Estimating probability distributions using stratified discrete normalizing flows

(Szacowanie rozkładów prawdopodobieństwa
z użyciem stratyfikowanych dyskretnych modeli przepływu)

Mateusz Basiak

Praca inżynierska

Promotor: dr Rafał Nowak

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

5 lipca 2023

Abstract

In this work, we attempt to estimate various functions of the data, such as mean or fraction of samples above a certain threshold, using only a small subset of the data. In order to do that, we generate new samples using the discrete normalizing flow models. Those deep learning models learn a given probability distribution of the data by learning its transformation into a well known distribution.
We further decrease variance of our estimators by using stratification. It is a technique where we divide the space of values that we sample into several areas and we sample from each independently. That allows us to distribute available budget of samples better and sample more from areas with higher variance, decreasing overall variance of the estimators.
We present both the theoretical foundations behind the described mechanisms and their practical implementation. We test those implemented algorithms on a variety of datasets, both those containing real-life meteorology data and artificial ones. We analyze results and attempt to draw conclusions about quality and viability of our methods.

W tej pracy próbujemy estymować funkcje zbioru danych, takie jak jego średnia czy liczba próbek powyżej pewnej wartości, mając dostępny tylko pewien niewielki podzbiór całego zbioru danych. W tym celu generujmy nowe próbki używając dyskretnych modeli przepływu. Są to modele oparte na sieciach neuronowych, które uczą się zadanego rozkładu danych poprzez próby transformacji go w pewien określony, znany rozkład.
Redukujemy wariancję naszych estymatorów poprzez wprowadzenie techniki zwanej stratyfikacją. Polega ona na dzieleniu przestrzeni możliwych wartości na obszary i niezależnym generowaniu próbek z każdego z nich. Pozwala to skupić generowanie próbek na obszarach o największej wariancji i w ten sposób zmniejszyć wariancję estymatorów.
W tej pracy prezentujemy zarówno teoretyczne podstawy przedstawianych algorytmów, jak i ich implementację. Następnie testujemy je na kilku różnorodnych zbiorach danych, zarówno takich zawierających rzeczywiste dane meteorologiczne, jak i na sztucznie wygenerowanych. Analizujemy i porównujemy wyniki wykonanych testów i próbujemy stwierdzić, jak dobrze radzą sobie przedstawiane przez nas techniki.

Contents

Chapter 1

Introduction

Data generation is one of the fundamental tasks in machine learning. In statistics, we calculate functions based on samples. In machine learning and deep learning, the models learn using given data. In both cases, accuracy of the results corresponds directly to the size (and quality) of the available dataset. Unfortunately, for many real-life problems large datasets do not exist, or are hard to access. That poses a natural problem of creating a model that can generate artificial samples similar to some small given dataset.

Over the years, many approaches were developed to combat this task. One of them is the autoregressive models, where each subsequent output depends not only on the input and stochastic variables, but also on previous outputs. Such models include Long Short-Term Memory [6] developed by Hochreiter and Schmidhuber for text generation, or PixelCNN [13] created by Van den Oord et al. for generating pictures. Those algorithms are sequential in nature and that makes them hard to parallelize and limits their computational efficiency.

Another common framework is called Generative Adversarial Networks (GANs) developed by Goodfellow et al. [4]. In a GAN, there are two separate neural networks: a **generator** that tries to generate a sample similar to those from a given set $\mathcal{D}$ and a **discriminator** whose task is to determine whether a given sample is from $\mathcal{D}$, or it was generated by the generator. These networks play a zero-sum game in which the generator is rewarded for fooling the discriminator and the discriminator is rewarded for guessing correctly. That allows GANs to train in the unsupervised setting.

Finally, there are the normalizing flow models introduced by Dinh et al. [2]. Here, the model learns a reversible transformation g that maps an arbitrarily complex distribution of the input data into a simple, well known latent distribution such as Gaussian or logistic distributions. We think of this process as a normalization, hence the name of the model. Then we sample from the known distribution and use g^{-1} to generate desired data points. Flow models can be further divided into

discrete flow models, where the transformation g is a combination of discrete operations such as addition, multiplication or convolution (NICE [2], Real NVP [3], GLOW [8]) and continuous flow models (FFJORD [5]).

In this thesis, we attempt to use discrete normalizing flow models NICE and Real NVP to generate samples. We train them on various one- and two-dimensional datasets, with both artificial and real-life data. We use those samples to estimate various statistics better and with more confidence than with the original data. Finally, we use a method called stratification to further reduce variance of our estimators.

In Chapter 2, we present the main ideas and some mathematical background behind the NICE and Real NVP models. In Chapter 3, we outline the idea of stratification and its various variants. We discuss our implementation of those ideas in Chapter 4 and then the results of our experiments in Chapter 5. Concluding remarks are presented in Chapter 6.

Chapter 2

Discrete normalizing flow models

Let us consider a D-dimensional random variable $X \in \mathbb{R}^D$ with distribution p_X. We sample it independently n times and thus obtain a dataset $\mathcal{D}$. We want to calculate the expected value of some real-valued function f on X, meaning $I = \mathbb{E}[f(X)]$ using only $\mathcal{D}$, without any further knowledge about X. Function f can be as straightforward as the mean of X, but also a more elaborate one.

One way to do that would be to assume that p_X is a known distribution, for example multidimensional Gaussian, and then estimate its parameters based on $\mathcal{D}$. We will not make such assumptions and accept that p_X can be arbitrarily complex.

It should be clear that the quality of our estimation corresponds directly to the size of $\mathcal{D}$. Therefore our strategy is to train a deep learning model to generate samples from p_X (or at least the distribution similar to p_X, based on the training on $\mathcal{D}$). Then our model will generate a much larger dataset $\mathcal{D}'$ and we will estimate I using $\mathcal{D} \cup \mathcal{D}'$. For our generative model, we choose discrete normalizing flow models. We will test two classical algorithms from that framework: NICE [2] and Real NVP [3].

2.1 General idea

Assumption 2.1 *We treat our generative model as a transformation $g : \mathbb{R}^D \to \mathbb{R}^D$. We want it to have two key properties.*

1. *For every $x \sim p_X$, transformation $y = g(x)$ satisfies $y \sim p_H$, where p_H is some known distribution called prior distribution. We use the standard Gaussian distribution $\mathcal{N}(0, 1)$ in every dimension as H.*

2. *Transformation g is invertible, meaning that g^{-1} can be quickly computed.*

Reasons for both of those properties arise from the training and generation. During training, we transform samples from $\mathcal{D}$ using model g. We calculate the loss based on the difference between the resulting distribution and p_H and therefore we need to know how to calculate $p_H(x)$, as shown in Section 2.1.1. During generation, we generate samples from p_H and transform them using g^{-1}. That also implies that, for practical reasons, p_H should be a well known distribution, so that there are known ways of sampling from it.

2.1.1 Loss function

In order to train our models, we need to define a loss function that will be maximized during training. The only information we have about the distribution of X are the samples from $\mathcal{D}$. Therefore our loss will maximize the probability in those points. First, by the change of variables formula,

$$p_X(x) = p_H(g(x)) \cdot \left| \det \frac{\partial g(x)}{\partial x} \right|. \tag{2.1}$$

The above formula can be understood in terms of the conservation of mass principle. Probability distributions p_X, p_H can be thought of as mass density functions. For any $A \subseteq \mathbb{R}^D$ we want transformation g to keep its mass. That means that the volume of A times density p_X on A should equal the volume of $g(A)$ times its density p_H. Equation (2.1) is derived by dividing by the volume of A and reducing A into a single point.

Now, by taking logarithms of both sides of (2.1), we obtain the loss:

$$\log(p_X(x)) = \log p_H(g(x)) + \log \left| \det \frac{\partial g(x)}{\partial x} \right|. \tag{2.2}$$

We note that the loss uses the Jacobian determinant of g. That places an important restriction on the transformation.

