University *of Ljubljana*
Faculty *of Mathematics and Physics*

Master program in Physics
Seminar 2

Use of Normalizing Flows in Particle Physics Simulations

Author: Jan Gavranovič
Advisor: prof. dr. Borut Paul Kerševan

Ljubljana, April 2022

Abstract

Normalizing flows are a family of machine learning methods for constructing learnable probability distributions using neural networks. The seminar presents normalizing flows in the context of fast event generation in High Energy Physics. First, the problem of large scale Monte Carlo simulations at the LHC is presented. After that, the basics of flow models and their mathematical design is discussed. What follows is a presentation of a particular flow design, called realNVP. At the end, a study where event generation is tested on a theoretical Higgs boson production simulated dataset is presented.

Contents

1 Introduction **1**
- 1.1 Computing at the LHC . . . 1
- 1.2 Simulated Higgs boson production . . . 2

2 Normalizing flows **3**
- 2.1 Basics . . . 4
- 2.2 Learning objective . . . 5

3 Affine coupling layers **5**
- 3.1 Affine transformer . . . 6
- 3.2 Coupling layer . . . 6
- 3.3 Permutation layer . . . 8

4 Results **8**
- 4.1 Data preprocessing . . . 8
- 4.2 ML event generation . . . 9

5 Discussion **10**

A Invertible 1×1 convolution **12**

B Batch normalization **12**

C Model parameters **13**

D Toy datasets **13**

1 Introduction

Simulating high energy particle collisions (events) requires the use of Monte Carlo (MC) generators. The basic idea of a MC generator is as follows: first, a random number is sampled and, then in the second step, an algorithm transforms this random number into a simulated physics event using various theoretical models. The results are used for characterizing a given signal hypothesis and to study potential background processes. A fundamental problem with these kind of methods is the need for large computational resources, which restrict the discovery process due to speed and high cost of CPU and disk space needed to generate and store MC events.

1.1 Computing at the LHC

Let us focus on the ATLAS detector, which is one of the general purpose detectors at the Large Hadron Collider (LHC) at CERN that began operation in 2008 [1]. The detector collects data from proton-proton collisions where protons come in bunches of up to 10^{11} and collide 40 million times per second with center of mass energies up to 14 TeV.

The full procedure of MC event generation at the LHC's detectors may take up to 10 minutes per event and produce $\sim$ 1 MB of data of which only $\sim$ 1 kB of high level features is used in the final analysis as reported in Refs. [2, 3].

The computing power for producing simulated events is provided by a worldwide supercomputer architecture called the *Worldwide LHC Computing Grid (WLCG)* [4]. It combines the computing resources of about 900 000 computer cores from over 170 sites in 42 countries, producing a massive distributed computing infrastructure that provides access to the LHC data

and the power to process it. It runs over 2 million tasks per day and, at the end of the LHC's Run 2, global transfer rates regularly exceeded 60 GB/s.

With the upcoming Run 3 and in the future the High-Luminosity LHC (HL-LHC) it is expected that the experiments will require even more computing power for MC event generation to match the larger collected datasets that will come with the increase in luminosity. Taken together, the data and MC requirements of the HL-LHC physics programme are formidable, and if the computing costs are to be kept within feasible levels, very significant improvements will need to be made both in terms of computing and storage as described in Ref. [5]. The majority of resources will be needed to produce simulated events, from physics modeling (event generation) to detector simulation, reconstruction and production of analysis formats (Figure 1).

One promising solution to the above-mentioned problems is the use of Machine Learning (ML), in particular deep generative modeling. The idea is to create large amounts of events at limited computing cost using a learning algorithm that was trained on a small set of MC simulated events. The algorithm will learn to model the D-dimensional distributions with density $p(\mathbf{x})$ of observables $\mathbf{x}$ that can be used in the final stage of a given physics analysis.

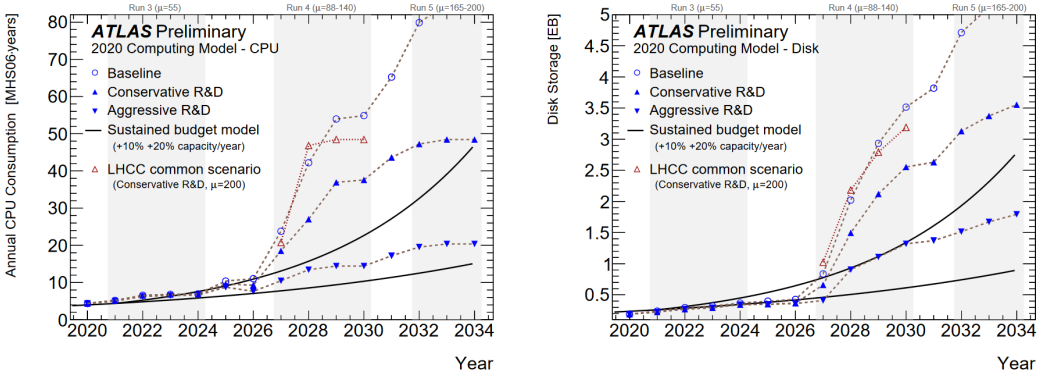
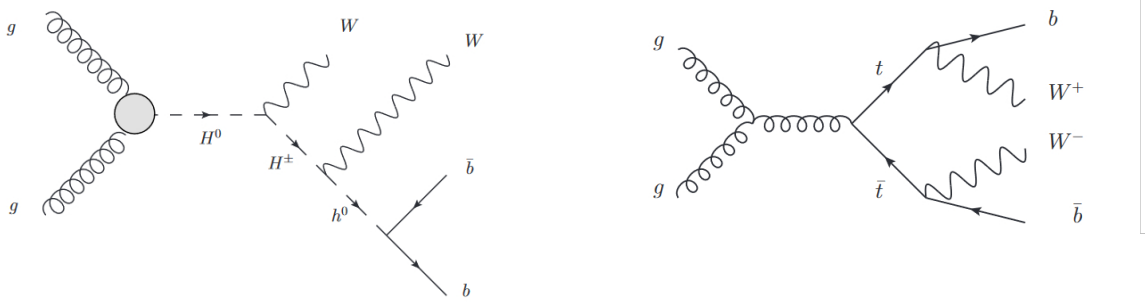


Figure 1: Estimated CPU and disk resources needed for the years 2020 to 2034 under the different scenarios (taken from Ref. [5]). Expected average number of inelastic proton-proton collisions per bunch crossing is denoted by μ. MHS06·years stands for 10^6 HEPSPEC06 per year, a standard CPU performance metric for High Energy Physics [6].

