.25**	1.498E-18	3.10E-04 ± 2.63E-04	1.323E+06 ± 2.78E+05	1.4285E-03 ± 7.27E-05	2.032E+06±3.43E+05	9.50E-01	2.170E-01	4.60
	0.35	1.091E-11	6.0787E-07 ± 8.88E-05	2.023E+06 ± 2.38E+05	2.083953E-06± 6.12E-08	5.534E+07±4.23E+06	3.70E-01	2.765E-01	3.60
2	**0.25**	1.561E-22	1.28E-04 ± 2.52E-04	4.173E+06 ± 3.34E+05	1.443E+02 ± 2.74E+01	2.323+06±5.32E+05	1.54	1.29E-04 *	7.74E+03 *
	0.35	1.262E-13	3.82E-03 ± 2.74E-03	7.873E+06 ± 6.74E+05	3.675E+00 ± 1.39E-01	1.530E+07±2.34E+06	7.80E-01	3.82E-03*	2.62E+02*
4	**0.25**	6.222E-23	8.91E-03 ± 3.31E-03	6.133E+06 ± 5.24E+05	5.31E ± 4.38E-01	1.023E+06±7.39E+05	4.68E-01	8.91E-03	1.12E+02
	0.35	3.356E-14	5.39E-05 ± 3.29E-05	4.294E+06 ± 3.94E+05	1.384E-01 ± 8.92E-02	3.183E+07±6.34E+06	1.04E+00	3.895E-04	2.57E+02
8	**0.25**	5.346E-20	5.12E-04 ± 3.07E-04	4.653E+06 ± 5.32E+01	6.58E+00 ± 3.33E+00	1.023E+06±9.34E+05	4.98E-01	5.12E-04 *	1.95E+03 *
	0.35	2.693E-12	1.164E-03 ± 5.27E-04	1.343E+06 ± 2.36E+05	1.55E+02 ± 1.30E+02	1.238E+07±6.23E+06	2.85E-01	1.164E-03 *	8.59E+02 *

Table 6.11 – Comparison of the performance of the PWQuad cells with VEGAS for 32 dimensions. VRF stands for the mean reduction factor of the variance. For starred values, the ratio was taken between the neural importance sampling variance and the variance without importance sampling, as VEGAS did not converge here. Runs with smaller peak widths failed for the neural importance sampling.

References

[1] B. R. Webber, « Monte Carlo simulation of hard hadronic processes », Annual Review of Nuclear and Particle Science **36**, 253 (1986).

[2] A. Buckley, J. Butterworth, S. Gieseke, D. Grellscheid, S. Höche, H. Hoeth, F. Krauss, L. Lönnblad, E. Nurse, P. Richardson and et al., « General-purpose event generators for LHC physics », Physics Reports **504**, 145 (2011).

[3] *The ATLAS collaboration (2020) - Computing and Software - Public Results*, `https://twiki.cern.ch/twiki/bin/view/AtlasPublic/ComputingandSoftwarePublicResults`, Accessed: 2020-03-30.

[4] A. Sirunyan et al., « Search for black holes and other new phenomena in high-multiplicity final states in proton–proton collisions at s=13TeV », Physics Letters B **774**, 279 (2017).

[5] J. A. Evans, Y. Kats, D. Shih and M. J. Strassler, « Toward full LHC coverage of natural supersymmetry », Journal of High Energy Physics **2014** (2014).

[6] S. Höche, S. Prestel and H. Schulz, « Simulation of vector boson plus many jet final states at the high luminosity LHC », Physical Review D **100** (2019).

[7] A. Buckley, *Computational challenges for MC event generation*, 2019, arXiv:`1908.00167 [hep-ph]`.

[8] G. P. Lepage, « A new algorithm for adaptive multidimensional integration », Journal of Computational Physics **27**, 192 (1978).

[9] G. P. Lepage, *VEGAS-An adaptive multi-dimensional integration program*, tech. rep., CLNS-80/447 (1980).

[10] J. H. Friedman and M. H. Wright, « A nested partitioning procedure for numerical multiple integration », ACM Transactions on Mathematical Software (TOMS) **7**, 76 (1981).