Observation 2.2 *The transformation g must have an efficiently computable Jacobian determinant.*

The second component of the sample loss is $\log p_H(g(x))$. For the Gaussian distribution, it is equal to:

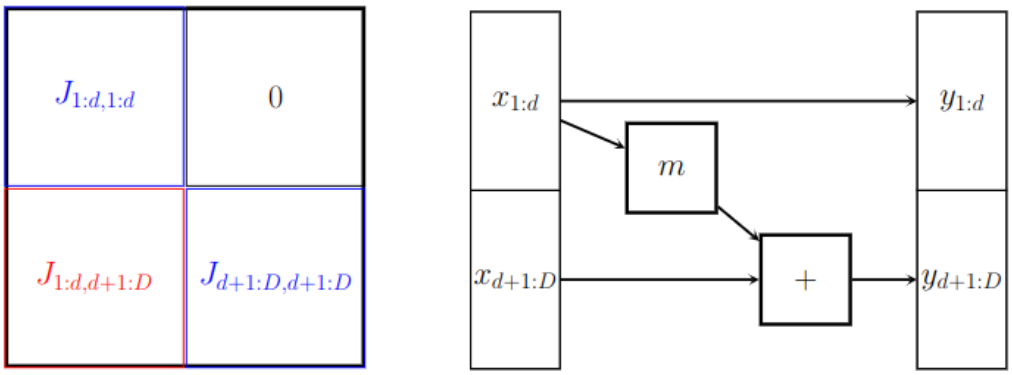
$$\log p_H(g(x)) = \sum_{i=1}^{D} \left(-\frac{1}{2} y_i^2 + \log(2\pi) \right), \tag{2.3}$$

where y_i is the value in the i-th dimension of $y = g(x)$.

2.2 NICE

Non-linear Independent Component Estimation (NICE) is the model introduced by Dinh et al. [2]. Their first key observation is that when the Jacobian matrix of g is triangular, its determinant is just the product of the elements on the diagonal and therefore it is easy to compute.

The NICE model contains multiple layers called **coupling layers**. In each of them, we divide input vector $x \in \mathbb{R}^D$ into two vectors: one $x_{1:d}$ containing the first d elements of x ($1 \leq d < D$) and the other $x_{d+1:D}$ containing the rest of the vector x. This naturally divides the Jacobian J of the transformation $g(x)$ into four parts, as illustrated in Figure 2.1a.



(a) NICE Jacobian matrix J.

(b) Computational graph of a NICE coupling layer.

Figure 2.1

To make matrix J lower triangular, matrices $J_{1:d,1:d}$ and $J_{d+1:D,d+1:D}$ need to be lower triangular. That means that the submatrix $J_{1:d,d+1:D}$ (highlighted red in Figure 2.1a) can be arbitrary, and the transformation connected with it can be very complex. Now we can define the NICE coupling layer [2]:

$$\begin{aligned} y_{1:d} &= x_{1:d}, \\ y_{d+1:D} &= x_{d+1:D} + m(x_{1:d}). \end{aligned} \tag{2.4}$$

Here the transformation m is a deep neural network. The flow of computations inside the layer is demonstrated in Figure 2.1b. Matrices $J_{1:d,1:d}$ and $J_{d+1:D,d+1:D}$ are identity matrices, so matrix J of this layer is triangular. Its determinant is equal to 1, therefore satisfying Observation 2.2. Moreover it satisfies property 2 from the Assumption 2.1, as it is easily reversible:

$$\begin{aligned} x_{1:d} &= y_{1:d}, \\ x_{d+1:D} &= y_{d+1:D} - m(y_{1:d}). \end{aligned}$$

In the subsequent layer the split is reversed, meaning that the identity is applied to $x_{d+1:D}$ and $x_{1:d}$ is modified. Therefore the model has the capacity to learn on

every coordinate and can satisfy property 1 from the Assumption 2.1. It is important to note, that the NICE model is a composition of reversible layers and therefore the entire model is also reversible.

2.2.1 Scaling

As mentioned above, Jacobian determinant of the coupling layer (2.4) is equal to 1. It is therefore volume-preserving and the NICE model, which is the composition of those layers, also preserves volume. To combat this issue, the NICE model introduces a non volume-preserving transformation [2]. The output vector of NICE is multiplied by a diagonal scaling matrix T of the form:

$$T = \begin{pmatrix} e^{t_1} & 0 & 0 & \dots & 0 \\ 0 & e^{t_2} & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & e^{t_D} \end{pmatrix}.$$

This allows the model to give more weight to certain dimensions. The values of $t_1, t_2, \dots, t_D$ are also learned, therefore we introduce to (2.2) an additional factor equal to $\sum_{i=1}^{D} t_i$ which is the logarithm of the determinant of T.

2.3 Real NVP

Real-valued Non-Volume Preserving (Real NVP) model was presented by Dinh et al. [3]. It builds on the NICE model by introducing a slightly more complicated transformation. Here, in each layer, we also divide input vector x into two parts. We introduce an additional deep neural network s. It also takes in vector $x_{1:d}$, but its results are multiplied with $x_{d+1:D}$, instead of being added to it.

The computations of the coupling layer are demonstrated in Figure 2.2 and are as follows:

$$\begin{aligned} y_{1:d} &= x_{1:d}, \\ y_{d+1:D} &= (x_{d+1:D} \odot \exp(s(x_{1:d}))) + m(x_{1:d}), \end{aligned} \tag{2.5}$$

where the $\odot$ operation is the Hadamard product or the element-wise product [3].

This transformation satisfies Assumption 2.1, as it remains reversible and, when stacked in layers similar to that in NICE, it has capacity to learn in each of its dimensions. The Jacobian matrix of the transformation remains triangular, but submatrix $J_{d+1:D,d+1:D}$ is no longer an identity matrix. Instead, it is diagonal and has elements of the vector $\exp(s(x_{1:d}))$ on its diagonal. Therefore the Jacobian determinant of this transformation is the exponential of the sum of the elements of vector $s(x_{1:d})$. We note that the transformation is no longer volume-preserving, therefore scaling is no longer required in Real NVP.

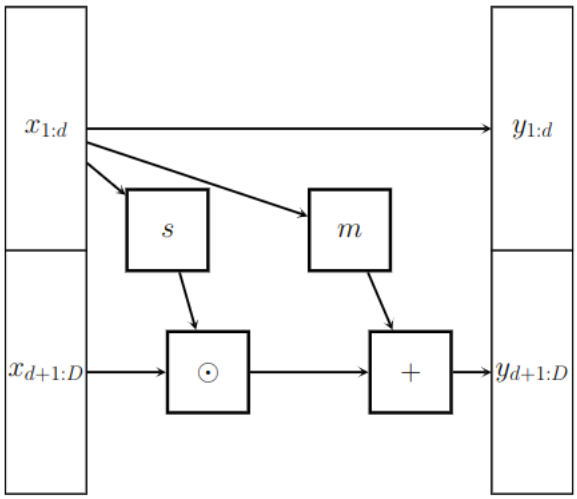


Figure 2.2: Computational graph of a Real NVP coupling layer.

Intuitively, Real NVP captures complexity faster than NICE, because it performs a full linear map in every dimension (among the dimensions $d+1$ to D) with coefficients given by the deep neural networks.

Chapter 3

Stratification

Now let us go back to the main problem of estimating $I = \mathbb{E}[f(X)]$. Let us assume (by the results presented in the previous chapter) that we can sample a random variable $Y = f(X)$ and want to sample it R times to obtain the best possible estimation of its expected value. We will call R the simulations budget.

Definition 3.1 *We call an estimator $\hat{Y}$ unbiased when $\mathbb{E}[\hat{Y}] = \mathbb{E}[Y]$.*

Let $\hat{Y}$ be any unbiased estimator. To give us confidence about its predictions, such estimator should have a possibly narrow confidence interval. By the central limit theorem, if the variance of Y is finite, the confidence interval of $\hat{Y}$ is correlated to its variance [1, Equation 23.5]:

$$\mathbb{P}\left(I \in \left[\hat{Y} - z_{1-\alpha/2} \cdot \sqrt{\mathrm{Var}(\hat{Y})}, \hat{Y} + z_{1-\alpha/2} \cdot \sqrt{\mathrm{Var}(\hat{Y})}\right]\right) \approx 1 - \alpha.$$

Therefore we want to select an estimator with the smallest possible variance.

Denote the R obtained samples as $Y_1, Y_2, \ldots, Y_R$. One possible estimator is the sample mean $\hat{Y}_m^{CMC} = 1/R \cdot \sum_{i=1}^{R} Y_i$, also called the Crude Monte Carlo (CMC) estimator. Its variance is $\mathrm{Var}(\hat{Y}_R^{CMC}) = \mathrm{Var}(Y)/R$. In practice we often do not know the true variance of Y, therefore we estimate it using sample variance.