1.2 Simulated Higgs boson production

A simulated dataset of a theoretical Beyond-Standard-Model Higgs boson production (Figure 2a) and a background process with the identical decay products in the final state but distinct kinematic features (Figure 2b) will be used to illustrate the performance of machine learning sampling in high dimensional feature spaces.



(a) Signal process H^0 and $H^{\pm}$.

(b) Background top quark process.

Figure 2: Diagrams for simulated Higgs dataset (taken from Ref. [7]).

The simulated dataset is from Ref. [7] and was originally used for training deep neural network classifiers for distinguishing signal from background (i.e. searching for new particles). The signal process is the fusion of two gluons gg into a heavy neutral Higgs boson H^0 that decays into heavy charged Higgs bosons $H^\pm$ and a W boson. The $H^\pm$ then decays to a second W boson and to a light Higgs boson h^0 that decays to b quarks. The whole signal process can be described as:

$$gg \to H^0 \to W^\mp H^\pm \to W^\mp W^\pm h^0 \to W^\mp W^\pm b\bar{b}\,. \tag{1}$$

The background process, without the Higgs boson intermediate state and the same final state $W^\mp W^\pm b\bar{b}$, is the production of a pair of top quarks. Events were generated assuming 8 TeV collisions of protons at the LHC with masses set to $m_{H^0} = 425$ GeV and $m_{H^\pm} = 325$ GeV.

Produced particles in the above processes cannot be directly observed because they decay almost immediately into other particles, called decay products. Observable decay products include electrically charged leptons ℓ (electrons or muons) and particle jets j (collimated streams of particles originating from quarks or gluons). The dataset contains semi-leptonic decay modes, where the first W boson decays into $\ell\nu$ and the second into jj giving us decay producst $\ell\nu b$ and jjb. Further restrictions are also set on transverse momentum p_T, pseudorapidity η, type and number of leptons and number of jets [1]. Events that satisfy the above requirements are characterized by a set of features that describe the measurements made by the detector. Low-level features are:

- jet p_T, η and azimuthal angle ϕ,
- jet b-tag (indicating that the jets are likely produced by b-quarks),
- lepton p_T, η and ϕ,
- missing energy carried away by the invisible particles (e.g. neutrinos),

which gives us 21 features (counting two jets per b quark and missing energy magnitude and angle ϕ). Further features can be obtained by reconstructing invariant masses of the different intermediate states of the two processes. These are called high-level features and are:

$$m_{jj},\ m_{jjj},\ m_{lv},\ m_{jlv},\ m_{b\bar{b}},\ m_{Wb\bar{b}},\ m_{WWb\bar{b}}\,.$$

Ignoring angles ϕ due to symmetry (uniform distribution), and focusing only on continuous features, gives us an 18-dimensional feature space that the machine learning model will attempt to learn.

This seminar will investigate the feasibility of using normalizing flows (Refs. [8, 9]) for generating distributions of observables that can achieve an adequate agreement with the true MC distributions.

2 Normalizing flows

One of the goals of statistical analysis using machine learning is modeling probability distributions using samples coming from such data-generating distributions. This is often referred to as generative modeling and is a part of unsupervised learning. The two most influential and well known generative algorithms are Generative Adversarial Networks (GANs) and Variational Auto Encoders (VAEs) introduced in Refs. [10] and [11]. Both of these methods have produced impressive results in tasks such as image generation (sampling). The main practical problem

[1] ATLAS detector uses a right-handed coordinate system with its origin at the nominal interaction point (IP) in the centre of the detector and the z-axis along the beam pipe. The x-axis points from the IP to the centre of the LHC ring, and the y-axis points upwards. Polar coordinates (r, ϕ) are used in the transverse plane, ϕ being the azimuthal angle around the z-axis. The pseudorapidity is defined in terms of the polar angle θ as $\eta = -\ln\tan(\theta/2)$.

of these two methods is that they are notoriously difficult to train. Another downside is that density estimation cannot be done exactly because neither of them allow for exact evaluation of the probability density at new data points. Normalizing flows are a competing approach that aim to provide an alternative framework for such generative modeling.

2.1 Basics

Let $\mathbf{u} \in \mathbb{R}^D$ be a random vector with a known probability density function $p_{\mathbf{u}}(\mathbf{u}) : \mathbb{R}^D \to \mathbb{R}$. Distribution $p_{\mathrm{u}}(\mathbf{u})$ is called a base distribution and is usually chosen to be something simple such as a normal distribution $\mathcal{N}(0, \mathbf{I})$. Given our data $\mathbf{x} \in \mathbb{R}^D$, we would like to know the distribution it was drawn from. The idea is to express $\mathbf{x}$ as a transformation T of a random variable $\mathbf{u}$ in such a way that

$$\mathbf{x} = T(\mathbf{u}) \,, \quad \mathbf{u} \sim p_{\mathrm{u}}(\mathbf{u}) \,, \tag{2}$$

where T is implemented using a neural network (see Refs. [12, 13]). The transformation T must be a *diffeomorphism*, meaning that it is invertible and both T and T^{-1} are differentiable. Under these conditions the density $p_{\mathrm{x}}(\mathbf{x})$ is well defined and can be calculated using the change of variables formula (see Ref. [14] for derivation)

$$p_{\mathrm{x}}(\mathbf{x}) = p_{\mathrm{u}}\left(T^{-1}\left(\mathbf{x}\right)\right) \left|\mathrm{det} J_T\left(T^{-1}\left(\mathbf{x}\right)\right)\right|^{-1} = p_{\mathrm{u}}\left(T^{-1}\left(\mathbf{x}\right)\right) \left|\mathrm{det} J_{T^{-1}}\left(\mathbf{x}\right)\right| \,, \tag{3}$$

where $J_T(\mathbf{u})$ is a $D \times D$ Jacobian matrix of partial derivatives:

$$J_T(\mathbf{u}) = \begin{bmatrix} \frac{\partial T_1}{\partial \mathrm{u}_1} & \cdots & \frac{\partial T_1}{\partial \mathrm{u}_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial T_D}{\partial \mathrm{u}_1} & \cdots & \frac{\partial T_D}{\partial \mathrm{u}_D} \end{bmatrix} \,. \tag{4}$$