[11] W. H. Press and G. R. Farrar, « Recursive stratified sampling for multidimensional Monte Carlo integration », Computers in Physics **4**, 190 (1990).

[12] T. Ohl, « Vegas revisited: Adaptive Monte Carlo integration beyond factorization », Computer Physics Communications **120**, 13 (1999).

[13] S. Jadach, « Foam: Multi-dimensional general purpose Monte Carlo generator with self-adapting simplical grid », Computer Physics Communications **130**, 244 (2000).

[14] T. Hahn, « Cuba—a library for multidimensional numerical integration », Computer Physics Communications **168**, 78 (2005).

[15] K. Kroeninger, S. Schumann and B. Willenberg, *(MC)**3 – a Multi-Channel Markov Chain Monte Carlo algorithm for phase-space sampling*, 2014, arXiv:`1404.4328 [hep-ph]`.

[16] S. Jadach, « Foam: A general-purpose cellular Monte Carlo event generator », Computer Physics Communications **152**, 55 (2003).

[17] W. Kilian, T. Ohl and J. Reuter, « WHIZARD—simulating multi-particle processes at LHC and ILC », The European Physical Journal C **71** (2011).

[18] 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** (2014).

[19] D. Bourilkov, « Machine and deep learning applications in particle physics », International Journal of Modern Physics A **34**, 1930019 (2019).

[20] J. Bendavid, *Efficient Monte Carlo Integration Using Boosted Decision Trees and Generative Deep Neural Networks*, 2017, arXiv:`1707.00028 [hep-ph]`.

[21] M. D. Klimek and M. Perelstein, *Neural Network-Based Approach to Phase Space Integration*, 2018, arXiv:`1810.11509 [hep-ph]`.

[22] D. J. Rezende and S. Mohamed, « Variational inference with normalizing flows », arXiv preprint arXiv:1505.05770 (2015).

[23] G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed and B. Lakshminarayanan, *Normalizing Flows for Probabilistic Modeling and Inference*, 2019, arXiv:`1912.02762 [stat.ML]`.

[24] L. Dinh, D. Krueger and Y. Bengio, *NICE: Non-linear Independent Components Estimation*, 2014, arXiv:`1410.8516 [cs.LG]`.

[25] L. Dinh, J. Sohl-Dickstein and S. Bengio, *Density estimation using Real NVP*, 2016, arXiv:`1605.08803 [cs.LG]`.

[26] T. Müller, B. McWilliams, F. Rousselle, M. Gross and J. Novák, *Neural Importance Sampling*, 2018, arXiv:`1808.03856 [cs.LG]`.

[27] C. Durkan, A. Bekasov, I. Murray and G. Papamakarios, *Neural Spline Flows*, 2019, arXiv:`1906.04032 [stat.ML]`.

[28] C. Gao, S. Hoeche, J. Isaacson, C. Krause and H. Schulz, *Event Generation with Normalizing Flows*, 2020, arXiv:`2001.10028 [hep-ph]`.

[29] E. Bothmann, T. Janßen, M. Knobbe, T. Schmale and S. Schumann, *Exploring phase space with Neural Importance Sampling*, 2020, arXiv:`2001.05478 [hep-ph]`.

[30] C. Gao, J. Isaacson and C. Krause, *i-flow: High-dimensional Integration and Sampling with Normalizing Flows*, 2020, arXiv:`2001.05486 [physics.comp-ph]`.

[31] V. Del Duca, C. Duhr, R. Haindl, A. Lazopoulos and M. Michel, « Tree-level splitting amplitudes for a quark into four collinear partons », JHEP **02**, 189 (2020), arXiv:`1912.06425 [hep-ph]`.

[32] R. Kleiss and R. Pittau, « Weight optimization in multichannel Monte Carlo », Computer Physics Communications **83**, 141 (1994).

[33] S. Carrazza and J. M. Cruz-Martinez, *VegasFlow: accelerating Monte Carlo simulation across multiple hardware platforms*, 2020, arXiv:`2002.12921 [physics.comp-ph]`.