Definition 3.2 *Sample variance $\hat{S}_R^2$ of a sample $Y_1, Y_2, \ldots, Y_R$ with mean $\overline{Y}_R$ is equal to:*

$$\hat{S}_R^2 = \frac{\sum_{i=1}^{R}(Y_i - \overline{Y}_R)^2}{R-1}.$$

This yields the estimated variance of the estimator:

$$\widehat{\mathrm{Var}}(\hat{Y}_R^{CMC}) = \frac{\sum_{i=1}^{R}(Y_i - \hat{Y}_R^{CMC})^2}{R \cdot (R-1)}.$$

While the sample mean is a good estimator when the distribution is smooth, its variance becomes very large when distribution has peaks and valleys. A natural idea is to sample from every area of the distribution to get more information about the local curvature of probability density function and attempt to sample more data points from regions with the highest variance. We try to achieve those goals through stratification. The stratification method described in this chapter follows the one explained by Lorek et al. [9].

Stratified sampling [12] is an estimation method designed to reduce variance of estimating $I = \mathbb{E}[Y]$ compared to the classic Monte Carlo sampling. For a given number $k \geq 1$, we divide the space of values of Y into k disjoint regions $A_1, A_2, \ldots, A_k$ such that

$$\mathbb{P}\left(Y \in \bigcup_{i=1}^{k} A_i\right) = 1.$$

Every such region A_j defines a stratum $B_j = \{\omega : Y(\omega) \in A_j\}$, however we will also refer to A_j itself as a stratum. For each A_j we also define $p_j = \mathbb{P}(Y \in A_j)$ and $I_j = E[Y|Y \in A_j]$. We will always assume that all strata have equal probability, meaning $p_j = 1/k$ for all j.

The method works as follows: we divide the simulations budget R among all the strata into $R_1, R_2, \ldots, R_k$. Then, for each stratum A_j, let Y^j be the variable Y limited to the stratum A_j. We generate R_j samples from it and estimate I_j via the sample mean $\hat{Y}^j = 1/R_j \cdot \sum_{i=1}^{R_j} Y_i^j$. Finally we calculate the stratified estimator:

$$\begin{aligned} \hat{Y}_R^{strat.} &= p_1\hat{Y}^1 + p_2\hat{Y}^2 + \ldots + p_k\hat{Y}^k \\ &= \frac{1}{k}\sum_{j=1}^{k}\hat{Y}^j \\ &= \frac{1}{k}\sum_{j=1}^{k}\frac{\sum_{i=1}^{R_j} Y_i^j}{R_j}. \end{aligned} \tag{3.1}$$

Its variance is equal to:

$$\mathrm{Var}(\hat{Y}_R^{strat.}) = \sum_{j=1}^{k}\frac{p_j^2}{R_j}\mathrm{Var}(Y^j) = \sum_{j=1}^{k}\frac{\mathrm{Var}(Y^j)}{k^2 \cdot R_j}.$$

Again, in practice we do not know variance of any Y^j, so we use sample variance.

$$\widehat{\mathrm{Var}}(\hat{Y}_R^{strat.}) = \sum_{j=1}^{k}\frac{\hat{S}_{R_j;j}^2}{k^2 \cdot R_j} \tag{3.2}$$

where $\hat{S}_{R_j;j}^2$ is the sample variance of the j-the stratum, calculated using R_j observations.

In the above description of the stratification method, there are two missing pieces: the procedure for creating strata and the procedure of dividing R among the strata. Both of them have many possible solutions leading to many possible versions of the stratification algorithm.

3.1 Allocation of samples to strata

We will start by addressing the second issue. One possible solution is to allocate the number of samples to each stratum proportionally to its p_j, meaning $R_j = R \cdot p_j = R/k$. This is called **proportional allocation**. We denote the estimator calculated using this allocation as $\hat{Y}_R^{PA}$. Equations (3.1) and (3.2) take the form of:

$$\begin{aligned}
\hat{Y}_R^{PA} &= \frac{1}{k}\sum_{j=1}^{k}\frac{\sum_{i=1}^{R_j} Y_i^j}{R_j} \\
&= \frac{1}{k}\sum_{j=1}^{k}\frac{k}{R}\cdot\sum_{i=1}^{R_j} Y_i^j \\
&= \frac{1}{R}\sum_{j=1}^{k}\sum_{i=1}^{R_j} Y_i^j \\
\widehat{\mathrm{Var}}(\hat{Y}_R^{PA}) &= \sum_{j=1}^{k}\frac{\hat{S}^2_{R_j;j}}{k^2\cdot R_j} \\
&= \frac{1}{k\cdot R}\sum_{j=1}^{k}\hat{S}^2_{R_j;j}.
\end{aligned}$$

Another possible strategy involves looking into the variance of each stratum. As illustrated in (3.2), variance of each stratum divided by the number of samples in that stratum is a component of the variance of the estimator. Therefore we want to place many samples in strata with high variance to compensate. This seems intuitive, as when the variance in the stratum is small, we obtain a lot of information about Y with every observation, meanwhile when the variance is high, the observation in one part of the stratum is not indicative of the values in the other parts. The following theorem from Madras [10] formalizes this intuition.

Theorem 3.3 *[10, Theorem 3.3] Let us fix strata $A_1, \ldots, A_k$ and simulations budget R. Let $\hat{Y}_R^{strat.}$ be a stratified estimator with a general split $R = R_1 + R_2 + \ldots + R_k$ and $\hat{Y}_R^{OPT}$ be a stratified estimator with a split:*

$$R_j = R \cdot \frac{p_j\sigma_j}{\sum_{i=1}^{k} p_i\sigma_i}.$$

where σ_j^2 is the variance of A_j. Then we have $Var(\hat{Y}_R^{OPT}) \leq Var(\hat{Y}_R^{strat.})$.

In our case probabilities cancel out, so the number of simulations is directly proportional to the variance of the stratum. We note that, to use this method, we need to know the variance of each stratum ahead of producing any samples. We do not know them in advance and using sample variance requires generating some observations first. Therefore we divide our original budget R into two parts: first R^{pilot} observations are distributed proportionally and are used to calculate

sample means of each stratum. The other $R' = R - R^{pilot}$ samples are distributed according to Theorem 3.3. That yields the final distribution of the budget:

$$R_j = R' \cdot \frac{\hat{S}_{R^{pilot}/k;j}}{\sum_{i=1}^{k} \hat{S}_{R^{pilot}/k;i}}.$$

We call this allocation strategy the **optimal allocation** and denote the estimator calculated with (3.1) with optimal allocation as $\hat{Y}_R^{OPT}$. After performing all observations we can recalculate all sample means. Then we can estimate the variance as:

$$\begin{aligned}
\widehat{\mathrm{Var}}(\hat{Y}_R^{OPT}) &= \sum_{j=1}^{k} \frac{\hat{S}_{R_j;j}^2}{k^2 \cdot R_j} \\
&= \sum_{j=1}^{k} \frac{1}{k^2} \cdot \hat{S}_{R_j;j}^2 \cdot \frac{\sum_{i=1}^{k} \hat{S}_{R_i;i}}{R \cdot \hat{S}_{R_j;j}} \\
&= \frac{1}{k^2 \cdot R} \left(\sum_{j=1}^{k} \hat{S}_{R_j;j} \cdot \sum_{i=1}^{k} \hat{S}_{R_i;i} \right) \\
&= \frac{1}{k^2 \cdot R} \left(\sum_{j=1}^{k} \hat{S}_{R_j;j} \right)^2.
\end{aligned}$$

3.2 Cartesian stratification

The other missing piece in the stratification algorithm is the method of creating the strata. Two main methods are the **cartesian stratification** where we divide the space in each dimension independently and the **spherical stratification** where we divide the space based on the spherical coordinates [9].

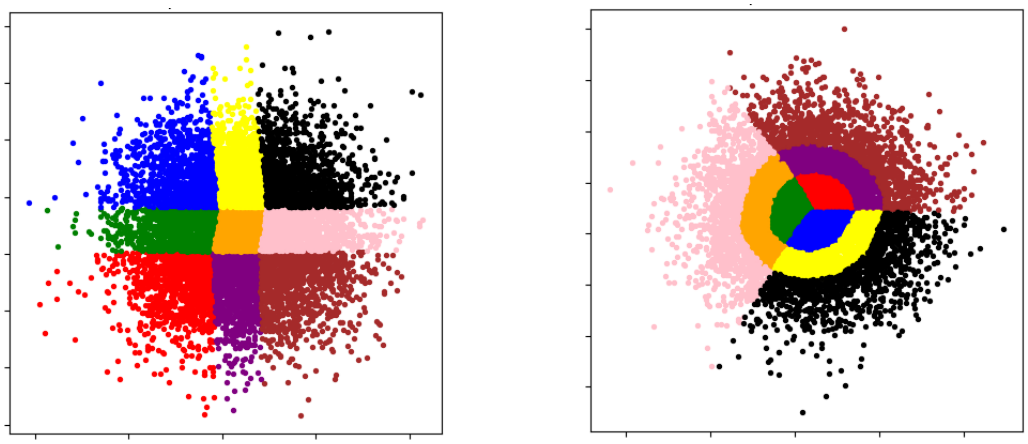
We note that, due to the form of our generative model, we only need to sample from the multivariate normal distribution, as we can pass such generated sample through g^{-1} and therefore generate a sample from the desired distribution.