Invertible and differentiable transformations are composible, which allows us to construct a flow by chaining together different transformations. This means that we can construct a complicated transformation T with more expressive power by composing together many simpler transformations:

$$T = T_K \circ \ldots \circ T_1 \quad \text{and} \quad T^{-1} = T_1^{-1} \circ \ldots \circ T_K^{-1} \,. \tag{5}$$

A flow is thus referring to the trajectory of samples from the base distribution as they get gradually transformed by each transformation to the target distribution. This is known as forward or generating direction (Figure 3). The word normalizing refers to the inverse taking samples from data and transforming them to the base distribution which is usually normal. This direction is called inverse or normalizing direction and is the direction we train the model in. Flows in forward and inverse direction are then, respectively,

$$\begin{aligned} \mathbf{z}_k &= T_k(\mathbf{z}_{k-1}) \quad \text{for} \quad k = 1, \ldots, K \,, \\ \mathbf{z}_{k-1} &= T_k^{-1}(\mathbf{z}_k) \quad \text{for} \quad k = K, \ldots, 1 \,, \end{aligned} \tag{6}$$

where $\mathbf{z}_0 = \mathbf{u}$ and $\mathbf{z}_K = \mathbf{x}$. The log-determinant of a flow is given by

$$\log \left|\det J_T\left(\mathbf{z}_0\right)\right| = \log \left| \prod_{k=1}^{K} \det J_{T_k}\left(\mathbf{z}_{k-1}\right) \right| = \sum_{k=1}^{K} \log \left|\det J_{T_k}\left(\mathbf{z}_{k-1}\right)\right| \,. \tag{7}$$

A trained flow model provides sampling by Eq. (2) and density estimation by Eq. (3).

2.2 Learning objective

Fitting a parametric flow model $p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})$ to a target distribution $p^*_{\mathrm{x}}(\mathbf{x})$ is done using maximum likelihood estimation that gives us the average log-likelihood over N data points

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N} \log p_{\mathrm{x}}(\mathbf{x}_n;\boldsymbol{\theta})\,, \tag{8}$$

which is also known as the loss function and is the quantity we optimize using gradient based methods during training. This can be done because the exact log-likelihood of input data is tractable in flow based models. In order to keep the computing load at a manageable level averaging is performed over batches of data and not on the whole dataset. Batches are typically of size $\mathcal{O}(10^2)$ and are randomly sampled from the whole dataset during training.

Using Eq. (3) we get

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N} \left[\log p_{\mathrm{u}}\left(T^{-1}(\mathbf{x}_n;\boldsymbol{\phi});\boldsymbol{\psi}\right) + \log\left|\det J_{T^{-1}}(\mathbf{x}_n;\boldsymbol{\phi})\right|\right] \tag{9}$$

where $\boldsymbol{\theta} = \{\boldsymbol{\phi},\boldsymbol{\psi}\}$ are parameters of transformation T and the base distribution, respectively. The parameters $\boldsymbol{\psi}$ of the base distribution are usually fixed, for example the zero mean and the unit variance of a normal distribution. From Eq. (9) we see that in order to fit a flow model we need to compute inverse transformation T^{-1}, Jacobian determinant, density $p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi})$ and be able to differentiate through all three. For sampling the flow model we must also know T and be able to sample from $p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi})$.

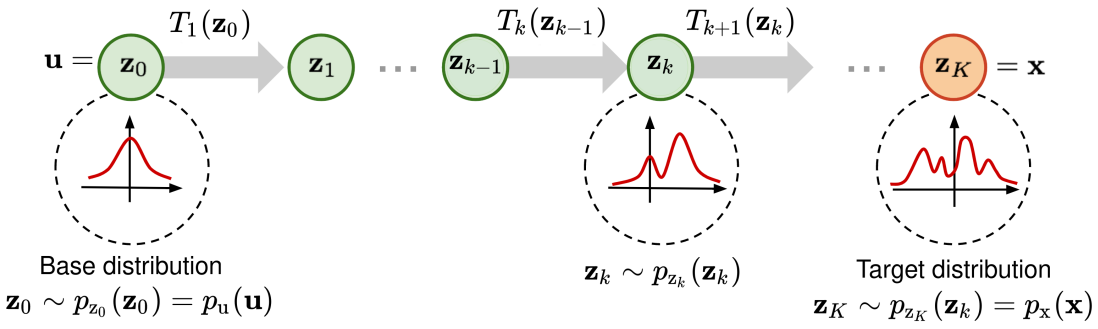


Figure 3: Flow in the generating direction (adapted from [15]).

3 Affine coupling layers

When designing a machine learning flow model $f_{\boldsymbol{\phi}}$ implementing the transformations T we must ensure that the model

1. is invertible and differentiable,
2. has an efficiently computable inverse and Jacobian determinant with both having at most linear $\mathcal{O}(D)$ time complexity.

A general flow implementation has the following form:

$$\mathrm{z}'_i = \tau(\mathrm{z}_i;\boldsymbol{h}_i)\,, \quad \mathrm{z}_i = \tau^{-1}(\mathrm{z}'_i;\boldsymbol{h}_i)\,, \quad \boldsymbol{h}_i = c_i(\mathbf{z}_{<i})\,, \tag{10}$$

where τ is the transformer, c_i is the i-th conditioner, $\mathbf{z}$ are inputs and $\mathbf{z}'$ are outputs of the model. The transformer is an invertible function and specifies how the flow acts on $\mathbf{z}$ to output $\mathbf{z}'$. The

conditioner determines the parameters of the transformer and thus modifies it. The conditioner is constrained in such a way that the i-th conditioner can only take in values with indices less then i denoted as $\mathbf{z}_{<i}$ in the above equation. This is known as the autoregressive constraint.