[34] S. Otten, S. Caron, W. de Swart, M. van Beekveld, L. Hendriks, C. van Leeuwen, D. Podareanu, R. R. de Austri and R. Verheyen, *Event Generation and Statistical Sampling for Physics with Deep Generative Models and a Density Information Buffer*, 2019, arXiv:`1901.00875 [hep-ph]`.

[35] R. Di Sipio, M. F. Giannelli, S. K. Haghighat and S. Palazzo, « DijetGAN: a Generative-Adversarial Network approach for the simulation of QCD dijet events at the LHC », Journal of High Energy Physics **2019** (2019).

[36] A. Butter, T. Plehn and R. Winterhalder, « How to GAN LHC events », SciPost Physics **7** (2019).

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

[38] A. Pinkus, « Approximation theory of the MLP model in neural networks », Acta Numerica **8**, 143 (1999).

[39] P. Kidger and T. Lyons, *Universal Approximation with Deep Narrow Networks*, 2019, arXiv:`1905.08539 [cs.LG]`.

[40] L. Bottou, « Stochastic Learning », in *Advanced Lectures on Machine Learning*, edited by O. Bousquet and U. von Luxburg, Lecture Notes in Artificial Intelligence, LNAI 3176 (Springer Verlag, Berlin, 2004), pp. 146–168.

[41] *PyTorch Documentation*, (2020) `https://pytorch.org/docs/stable/torch.html` (visited on 25/05/2020).

[42] S. Ioffe and C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015, arXiv:`1502.03167 [cs.LG]`.

[43] D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization*, 2014, arXiv:`1412.6980 [cs.LG]`.

[44] Y. Yao, L. Rosasco and A. Caponnetto, « On early stopping in gradient descent learning », Constr. Approx, 289 (2007).

[45] *Documentation of the* VEGAS *pacakge*, (2020) `https://vegas.readthedocs.io/en/latest/index.html` (visited on 25/05/2020).

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

[47] S. Plätzer, *RAMBO on diet*, 2013, arXiv:`1308.2922 [hep-ph]`.

[48] R. Kleiss, W. Stirling and S. Ellis, « A new Monte Carlo treatment of multiparticle phase space at high energies », Computer Physics Communications **40**, 359 (1986).

[49] F. James, « Monte Carlo theory and practice », Reports on progress in Physics **43**, 1145 (1980).

[50] R. D. Field, *Applications of Perturbative QCD* (Addison-Wesley, Redwood City, 1989).

[51] 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** (2015).

[52] R. D. Ball, V. Bertone, S. Carrazza, C. S. Deans, L. Del Debbio, S. Forte, A. Guffanti, N. P. Hartland, J. I. Latorre, J. Rojo and et al., « Parton distributions with LHC data », Nuclear Physics B **867**, 244 (2013).

List of Figures

1.1 The VEGAS grid for two aligned gaussian peaks. 8

1.2 The VEGAS grid for two non-aligned gaussian peaks. 9

2.1 A 2-layer fully connected NN (one hidden layer of 4 neurons and one output layer with 2 neurons), and three inputs. The white circles are the artificial neurons. . . . 11

2.2 A sketch of the structure of an artificial neuron with three weighted inputs and an activation function σ. 11

2.3 Schematic structure of a coupling layer. 15

2.4 A demonstration of the mapping for piecewise-linear cells in 1D. The orange dotted line is the function to be learned, the blue line is the network prediction/the coupling transform. 18

2.5 A demonstration of the mapping for piecewise-quadratic cells in 1D. The orange dotted line is the function to be learned, the blue line is the network prediction/the coupling transform. 19

3.1 A sketch of the structure of the code for the case of 2 coupling cells. Black are classes with their most important properties, blue are methods and the red arrows indicate their relations. 22

3.2 The structure of the rectangular neural network used for the coupling cells. Each block refers to one of the repeating units. The number of these units is meant with the length of the NN. The width refers to the shape of the linear transformation. The shape of the last linear transformation is set by the number of parameters required by the coupling cell. The red line symbolises the flow of calculation. 23

3.3 Stable (left) and instable (right) behaviour of the loss during the training 24

3.4 The untrained mapping of the affine (left) and piecewise-linear (right) coupling cell, for a single cell in 2 dimensions. The peaked behaviour of the untrained affine coupling cells reduces the ability to correctly train features. 25