Let $(Z_1, \ldots, Z_D)$ be the standard normal D-dimensional random variable. To perform cartesian stratification, we want to split each of its dimensions independently into m strata, obtaining $m^D = k$ strata (see Figure 3.1a).

We now focus on one dimension. Let $Z \sim \mathcal{N}(0, 1)$ be a random variable with probability distribution $F_{\mathcal{N}}$. For the set of probabilities $p_1, p_2, \ldots, p_m$ (in our case $p_i = 1/m$ for every i), we define the strata as intervals $A_1 = (a_0, a_1], A_2 = (a_1, a_2], \ldots, A_m = (a_{m-1}, a_m)$ where $a_0 = -\infty, a_m = +\infty$ and for every other i, $a_i = F_{\mathcal{N}}^{-1}(p_1 + \ldots + p_i)$. Such definition guarantees that $\mathbb{P}(Z \in A_i) = p_i$ for every i. Let $U \sim \mathcal{U}[0, 1]$ be the random variable with an uniform distribution and let Z^j be

the random variable from the j-th stratum of Z. We sample Z^j as follows:

$$Z^j = F_{\mathcal{N}}^{-1}\left(\sum_{i=1}^{j-1} p_i + p_j \cdot U\right). \tag{3.3}$$



(a) (b)

Figure 3.1: Example of stratified two-dimensional space for $k = 9$ strata. Different strata are colored in different colors. Cartesian stratification is on the left and spherical stratification is on the right.

3.3 Spherical stratification

Every point in the D-dimensional space $\mathbb{R}^D$ can be represented in spherical coordinates as a combination of angles on $(D-1)$-dimensional unit hypersphere S_{D-1} and a radius. The normal distribution is symmetric, so we can stratify each of those values independently into k_{ang} and k_{rad} strata, respectively, and therefore obtain spherical stratification with $k_{ang} \cdot k_{rad} = k$ strata (see Figure 3.1b).

Let $Z = (Z_1, \ldots, Z_D)$ be the standard normal D-dimensional random variable. Radius B of Z from the origin can be calculated with $B^2 = Z_1^2 + Z_2^2 + \ldots + Z_D^2$. Therefore B^2 has distribution χ_D^2 (χ^2 with D degrees of freedom). That means that we can stratify radius similarly to the cartesian stratification, using the formula

$$(B^j)^2 = F_{\chi_D^2}^{-1}\left(\sum_{i=1}^{j-1} p_i + p_j \cdot U\right). \tag{3.4}$$

for the j-th stratum. We observe that the points from normalized Z, meaning $(Z_1/||Z||, Z_2/||Z||, \ldots, Z_D/||Z||)$ are distributed uniformly on S_{D-1} [11]. Therefore we sample uniformly from S_{D-1} and multiply every dimension by B^j to obtain the variable with the same distribution as Z.

We also want to stratify the angles. That means that we want to divide S_{D-1} into k_{ang} disjoint areas that will be our strata. In general it is not an easy task,

however, as mentioned in Chapter 1, we will only operate on one-dimensional and two-dimensional distributions. In one dimension the hypersphere is reduced to a pair of points, so the stratification with regards to angle loses a lot of its meaning and the stratification yields results very similar to the cartesian stratification. Therefore we do not perform spherical stratification for one-dimensional distributions.

In two dimensions, any point $x \in S_1$ can be represented by a pair of coordinates

$$\begin{aligned} x_1 &= \sin(\phi), \\ x_2 &= \cos(\phi) \end{aligned} \tag{3.5}$$

where $\phi \in [0, 2\pi)$. Both of the coordinates depend on one variable ϕ. Therefore we can divide $[0, 2\pi)$ into k_{ang} strata with lengths proportional to their probabilities. Let $p_1, p_2, \ldots, p_{k_{ang}}$ be the desired probabilities of the strata and $b_0 = 0, b_1 = p_1, b_2 = p_1 + p_2, \ldots, b_{k_{ang}} = \sum_{i=1}^{k_{ang}} p_i$. We sample from the j-th stratum using the formula:

$$\phi^j = b_{j-1} + (b_j - b_{j-1})U. \tag{3.6}$$

Inserting that to (3.5) and combining with radius stratification yields full spherical stratification.

Chapter 4

Implementation

In this chapter, we discuss our implementation of the ideas described in Chapter 2 and Chapter 3 and we present key fragments of our code. For our implementation, we used Python version 3.10 with NumPy and PyTorch libraries.

4.1 Models

First, we introduce the `CouplingLayer` class that implements a single coupling layer. It is used both by the NICE and Real NVP models, as the mechanics of their layers do not differ much. It is initialized with the following signature.

```
1 class CouplingLayer(nn.Module):
2   def __init__(self, in_dim=2, layer_type='additive',
3                parity=False, hidden_layers=5, hidden_dim=40):
```

Listing 4.1: The initialization method signature of the `CouplingLayer` class.

The `layer_type` variable signals the type of model that the layer belongs to. It can be either 'additive', meaning the NICE model or 'multiplicative' for the Real NVP model.

In both of the models, we divide the dimensions of the input vector x into $x_{1:d}$ and $x_{d+1:D}$. That poses a problem, when $D = 1$. In that case we duplicate the input to create a two-dimensional input vector, therefore we always assume that $D = 2$. That yields $d = D/2 = 1$ as the only possibility. The layers can therefore be divided into two groups, depending on which input is changed. Information about the layer group is given in the boolean variable `parity`. The other variables `hidden_layers`, `hidden_dim` refer to the dimensions of the neural networks used by the layer. Each of the networks m, s used in (2.4) and (2.5) has the same dimensions with 5 hidden layers, each with hidden dimension size 40. It totals 5041 parameters for a NICE layer and 10082 parameters for a Real NVP layer.

The most important methods implemented in the `CouplingLayer` class are the `forward` and `backward` methods. They perform the transformation g described in Chapter 2 and its inverse, respectively. In Listing 4.2, we present the `forward` method, that implements transformations (2.4) and (2.5).

```
def forward(self, x):
  x0 = x[self.mask1]
  m = self.add_net(x0)
  if self.layer_type == 'multiplicative':
    s = self.mult_net(x0)
    s = torch.tanh(s)
    s1 = torch.matmul(s, self.unmask)
    x = x * torch.exp(s1)
    y = x + torch.matmul(m, self.unmask)
    return y, torch.abs(torch.sum(s))
  elif self.layer_type == 'additive':
    y = x + torch.matmul(m, self.unmask)
    return y
```

Listing 4.2: The forward method of the `CouplingLayer` class.

Instead of creating the vector $x_{1:d}$ explicitly, it masks one of the inputs using `self.mask1`, changing its value to 0. Then the result of each neural network computation is masked again using `self.unmask`, which zeroes the input other than `self.mask1`.

Next, we implement classes `NICEModel` and `RNVPModel` that represent the entire structure of the models. Each of them contains a very simple `forward` method that runs the input through the subsequent layers. In our tests, we use 4 coupling layers in each model with the masked input alternating between the neighbouring layers. That yields the total of 20166 parameters for each of the NICE models and 40330 parameters for each of the Real NVP models.

```
def forward(self, x):
  if self.dim_parity:
    x = x.tile(2)
  for layer in self.layers:
    x = layer(x)
  y = x * torch.exp(self.scaling)
  loss = self.loss_fun(y) + torch.sum(self.scaling)
  return y, loss
```

Listing 4.3: The forward method from the `NICEModel` class.

The total numbers of parameters calculated above for each model includes ad-

ditional 2 parameters used for scaling, as described in Section 2.2.1. The Real NVP model does not theoretically require scaling, but our experiments show that it improves its results, therefore it is implemented for that model as well. The scaling diagonal matrix T is implemented as a vector `self.scaling` $= (t_1, t_2)$.

The models calculate loss using the Gaussian loss described in (2.2) and (2.3).