The Jacobian of transformation (10) is triangular. This is due to the fact that z'_i does not depend on $\mathbf{z}_{>i}$ which makes partial derivatives of z'_i with respect to $\mathrm{z}_{j>i}$ zero. The Jacobian of $f_{\boldsymbol{\phi}}$ is thus

$$J_{f_{\boldsymbol{\phi}}}(\mathbf{z}) = \begin{bmatrix} \frac{\partial \tau}{\partial \mathrm{z}_1}(\mathrm{z}_1; \boldsymbol{h}_1) & & \mathbf{0} \\ & \ddots & \\ \mathbf{L}(\mathbf{z}) & & \frac{\partial \tau}{\partial \mathrm{z}_D}(\mathrm{z}_D; \boldsymbol{h}_D) \end{bmatrix}, \tag{11}$$

where $\mathbf{L}(\mathbf{z})$ is a lower triangular matrix that we do not need to calculate. This means that the log-determinant can be computed in linear time since the determinant of a lower or an upper triangular matrix is a simple multiplication of diagonal terms:

$$\log \left|\det J_{f_\phi}(\mathbf{z})\right| = \log \left|\prod_{i=1}^{D} \frac{\partial \tau}{\partial \mathrm{z}_i}(\mathrm{z}_i; \boldsymbol{h}_i)\right| = \sum_{i=1}^{D} \log \left|\frac{\partial \tau}{\partial \mathrm{z}_i}(\mathrm{z}_i; \boldsymbol{h}_i)\right| . \tag{12}$$

3.1 Affine transformer

One of the choices for the transformer is an affine transformation

$$\tau(\mathrm{z}_i; \boldsymbol{h}_i) = \alpha_i \mathrm{z}_i + \beta_i , \quad \boldsymbol{h}_i = \{\alpha_i, \beta_i\} , \tag{13}$$

which simply scales and shifts the input whereby α and β are in fact neural network outputs for specific $\mathbf{z}_{<i}$. Invertibility can be enforced by taking $\alpha_i = \exp \tilde{\alpha}_i$ and writing $\boldsymbol{h}_i = \{\tilde{\alpha}_i, \beta_i\}$, where α_i is unconstrained. The Jacobian determinant (12) is then

$$\log \left|\det J_{f_{\boldsymbol{\phi}}}(\mathbf{z})\right| = \sum_{i=1}^{D} \log |\alpha_i| = \sum_{i=1}^{D} \tilde{\alpha}_i . \tag{14}$$

One major downside of this simple transformation is that it has limited expressiveness, which means that in practice, stacking multiple affine layers is needed.

3.2 Coupling layer

The conditioner $c_i(\mathbf{z}_{<i})$ can be any function of $\mathbf{z}_{<i}$. This means that a conditioner can be implemented using any model with input $\mathbf{z}_{<i}$ and output $\boldsymbol{h}$. We shall implement it using coupling layers as in *real-valued non-volume preserving (realNVP)* model [16].

A coupling layer splits $\mathbf{z}$ into two parts at index d and transforms the second part as a function of the first part (Figure 4). Each of these two parts is referred to as an affine coupling layer

$$\begin{aligned} \mathbf{z}'_{\leq d} &= \mathbf{z}_{\leq d} , \\ \boldsymbol{h}_{>d} &= F(\mathbf{z}_{\leq d}) , \\ \mathbf{z}'_{>d} &= \tau(\mathbf{z}_{>d}; \boldsymbol{h}_{>d}) , \end{aligned} \tag{15}$$

where F is an arbitrary function (a neural network). In the case of realNVP the implementation is a follows:

$$\begin{aligned} \mathbf{z}'_{\leq d} &= \mathbf{z}_{\leq d} , \\ \boldsymbol{h}_{>d} &= \{s(\mathbf{z}_{\leq d}),\ t(\mathbf{z}_{\leq d})\} , \\ \mathbf{z}'_{>d} &= \mathbf{z}_{>d} \odot \exp\left(s(\mathbf{z}_{\leq d})\right) + t(\mathbf{z}_{\leq d}) , \end{aligned} \tag{16}$$

where the affine transformation τ, consisting of shifting $\alpha = s$ and translation $\beta = t$ operations, is implemented by two separate neural networks. In the above, $\odot$ denotes elementwise multiplication. The inverse is straightforward

$$\begin{aligned} \mathbf{z}_{\leq d} &= \mathbf{z}'_{\leq d} \,, \\ \mathbf{z}_{>d} &= \left(\mathbf{z}'_{>d} - t(\mathbf{z}'_{\leq d})\right) \odot \exp\left(-s(\mathbf{z}'_{\leq d})\right) \,, \end{aligned} \tag{17}$$

and does not require computing inverses of s and t, thus allowing them to become arbitrarily complex and difficult to invert. The Jacobian is lower triangular with block like structure

$$J_{f_\phi} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{A} & \mathbf{D} \end{bmatrix} \,, \tag{18}$$

where $\mathbf{I}$ is $d \times d$ identity matrix, $\mathbf{D}$ is $(D-d) \times (D-d)$ diagonal matrix and $\mathbf{A}$ is $(D-d) \times d$ matrix of partial derivatives. The only relevant part is

$$\mathbf{D} = \mathrm{diag}\left[\exp(s(\mathbf{z}_{\leq d}))\right] \,, \tag{19}$$

which makes the determinant easy to calculate:

$$\log\left|\det J_{f_\phi}(\mathbf{z})\right| = \sum_j s(\mathbf{z}_{\leq d})_j \,. \tag{20}$$