3.5 The mapping of the affine coupling cell after training on a gaussian. A wrong local minimum of the loss was found, referring to a vanishing Jacobian. 25
3.6 Plot of the "camel" function 26
3.7 Plot of the slashed circle function 27
3.8 The output of the piecewise-linear coupling cell after hyperparameter optimisation in the best (left) and second best (right) configuration 28
3.9 The output of the piecewise-linear coupling cell for the slashed circle 29
3.10 The output of the piecewise-quadratic coupling cell in the optimal configuration (left) and the second best (right) 29
3.11 The output of the piecewise-quadratic coupling cell for the slashed circle 30
3.12 Overview of the minimal relative variance achieved of the different setups of the piecewise-quadratic coupling cells versus the hyperparameters of the setup. From left to right: learning rate, number of bins, width of the NN, length of the NN. . . . 30
3.13 Overview of the minimal relative variance achieved in relation to the batch size and the learning rate. 31
3.14 Time evolution of the mapping for the gaussian double peak for piecewise-linear (top) and piecewise-quadratic (bottom) coupling cells. 32
3.15 Time evolution of the mapping for the slashed circle for piecewise-linear (top) and piecewise-quadratic (bottom) coupling cells. 32
3.16 Bins of the first (left) and second (right) coupling cell of the optimal configuration. Top: untrained, bottom: trained model. The axis parallel to the lines refers to the non-transformed dimension (in the right image, this dimension was transformed by the first coupling cell already), the position of the lines to the right bin boundary. The colour encodes the value of the cumulative distribution function at the right bin boundary, e.g. the value to which the input dimension will be transformed at the bin boundary. 33
3.17 Left: the colour-encoded 2D plane. Each point got encoded in the RGB-scheme as $(x_0, x_1, 0.5)$. Middle: the same set of points after transformation with a model trained on the double gaussian, but with the colour-encoding of their original position. Right: the same, but mapped with a model trained on the slashed circle. 34
3.18 The adaptive grid of VEGAS at the end of the training for the gaussian peak. . . . 35
3.19 The adaptive grid of the VEGAS algorithm at the end of the training for the gaussian double peak. Note that instead of two peaks, four were learned. 36
3.20 The output of the VEGAS grid for the slashed circle 36

4.1 The absolute speedup of neural importance sampling in dependency of the number of peaks and dimensions. The colour encodes the logarithm of the speedup. Left: peak width 0.15, in the middle 0.25 and to the right 0.35. White fields represent non-existing data points. 41

4.2 The relative speedup between neural importance sampling and VEGAS in dependency of the number of peaks and dimensions. The colour encodes the logarithm of the relative speedup. Left: peak width 0.15, in the middle 0.25 and to the right 0.35. White fields represent non-existing data points. 41
4.3 The trained model for the 4D double gaussian peak in multiple projections. 42
4.4 The trained model for the 4D mixed gaussian peaks in multiple projections. 43
4.5 Distribution of the hyperparameters for the best runs in 8 dimensions of the gaussian double peak with peak width 0.25. 43
4.6 Distribution of the hyperparameters for the best runs in 8 dimensions of the gaussian double peak with peak width 0.25. Top: data for the quarter which converged after the smallest number of epochs. Lower: data for the quarter which converged after the biggest number of epochs. 45
4.7 Distribution of the hyperparameters for the best runs in 8 dimensions of the gaussian double peak with peak width 0.25. Top: data for the quarter which converged after the shortest time. Lower: data for the quarter which converged after the longest time. 46

5.1 Comparison of the runtime of different implementations for the generation for 4-particle massive final states. The batches were generated sequentially. 53