Both models include an identical `sample()` method used to produce a single sample. This procedure samples from the multivariate normal distribution using the `torch.randn()` method from PyTorch and then calls the `backward` method of the model to obtain the sample from the desired distribution. We note that the models always transform a two-dimensional vector into the two-dimensional vector. That is not a problem in the `forward` method, but when we generate samples using the `backward` method, we sometimes expect one-dimensional samples. In those cases we average over the output to obtain a single-dimensional data point.

```
1 def sample(self):
2   y = torch.randn(self.in_dim)
3   x = self.backward(y)
4   return x
```

Listing 4.4: The sample method from the `NICEModel` class.

4.2 Training

We train using the optimizer Adam [7] from PyTorch (with learning rate $\gamma = 0.001$, betas $\beta_1 = 0.9$, $\beta_2 = 0.999$, epsilon $\epsilon = 10^{-8}$ and weight decay $\lambda = 0$). We divide the given dataset into the training set containing roughly 90% of samples and the evaluation set. Training lasts 300 epochs with each epoch being a pass through the entire training dataset. We backpropagate after every 32 samples. In every pass we add small Gaussian noise to each sample to avoid overfitting.

4.3 Statistics

We evaluate the performance of our generative models on three statistics. For each of them, the input is a set of samples X.

1. $I_1(X)$ — fraction of samples from X that has values above 60 in every dimension.
2. $I_2(X)$ — the mean of the values of X across all dimensions.
3. $I_3(X)$ — sum of $x-50$ for each $x > 50$ in all samples in X across all dimensions.

We round all the results to 6 decimal places and multiply by 1000 the variance of the estimators of the first and third statistic to obtain clearer results.

4.4 Stratification

The basic generating function is presented in the Listing 4.5. It uses the `sample` method implemented in the models.

```
1 def generate(model, R=10000):
2   model.eval()
3   generated_values = []
4   with torch.no_grad():
5     for _ in range(R):
6       y = model.sample().numpy()
7       generated_values.append(tuple([i for i in y]))
8
9   return generated_values
```

Listing 4.5: The basic generating function.

4.4.1 Cartesian stratification

We improve the generation by implementing stratification, as explained in Chapter 3. We use $k = 9$ strata, three in each dimension. In cartesian stratification, each stratum s is a vector of segments, each being a part of the interval $[0, 1]$ of length $1/3$. To sample from stratum s, we use `cartesian_one_stratum` function presented in Listing 4.6. For each dimension, it samples a probability uniformly from the appropriate segment and uses the probability point function (from the SciPy library) to transform it into a point from the corresponding segment of $\mathcal{N}(0, 1)$, as in (3.3). With the values in all dimensions, it calls the `backward` method of the model directly.

```
1  def cartesian_one_stratum(model, s, R, m):
2    values = []
3    with torch.no_grad():
4      for _ in range(R):
5        x = []
6        for i in range(len(s)):
7          p = random.uniform(s[i]/m, (s[i]+1)/m)
8          x.append(norm.ppf(p))
9        y = model.backward(torch.tensor(x, dtype=torch.float)).numpy()
10       values.append(tuple([i for i in y]))
11   return values
```

```
12
13 def generate_cart_pro(model, R=2000, m=3, D=2):
14   model.eval()
15   Rj = R // int(math.pow(m, D))
16   strata = [s for s in product(range(m), repeat=D)]
17   generated_values = [cartesian_one_stratum(model, s, Rj, m) for s
         in strata]
18   return generated_values
```

Listing 4.6: Proportional cartesian stratification.

Function `generate_cart_pro` presented in Listing 4.6 implements proportional cartesian stratification. It creates m^D strata as tuples of numbers from 0 to $m-1$. Then, for each stratum s, it calls function `cartesian_one_stratum` with budget $Rj = R/9$.

Listing 4.7 shows the implementation of the optimal cartesian allocation. We first allocate budget of $R_{pilot} = R/8$ samples equally to every stratum. After generating those samples in lines 9-13, we use Theorem 3.3 to produce optimal allocation of the simulations budget in line 17.

```
1  def generate_cart_opt(model, statistic, R=2000, m=3, D=2):
2    model.eval()
3    R_pilot = R // 8
4    Rj_pilot = R_pilot // int(math.pow(m, D))
5    R -= R_pilot
6    strata = [s for s in product(range(m), repeat=D)]
7    generated_values = []
8
9    strata_std_deviations = []
10   for i in range(len(strata)):
11     values = cartesian_one_stratum(model, strata[i], Rj_pilot, m)
12     _, var = statistic(values)
13     strata_std_deviations.append(math.sqrt(var))
14
15   sum_std_deviations = sum(strata_std_deviations)
16   for i in range(len(strata)):
17     Rj = round(R * strata_std_deviations[i] / sum_std_deviations)
18     generated_values[i] += cartesian_one_stratum(model, strata[i],
           Rj, m)
19   return generated_values
```

Listing 4.7: Optimal cartesian stratification.

4.4.2 Spherical stratification

In spherical stratification, each stratum s is represented by a pair of intervals, one $[sr0, sr1]$ for the radius and one $[sa0, sa1]$ for the angle. They are both subsegments of length $1/3$ of the segment $[0, 1]$. Generation from one stratum s is presented in Listing 4.8. First, we sample the radius in accordance with (3.4) in line 6. Then in line 7 we sample the angle as was described in (3.6). Finally, we compute the coordinates in line 8 using (3.5) and pass them to the `backward` method of the model. The functions implementing the proportional and optimal allocation have a very similar implementation to the ones used in the cartesian stratification.

```
1  def spherical_one_stratum(model, sr0, sr1, sa0, sa1, R):
2    values = []
3    with torch.no_grad():
4      for i in range(R):
5        # Generating a sample by choosing radius and angle
6        r = math.sqrt(chi2.ppf(random.uniform(sr0, sr1), df=2))
7        phi = random.uniform(sa0, sa1) * 2.0 * math.pi
8        x = [r * math.cos(phi), r * math.sin(phi)]
9        y = model.backward(torch.tensor(x, dtype=torch.float)).numpy()
10       values.append(tuple([i for i in y]))
11
12   return values
```

Listing 4.8: Generation from one stratum in spherical stratification.

4.5 Data preparation

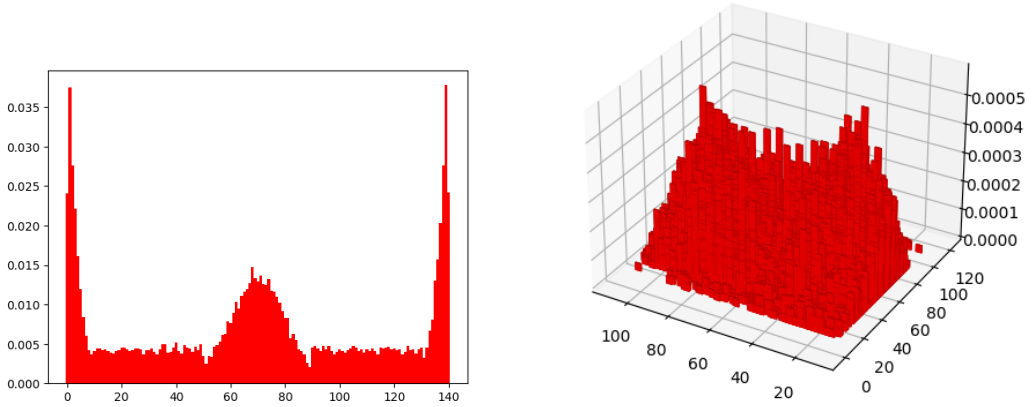
In our experiments, we use both artificial and real datasets. The real data comes from the European Climate Assessment & Dataset project, which accumulates the weather data from many stations across Europe. We concentrate on the data about temperature and humidity from the Wrocław station. They were recorded once every day from the beginning of 1951, and the available data ends in May of 2023, which yields around 26 000 observations. The temperature data is the minimum temperature recorded during that day with precision of 0.1 degrees Celsius. It is multiplied by 10, so the value is approximately from the range of -300 to 400. The humidity data is the average humidity during the preceding 24 hours measured in percentage.

Some of the data is flagged as unclear or missing, there are also samples outside the reasonable range of values. After erasing those data points, we obtained the 'temp' and 'humi' datasets with around 16 000 data points each. We also combined that data, to obtain a two-dimensional dataset 'temp_humi'.