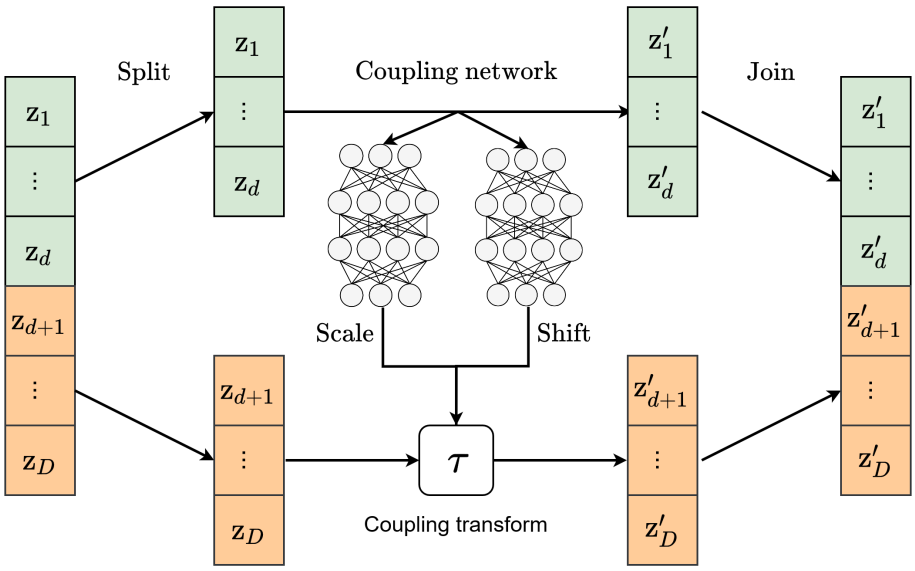


Figure 4: Illustration of forward direction of an affine coupling layer.

Splitting and joining can be implemented using binary masks $\mathbf{b}$. Using $\mathbf{b}$ we can rewrite Eq. (16) and (17) more compactly as:

$$\mathbf{z}' = \mathbf{b} \odot \mathbf{z} + (1 - \mathbf{b}) \odot (\mathbf{z} \odot \exp(s(\mathbf{b} \odot \mathbf{z})) + t(\mathbf{b} \odot \mathbf{z})) \,, \tag{21}$$

$$\mathbf{z} = \mathbf{b} \odot \mathbf{z}' + (1 - \mathbf{b}) \odot (\mathbf{z}' - t(\mathbf{b} \odot \mathbf{z}')) \odot \exp(-s(\mathbf{b} \odot \mathbf{z}')) \,, \tag{22}$$

where partitioning is handled implicitly by binary mask construction. An example of $\mathbf{b}$ would be a checkerboard mask

$$b_d = d \bmod 2 \quad \text{where} \quad d = 0, \ldots, D-1 \,. \tag{23}$$

3.3 Permutation layer

From Figure 4 it is clear that the part $\mathbf{z}_{<d}$ remains unchanged when stacking coupling layers into a flow, meaning that we need to reorder the output $\mathbf{z}'$ in some way. This can be done by reversing the ordering of output features or, if using checkerboard masks (23), by alternating between even and odd indices:

$$b_d^{\text{even}} = d \bmod 2 \quad \text{and} \quad b_d^{\text{odd}} = 1 - d \bmod 2 \,. \tag{24}$$

Both reorderings work, however there exists an alternative approach that tries to learn feature permutations using 1×1 convolution [17] (see appendix A).

The idea is to learn a weight matrix $\mathbf{W}$ that will serve as a generalization of a permutation matrix. To construct a flow model we need to calculate $\log|\det\mathbf{W}|$, which has a cost of $\mathcal{O}(D^3)$. In order to reduce this to $\mathcal{O}(D)$, LU decomposition is used. The weights $\mathbf{W}$ are first initialized as a random rotation matrix with determinant of 0

$$\mathbf{W}, \mathbf{R} = \text{QR}\left[\mathcal{N}(0, \mathbf{I}_{D\times D})\right] \,, \tag{25}$$

where we have used QR decomposition to get an orthogonal matrix discarding the upper triangular matrix $\mathbf{R}$. Parameterizing $\mathbf{W}$ by its LU decomposition gives

$$\mathbf{W} = \mathbf{PL}(\mathbf{U} + \text{diag}(\mathbf{s})) \,, \tag{26}$$

where $\mathbf{P}$ is a permutation matrix, $\mathbf{L}$ is a lower triangular matrix with ones on the diagonal, $\mathbf{U}$ is an upper triangular matrix with zeros on the diagonal and $\mathbf{s}$ is a vector. The determinant is then

$$\log|\det\mathbf{W}| = \sum_{i=1}^{D} \log|\mathbf{s}_i| \,. \tag{27}$$

Learning (optimization) is performed on $\mathbf{L}$, $\mathbf{U}$ and $\mathbf{s}$ with $\mathbf{P}$ fixed by its initial value. The difference in computational time becomes significant for large dimensions D.

4 Results

4.1 Data preprocessing

Machine learning algorithms do not perform well when input features have very different scales (i.e. numerical values). That is why data normalization (also known as feature scaling) has to be performed before training. In the case of described Higgs dataset this was done in three steps:

1. min-max normalization
$$\mathbf{x}_1 = \frac{\mathbf{x} - \min(\mathbf{x})}{\max(\mathbf{x}) - \min(\mathbf{x})} \,, \tag{28}$$
2. logit transformation (inverse of logistic function)
$$\mathbf{x}_2 = \log\frac{\mathbf{x}_1}{1 - \mathbf{x}_1} \,, \tag{29}$$
3. standardization
$$\mathbf{x}_3 = \frac{\mathbf{x}_2 - \boldsymbol{\mu}(\mathbf{x}_2)}{\boldsymbol{\sigma}(\mathbf{x}_2)} \,, \tag{30}$$

with similar inverse transformations. The reasoning behind this particular normalization is the observation that flows do not learn well on distributions with sharp edges (lepton and jet p_T distributions in our example). Min-max normalization scales the data to $[0, 1]$ range for the logit transformation that scales it to $(-\infty, \infty)$ range. In the last step standardization makes the values of each feature have zero mean and unit variance, which further helps with neural network training.