6.1 Histogram of the performance of the different setups during hyperparameter optimisation for $e^+e^- \to \mu^+\mu^-$. The blue line is the variance reduction achieved by VEGAS with its standard deviation. 59
6.2 Histogram of the performance of the different setups during hyperparameter optimisation for $gu \to gu$. The blue line is the variance reduction achieved by VEGAS with its standard deviation. The yellow bin refers to setups which could achieve no variance better than 20% of the original value, the red bin refers to setups which achieved no improvement. 60
6.3 Histogram of the performance of the different setups during hyperparameter optimisation for $uc \to ucg$. The blue line is the variance reduction achieved by VEGAS with its standard deviation. The yellow bin refers to setups which could achieve no variance better than 20% of the original value, the red bin refers to setups which achieved no improvement. 61
6.4 The $(x_0 - x_2)$ plane, referring to θ and the Bjorken x of the d, of the integrand (left) and the trained model (right). 63
6.5 The plane of the Bjorken x of the integrand (left) and the trained model (right). . . 63
6.6 Both planes in the VEGAS grid. 64

List of Tables

2.1 Determination of the transformed dimensions for each coupling cell 17

3.1 Performance of the piecewise-linear coupling cell setups. 27
3.2 Performance of the piecewise-quadratic coupling cell setups. 28
3.3 Comparison of the performance of the piecewise-linear and piecewise quadratic coupling cell. 31
3.4 Performance for the two integration strategies for the single gaussian peak. 35
3.5 Performance for the two integration strategies for the double gaussian peak. 36

5.1 Comparison of the maximal and average difference of 40000 runs between initial and final state energies in the phase space generator for the generation of 4-particle massive final states. 53
5.2 Unweighting efficiency of 40000 runs with massless massive final and initial states at $E_{cm} = 1000\text{GeV}^2$. 53
5.3 Comparison of the cross section calculated from the MadGraph matrix element, either with the integrated integration tool or using the developed PyTorch PS generator and importance sampling with VEGAS. $E_{cm} = 1000\text{GeV}^2, p_{T,min} = 10\text{GeV}, \eta_{max} = 2.4, \Delta_{R,min} = 0.4$. 55
5.4 Comparison of the cross section calculated from the MadGraph matrix element, either with the integrated integration tool or using the developed PyTorch PS generator and importance sampling with VEGAS $E_{cm} = 1000\text{GeV}^2, p_{T,min} = 10\text{GeV}, \eta_{max} = 2.4, \Delta_{R,min} = 0.4$, PDF set NNPDF23_nlo_as_0119. 56

6.1 Comparison of the performance of the best NIS setup and VEGAS for LO $e^+e^- \to \mu^+\mu^-$. VRF stands for the mean reduction factor of the variance. 59
6.2 Comparison of the performance of the best NIS setup and VEGAS for QCD LO $gu \to gu$. VRF stands for the mean reduction factor of the variance. 60
6.3 Comparison of the performance of the NIS and VEGAS for QCD LO $uc \to ucg$. VRF stands for the mean reduction factor of the variance. 61

6.4 Comparison of the performance of the NIS and VEGAS for $d\bar{d} \to d\bar{d}$ via Z. VRF stands for the mean reduction factor of the variance. 62
6.5 Overview of the result for $d\bar{d} \to d\bar{d}$ via Z for different integrators. 62
6.6 Overview of the result for $d\bar{d} \to d\bar{d}$ via Z for VEGAS at different values for the number of function evaluations. 62
6.7 Comparison of the performance of the PWQuad cells with VEGAS for 2 dimensions. VRF stands for the mean reduction factor of the variance. 74
6.8 Comparison of the performance of the PWQuad cells with VEGAS for 4 dimensions. VRF stands for the mean reduction factor of the variance. 74
6.9 Comparison of the performance of the PWQuad cells with VEGAS for 8 dimensions. VRF stands for the mean reduction factor of the variance. 74
6.10 Comparison of the performance of the PWQuad cells with VEGAS for 16 dimensions. VRF stands for the mean reduction factor of the variance. For starred values, the ratio was taken between the neural importance sampling variance and the variance without importance sampling, as VEGAS did not converge here 75
6.11 Comparison of the performance of the PWQuad cells with VEGAS for 32 dimensions. VRF stands for the mean reduction factor of the variance. For starred values, the ratio was taken between the neural importance sampling variance and the variance without importance sampling, as VEGAS did not converge here. Runs with smaller peak widths failed for the neural importance sampling. 75

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Machine learning techniques applied to Monte Carlo integration

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	Vorname(n):
Götz	Niklas Matthias

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge" beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	Unterschrift(en)
Zürich, 07.09.2020	Götz

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.