We also created two artificial datasets. One of them, called 'art1d', is one-dimensional, and the other, called 'art2d', is two-dimensional. The 'art1d' dataset (see Figure 4.1a) was created by dividing the space into regions and then sampling each region from another distribution. Such variety is supposed to test the capabilities of our models and mechanisms. It is comprised of the following parts:

- 5000 samples from the exponential distribution with $\lambda = 0.3$, filtered to contain only samples from the interval $[0, 10]$,
- 5000 samples from the $\mathcal{U}[10, 50]$ distribution,
- 10000 samples from the $\mathcal{N}(70, 10)$ distribution,
- 5000 samples from the $\mathcal{U}[90, 130]$ distribution,
- 5000 samples from the exponential distribution with $\lambda = -0.3$, shifted by 140 and filtered to contain only samples from the interval $[130, 140]$.

That yields a total of 29 157 data points after filtering out all the unwanted samples.



(a) PDF of 'art1d' dataset.

(b) PDF of 'art2d' dataset.

Figure 4.1

The 'art2d' dataset (see Figure 4.1b) was created by first sampling 20 000 samples of the form $(80 - ex(0.04), 20 + ex(0.06))$ where $ex(x)$ is the exponential distribution with the parameter x. Then we added noise sampled from $\mathcal{N}(0, 30)$ in each dimension. Finally, for each point (x, y) in the dataset, we added $(110 - x, 110 - y)$, to create a second peak and we restricted the data to points within the $[0, 110] \times [0, 110]$ rectangle. That procedure yielded the dataset of 37 091 samples.

Chapter 5

Experiments

From each of our datasets, we randomly choose $N = 1000$ samples that are used as data available to train the models. Then, in each of our experiments, we sample $R = 10000$ observations from the model and compare the results from the original data, training dataset and generated sample on each of the three statistics from Section 4.3. We set seed of all the random generators to 0 to allow the reproducibility of our results. For the purposes of plotting, all data is rounded to the nearest integer. All the stratification for the plots is done using I_2. We present the results in tables, where for each estimator we calculate its error relative to the true value of the statistic (calculated based on the input data) and its variance.

5.1 One-dimensional datasets

We start our experiments with the one-dimensional datasets. As was mentioned in Section 3.3, we do not perform spherical stratification. We compare five results for each model and each dataset: the original dataset (called input data), small training set of size N, sample generated with Crude MC method using code from Listing 4.5, sample generated using proportional cartesian stratification and sample generated using optimal cartesian stratification. For each of the five mentioned results, we plot its probability distribution function (PDF) as well as its cumulative distribution function (CDF). We additionally plot the difference between the CDFs of the input data and the sample.

5.1.1 Humidity dataset

We start with the humidity dataset, as it resembles the Gaussian distribution the most. It should therefore be the easiest one to learn for our models, as they use the Gaussian distribution as their latent distribution. We use the NICE model. In Figure 5.1 and Table 5.2 we present the results of our experiments.

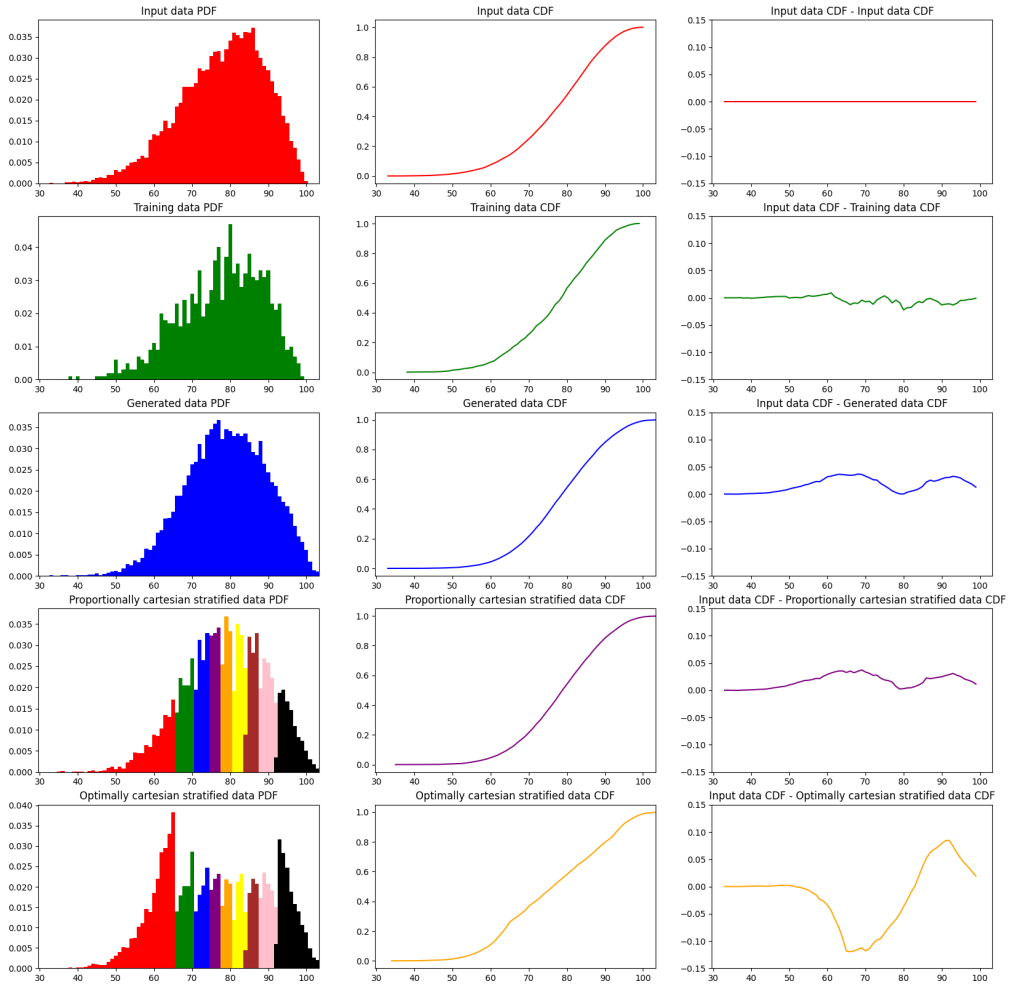


Figure 5.1: Plots of all generated data for the humidity dataset.

		True value	Training data	Crude Monte Carlo	Cartesian prop. stratified MC	Cartesian opt. stratified MC
I_1	Value	0.924280	0.931000	0.959100	0.957796	0.958033
	Relative error		0.007271	0.037673	0.036261	0.036519
	Variance		0.064303	0.003923	0.002620	0.000290
I_2	Value	77.899479	77.682000	79.033870	79.033325	79.056349
	Relative error		0.002792	0.014562	0.014555	0.014851
	Variance		0.119665	0.011406	0.000421	0.000287
I_3	Value	27.952408	27.722000	29.053992	29.050022	29.073621
	Relative error		0.008243	0.039409	0.039267	0.040112
	Variance		117.141858	11.273537	0.373547	0.268542

Table 5.2: Results of experiments using the NICE model for the humidity dataset. All columns except 'True value' represent estimators based on the given samples.

We observe that all the PDFs and CDFs, except the last one, are very similar to the original data. The model, even without stratification, learns to reproduce the data distribution. With optimal stratification, the strata at the endpoints have the highest variance, therefore they get the most samples in the allocation and the shape of the plot differs from the others in those spots. Nevertheless, this strategy

obtains very similar relative error to the other strategies on all statistics. All of the samples have relative errors from five to seven times worse than the training data the models were trained on. On the other hand, they obtain much lower variance than the training data – from ten times with the CMC estimator to about 500 times lower on I_2 by the cartesian optimally stratified estimator. With the relative error always remaining below 0.05, such difference in variance is a big improvement.

5.1.2 Temperature dataset

The results of our experiments for the temperature dataset are presented in Figure 5.3, Table 5.5 and Table 5.4. Plots for the NICE model are skipped, as they are very similar to those presented for the Real NVP model in Figure 5.3. Like for the humidity dataset, our model clearly learned the data distribution. The distribution still holds strong similarity to the normal distribution, therefore optimal stratification encounters the same problem as the one discussed in the previous section.