4.2 ML event generation

The flow was constructed using 20 building blocks or stages. Each block consisted of batch normalization (see appendix B), permutation layer and a coupling layer in that order (Figure 6). The model was trained on $5 \cdot 10^5$ signal and background events. A list of all parameters is given in appendix C. Comparison between MC events and ML events is presented in Figure 5. Results show that a normalizing flow can learn the multidimensional distribution well. The precision of such a generative model would have to be on a level of resampling in order to replace MC generation from some data point onward. The total achievable precision of our model is sadly still insufficient to meet this rigorous requirement and one would have to invest in a more complex setup. Generated p_T distributions are somewhat acceptable while other generated features are inadequate. Distribution matching, which will not be further considered here, can be more rigorously investigated using two sample tests that determine whether the difference between two populations is statistically significant.

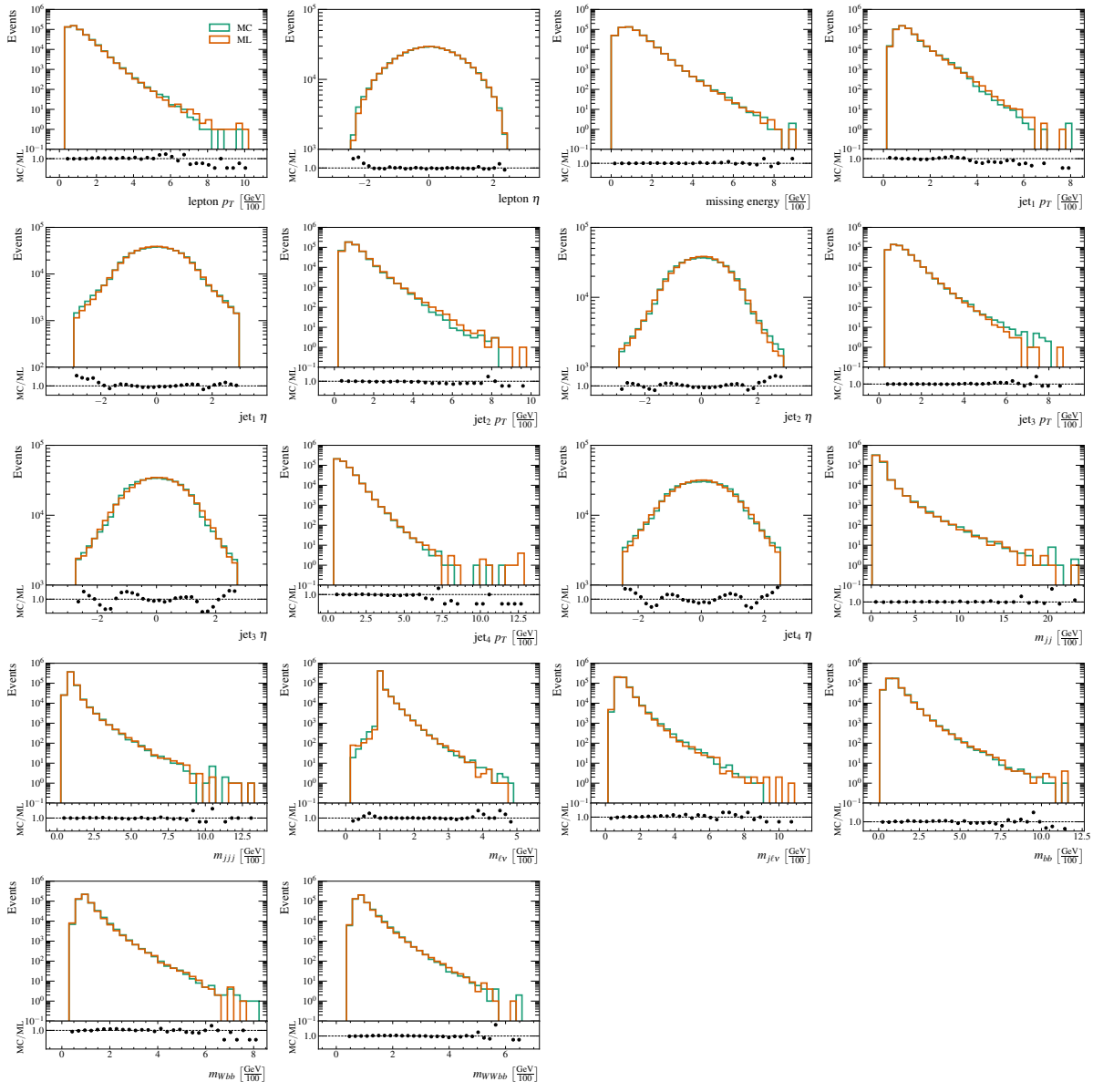


Figure 5: Comparison between MC and flow generated distributions of observables.

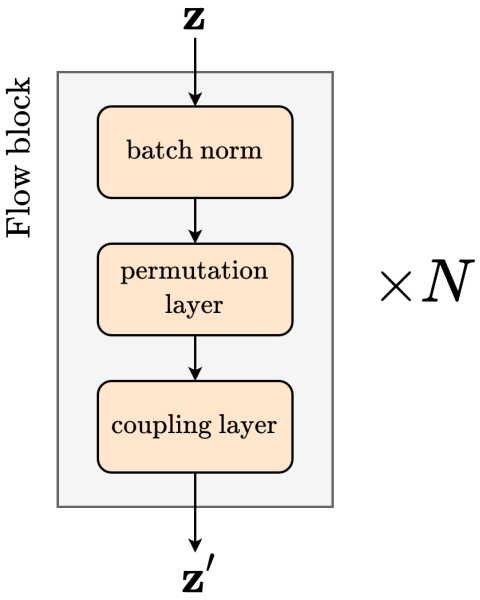


Figure 6: One building block of a flow adapted from Ref. [17].

5 Discussion

Machine learning is a rapidly advancing field of study, producing methods that are potentially applicable in particle physics tasks. One class of methods are generative models that learn probability distributions of observables and are capable of simulating events that closely match simulated or experimental data. The seminar described an exact log-likelihood generative approach to event generation called a normalizing flow. Flow based models are capable of density estimation and sampling. Density estimation is done using the change of variables formula, while sampling is done by generating data points from a simple distribution and then transforming to a more complex distribution using functions implemented by deep neural networks. In order to accomplish this a flow needs to be invertible, differentiable and have an efficiently computable Jacobian determinant. The seminar presented a concrete solution to this problem using affine coupling layers. The method was then used on a simulated dataset of theoretical Higgs boson production and showed promising results when used for sampling (event generation).

The flow was constructed using affine coupling layers implemented using realNVP model and then further enchanced with learnable permutations and batch normalization as in Glow model (some toy examples are in appendix D). The results, although promising, are not the current best generative models can do. The method was primarily chosen due to its simplicity and importance because it serves as a base to newer more complex architectures. For example: the affine transformation can be changed to include linear or quadratic splines [18], coupling layers can be enhanced with adding stochastic layers [19] and the base distribution can be replaced with the resampled version [20]. All of this changes increase the flexibility and add to the expressiveness of the flow, thus making it more accurate.

In conclusion, let us look at the future of particle physics experiments and generative modeling. First, looking at the LHC computing requirements for physics simulation and analysis it is clear that the increase in the rate of collision will result in much higher data amounts and even more complex events to analyze. The machine learning techniques, which are especially well suited for large amounts of complex data, will play an important role in new physics searches. Secondly, looking at the machine learning community we can see that the research in generative modeling has shifted to diffusion models in the last year or so (see Ref. [21]). This kind of models are already showing promising results (even beating GANs in image generation) but are still in very early stages.

References

[1] ATLAS Collaboration. "The ATLAS Experiment at the CERN Large Hadron Collider". In: *Journal of Instrumentation* 3.08 (Aug. 2008), S08003–S08003. DOI: 10.1088/1748-0221/3/08/s08003.

[2] Bobak Hashemi et al. *LHC analysis-specific datasets with Generative Adversarial Networks*. 2019. arXiv: 1901.05282 [hep-ex].

[3] Sydney Otten et al. *Event Generation and Statistical Sampling for Physics with Deep Generative Models and a Density Information Buffer*. 2019. arXiv: 1901.00875 [hep-ph].

[4] *Worldwide LHC Computing Grid*. URL: https://wlcg-public.web.cern.ch/.

[5] P Calafiura et al. *ATLAS HL-LHC Computing Conceptual Design Report*. Tech. rep. Geneva: CERN, Sept. 2020. URL: https://cds.cern.ch/record/2729668.

[6] HEPiX Benchmarking Working Group. *HEP-SPEC06*. 2017. URL: https://w3.hepix.org/benchmarking.html.

[7] P. Baldi, P. Sadowski, and D. Whiteson. "Searching for exotic particles in high-energy physics with deep learning". In: *Nature Communications* 5.1 (July 2014). ISSN: 2041-1723. URL: http://dx.doi.org/10.1038/ncomms5308.

[8] Ivan Kobyzev, Simon JD Prince, and Marcus A Brubaker. *Normalizing Flows: An Introduction and Review of Current Methods*. 2020. arXiv: 1908.09257 [stat-ML].

[9] George Papamakarios et al. "Normalizing flows for probabilistic modeling and inference". In: *Journal of Machine Learning Research* 22.57 (2021), pp. 1–64.

[10] Ian Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].

[11] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2014. arXiv: 1312.6114 [stat.ML].

[12] Kevin P. Murphy. *Probabilistic Machine Learning: An introduction*. https://probml.github.io/pml-book/book1.html. MIT Press, 2022.

[13] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG]. URL: https://pytorch.org/.

[14] Simon Širca. *Probability for physicists*. Springer, 2016.

[15] Lilian Weng. "Flow-based Deep Generative Models". In: (2018). Accessed February 28 2022. URL: https://lilianweng.github.io/posts/2018-10-13-flow-models/.

[16] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. *Density estimation using Real NVP*. 2016. arXiv: 1605.08803 [cs.LG].

[17] Durk P Kingma and Prafulla Dhariwal. "Glow: Generative flow with invertible 1x1 convolutions". In: *Advances in neural information processing systems* 31 (2018).

[18] Thomas Müller et al. "Neural importance sampling". In: *ACM Transactions on Graphics (TOG)* 38.5 (2019), pp. 1–19. arXiv: 1808.03856 [cs.LG].

[19] Hao Wu, Jonas Köhler, and Frank Noé. "Stochastic normalizing flows". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 5933–5944.

[20] Vincent Stimper, Bernhard Schölkopf, and José Miguel Hernández-Lobato. "Resampling Base Distributions of Normalizing Flows". In: (2021). arXiv: 2110.15828 [stat.ML].

[21] *Awesome Diffusion Models*. Accessed March 13 2022. URL: https://github.com/heejkoo/Awesome-Diffusion-Models.

A Invertible 1×1 convolution

In machine learning convolution operation is referring to two dimensional cross correlation

$$[\mathbf{W} \star \mathbf{X}](k,i,j) = \sum_{u=1}^{H} \sum_{v=1}^{W} \sum_{c=1}^{C_{\text{in}}} w_{k,u,v,c}\, x_{i+u,j+v,c}\,, \tag{31}$$

where k indexes the output channels C_{out}. The above equation is meant for images $\mathbf{X}$ with height H, width W and number of input channels C_{in}. Matrix $\mathbf{W}$ is known as a filter matrix or a kernel of convolution and is learned by the neural network. A 1×1 convolution, i.e. $H = W = 1$, is given by

$$[\mathbf{W} \star \mathbf{X}](k,i,j) = \sum_{c=1}^{C_{\text{in}}} w_{k,c,1,1}\, x_{i,j,c} \tag{32}$$

and changes the number of channels from C_{in} to C_{out}, without changing the spatial dimensionality. This can be written as matrix multiplication

$$[\mathbf{W} \star \mathbf{X}](k,l) = \sum_{c=1}^{C_{\text{in}}} w_{k,c} x_{c,l} = \mathbf{W}\mathbf{X}\,, \tag{33}$$

where we have reshaped $\mathbf{W}$ to $C_{\text{out}} \times C_{\text{in}}$ and $\mathbf{X}$ to $C_{\text{in}} \times (H \cdot W)$. The dimensionality has thus been changed from $C_{\text{in}} \times H \times W$ to $C_{\text{out}} \times H \times W$. This can be seen as a convolutional layer consisting of fully connected layer applied at every single i, j pixel location to transform the C_{in} corresponding input values into C_{out} output values. In practice this is typically used to adjust the number of channels between network layers and to control model complexity. Since we are mostly interested in tabular data and not in images we need to replace channels with features giving us

$$[\mathbf{W} \star \mathbf{x}](k) = \sum_{d=1}^{D_{\text{in}}} w_{k,d} x_d = \mathbf{W}\mathbf{x}\,, \tag{34}$$

where k goes from 1 to D_{out}. The above can be used to learn a generalization of a permutation matrix $\mathbf{W}$, if we set $D_{\text{in}} = D_{\text{out}}$.