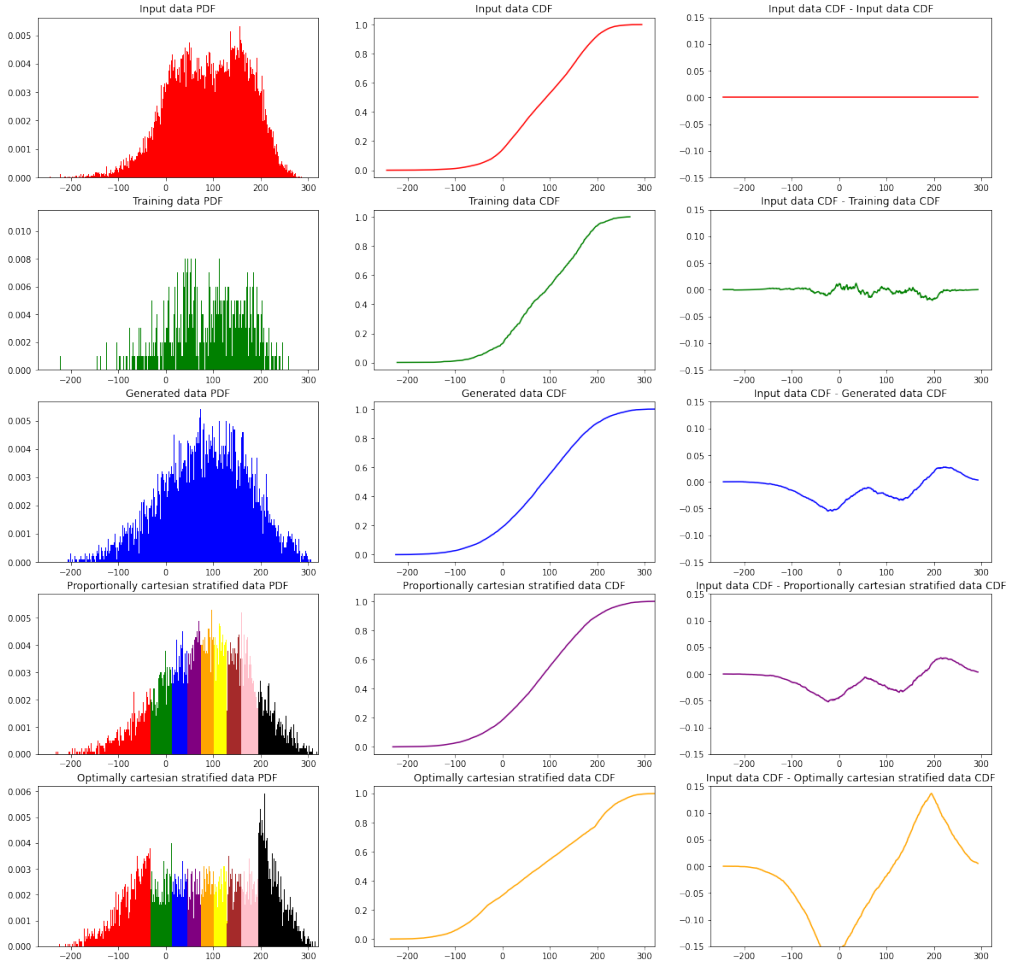


Figure 5.3: Plots of all generated data for the temperature dataset with the Real NVP model.

In Table 5.4 we can see that the NICE model obtains results similar to the humility dataset. The relative error is from 3 to 11 times worse than the one for the training data, but the variance drops significantly, even more so for the stratified data. Table 5.5 shows that Real NVP achieves a better performance. For I_1, it has at most approximately 3 times higher relative error than the training data and for I_3 it has much lower relative error, when using stratification, while keeping the variance low. There are almost no differences in relative error between both types of stratification, but optimal stratification has a lower variance, as was expected.

		True value	Training data	Crude Monte Carlo	Cartesian prop. stratified MC	Cartesian opt. stratified MC
I_1	Value	0.620694	0.616000	0.631300	0.639164	0.635476
	Relative error		0.007563	0.017087	0.029757	0.023815
	Variance		0.236781	0.023278	0.002072	0.000250
I_2	Value	89.978439	88.890000	93.719100	94.064706	93.744610
	Relative error		0.012097	0.041573	0.045414	0.041856
	Variance		6.423964	1.107149	0.026863	0.021021
I_3	Value	57.416724	56.258000	70.108000	70.290429	70.263246
	Relative error		0.020181	0.221038	0.224215	0.223742
	Variance		3273.020456	491.924986	8.143537	5.259442

Table 5.4: Results of experiments using the NICE model for the temperature dataset. All columns except 'True value' represent estimators based on the given samples.

		True value	Training data	Crude Monte Carlo	Cartesian prop. stratified MC	Cartesian opt. stratified MC
I_1	Value	0.620694	0.616000	0.605800	0.612061	0.610149
	Relative error		0.007563	0.023996	0.013908	0.016990
	Variance		0.236781	0.023883	0.002780	0.000309
I_2	Value	89.978439	88.890000	83.162900	84.403040	84.066246
	Relative error		0.012097	0.075746	0.061964	0.065707
	Variance		6.423964	0.828868	0.031176	0.021208
I_3	Value	57.416724	56.258000	56.134400	57.147815	57.085846
	Relative error		0.020181	0.022334	0.004683	0.005763
	Variance		3273.020456	383.063240	12.867153	6.161702

Table 5.5: Results of experiments using the Real NVP model for the temperature dataset. All columns except 'True value' represent estimators based on the given samples.

5.1.3 Artificial dataset

Results for the artificially generated dataset are presented in Figure 5.6, Table 5.8 and Table 5.7. The Real NVP model managed to capture some properties of the distribution, such as the spike on the left and a smaller peak in the middle, but the generated distributions resemble the original way less than with the previous distributions. That makes sense, as this distribution is composed of many very different parts, which makes it difficult to learn.

Table 5.7 shows that the NICE model obtains the relative error from 0.03 to 0.09 on all statistics. Once again stratification, while improving variance, does not

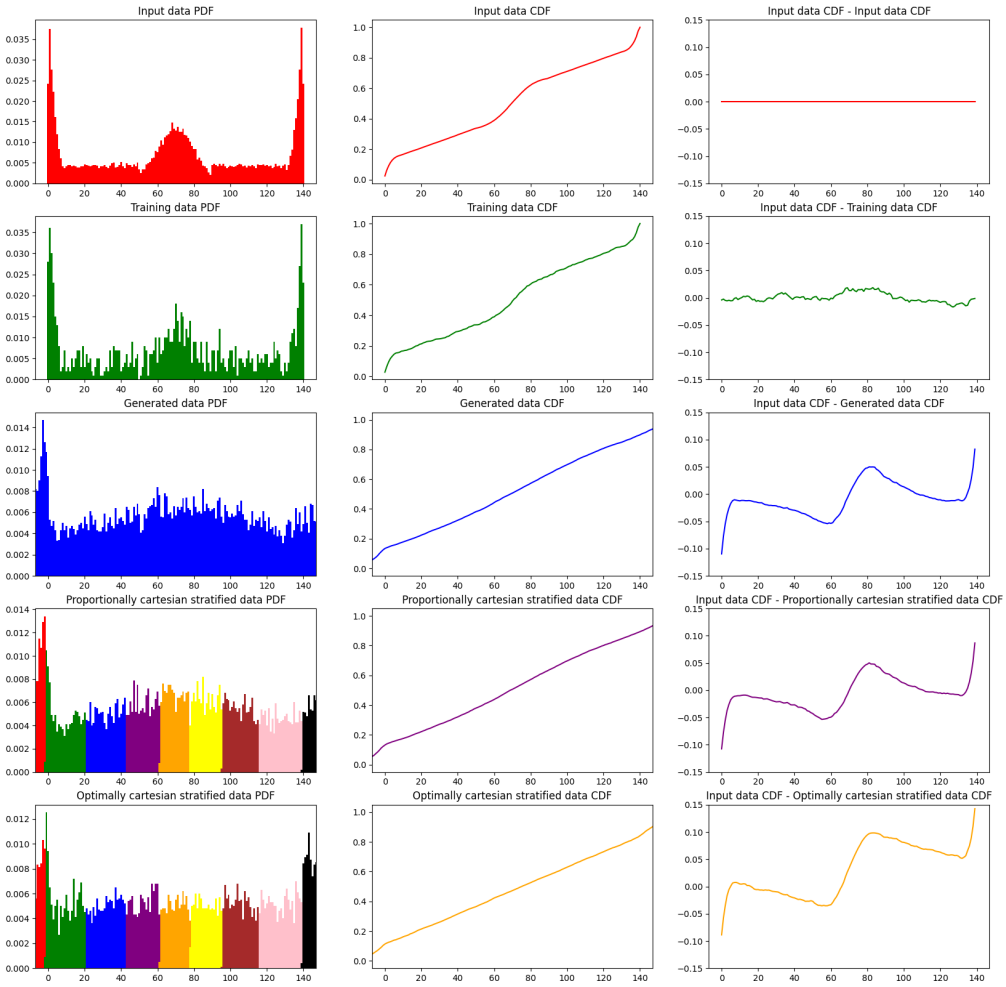


Figure 5.6: Plots of all generated data for art1d dataset with the Real NVP model.