B Batch normalization

In order to further speed up and stabilize flow training batch normalization is introduced at the end of each coupling layer. Batch norm and its inverse consist of two affine transformations:

$$\text{BN}(\mathbf{z}) = \boldsymbol{\alpha} \odot \frac{\mathbf{z} - \boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2 + \boldsymbol{\epsilon}}} + \boldsymbol{\beta}\,, \tag{35}$$

$$\text{BN}^{-1}(\mathbf{z}') = \boldsymbol{\mu} + \frac{\mathbf{z}' - \boldsymbol{\beta}}{\boldsymbol{\alpha}} \odot \sqrt{\boldsymbol{\sigma}^2 + \boldsymbol{\epsilon}}\,. \tag{36}$$

The first has scale $\boldsymbol{\sigma}^2$ and translation parameters $\boldsymbol{\mu}$ set by the batch statistics, the second has optimization parameters $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ that again represent scale and translation. A fixed parameter $\boldsymbol{\epsilon}$ is added for numerical stability. Updates of batch parameters are done as

$$\begin{aligned} \boldsymbol{\mu}_{b+1} &= \rho \boldsymbol{\mu}_{b-1} + (1-\rho)\boldsymbol{\mu}_b\,, \\ \boldsymbol{\sigma}^2_{b+1} &= \rho \boldsymbol{\sigma}^2_{b-1} + (1-\rho)\boldsymbol{\sigma}^2_b\,, \end{aligned} \tag{37}$$

where b is an index of the current batch and ρ a hyperparameter known as momentum. Batch norm is an elementwise operation and thus has an easy to compute determinant

$$\log|\det J_{\text{BN}}(\mathbf{z})| = \sum_{i=1}^{D} \log \frac{\alpha_i}{\sqrt{\sigma_i^2 + \epsilon_i}}\,, \tag{38}$$

which is needed in flow construction. Batch norm can thus be viewed as yet another flow layer and can be inserted between consecutive coupling layers:

$$\ldots \circ T_k \circ \mathrm{BN} \circ T_{k-1} \circ \ldots \tag{39}$$

It is important to note that the above procedure approximates dataset statistics with batch statistics which in turn only works for large enough batches.

C Model parameters

model	value	training	value
activation	ReLU	learning rate	10^{-4}
hidden layer dim.	512	L2	10^{-6}
BN ρ	10^{-5}	optimizer	Adam
# hidden layers	2	epochs	35
# blocks	20	batch size	1024

Table 1: Model and training parameters.

D Toy datasets

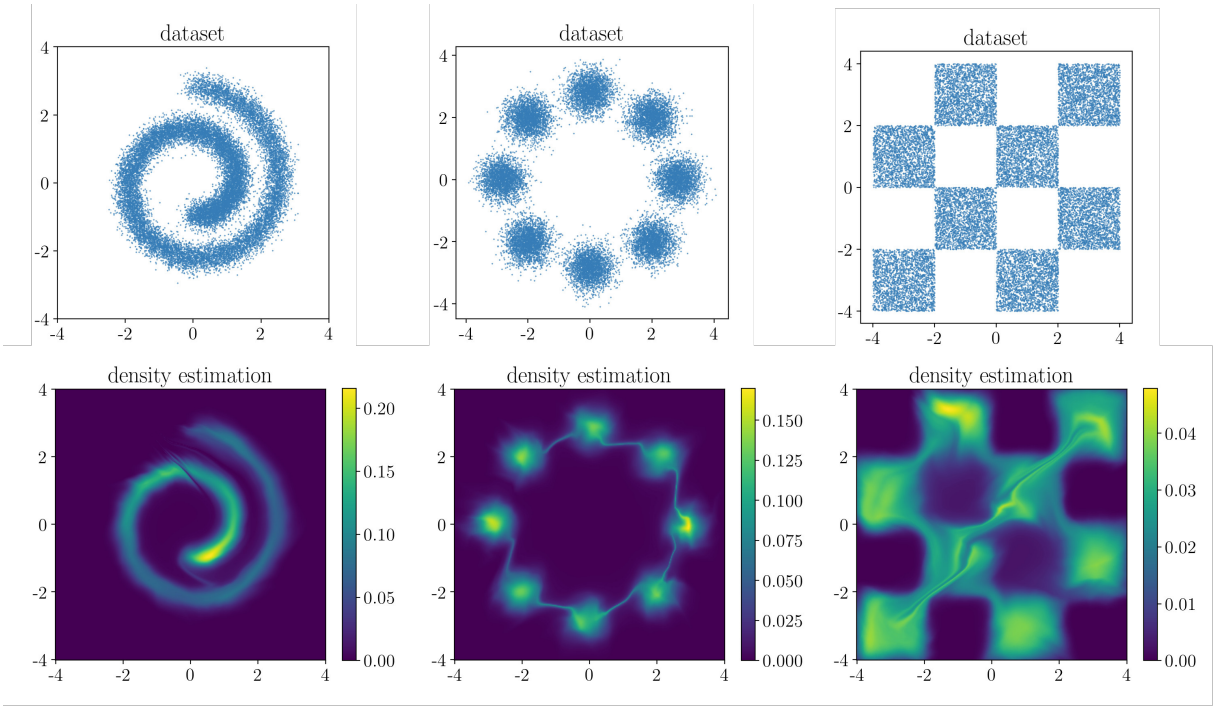


Figure 7: Swissroll, eight Gaussians and checkerboard density estimation.