		True value	Training data	Crude Monte Carlo	Cartesian prop. stratified MC	Cartesian opt. stratified MC
I_1	Value	0.610591	0.610000	0.557600	0.56557	0.566069
	Relative error		0.000968	0.086786	0.072118	0.072916
	Variance		0.238138	0.024671	0.000992	0.000106
I_2	Value	69.959186	70.018000	67.114000	67.236324	67.375657
	Relative error		0.000841	0.040669	0.038921	0.036929
	Variance		2.077960	0.289943	0.004370	0.003524
I_3	Value	31.094488	31.176000	32.814700	32.835283	32.923586
	Relative error		0.002621	0.055322	0.055984	0.058824
	Variance		1046.167191	133.030799	2.365059	1.499874

Table 5.7: Results of experiments using the NICE model for the art1d dataset. All columns except 'True value' represent estimators based on the given samples.

change the relative error by much. We note that the relative error is always at least 25 times higher than the relative error of the training data.

The Real NVP model, while performing similarly on I_1 and I_3, outperforms NICE model on I_2. Version with the optimal stratification has relative error around 8 times smaller than its NICE counterpart and is only around six times worse than

		True value	Training data	Crude Monte Carlo	Cartesian prop. stratified MC	Cartesian opt. stratified MC
I_1	Value	0.610591	0.610000	0.557300	0.561256	0.560944
	Relative error		0.000968	0.087278	0.080799	0.081310
	Variance		0.238138	0.024674	0.000541	0.000057
I_2	Value	69.959186	70.018000	68.827900	69.498650	69.632489
	Relative error		0.000841	0.016171	0.006583	0.004670
	Variance		2.077960	0.257366	0.004180	0.003973
I_3	Value	31.094488	31.176000	32.089500	32.695070	32.788185
	Relative error		0.002621	0.032000	0.051475	0.054469
	Variance		1046.167191	128.529862	2.745783	1.647354

Table 5.8: Results of experiments using the Real NVP model for the art1d dataset. All columns except 'True value' represent estimators based on the given samples.

the training data, while having a 500 times smaller variance. We also note that the proportionally and optimally stratified data has 2.5 and 3.5 times smaller relative error than the Crude MC version, respectively. That trend is reversed on I_3, where both stratified strategies achieve worse results than the version without stratification.

5.2 Two-dimensional datasets

Now we move on to the two-dimensional datasets. For each dataset, apart from the five results we had previously, we now add two sets from the spherical stratification: proportional and optimal. We also add the fourth column of the plot, where we track the distribution of samples on the two-dimensional plane in order to understand how the particular strata are transformed by our model.

5.2.1 Temperature-humidity dataset

The results of the NICE model for the temperature-humidity dataset are presented in Figure 5.9 and Table 5.10. The results of the Real NVP are presented in Figure 5.12 and Table 5.11. Both of the models preserve the general shape of the original data distribution. NICE transformation seems to twist the space more, as the layout of the strata is much more twisted and different from the one in Figure 3.1 than the layout of strata in the Real NVP transformation. The type of the stratification does not have such a big effect on the plot as it did in one dimension with the temperature and humidity distributions.

The NICE model performs similarly on all statistics, regardless of the type of stratification or lack thereof. It achieves from 0.03 to 0.12 relative error, which is several orders of magnitude higher than the training data in case of the first two statistics. It performs best on I_3, where its relative error is about ten times bigger compared to the training data. Its variance is smaller, but compared to the one dimensional datasets the difference is smaller, the Crude MC estimator has about ten times lower variance than the training data. We note that the spherical stratification

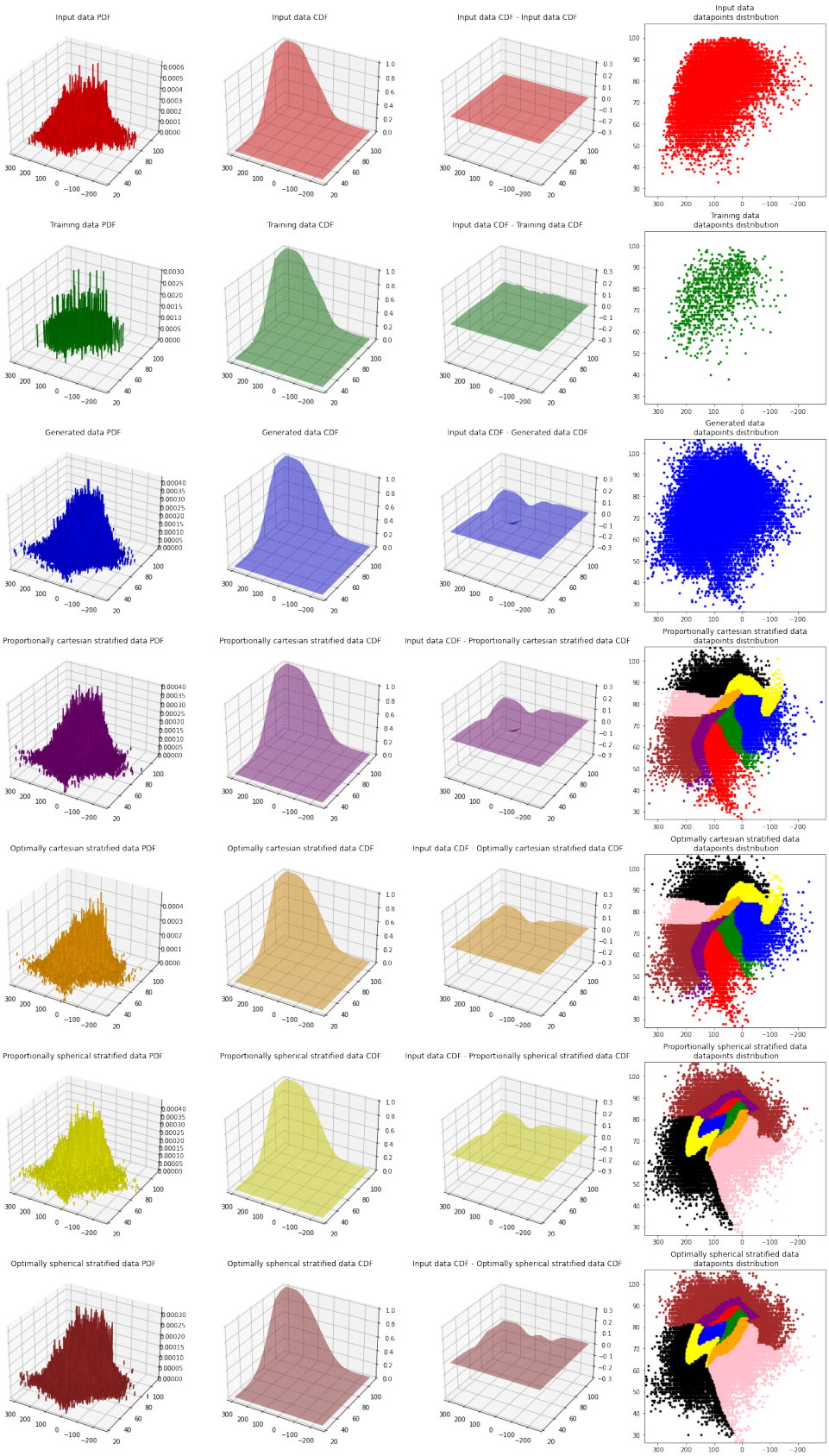


Figure 5.9: Plots of all generated data for temp_humi dataset with the NICE model.

	I_1		
	Value	Relative error	Variance
True value	0.549382		
Training data	0.549000	0.000695	0.247847
Crude Monte Carlo	0.529500	0.036190	0.024915
Cartesian prop. stratified MC	0.521952	0.049928	0.012869
Cartesian opt. stratified MC	0.522565	0.048814	0.009009
Spherical prop. stratified MC	0.528653	0.037732	0.017107
Spherical opt. stratified MC	0.522512	0.048910	0.014688
	I_2		
	Value	Relative error	Variance
True value	167.509073		
Training data	167.400000	0.000651	5.681443
Crude Monte Carlo	158.781500	0.052102	0.552096
Cartesian prop. stratified MC	158.719572	0.052472	0.121722
Cartesian opt. stratified MC	158.035588	0.056555	0.116612
Spherical prop. stratified MC	158.412241	0.054307	0.285203
Spherical opt. stratified MC	157.080749	0.062255	0.250576
	I_3		
	Value	Relative error	Variance
True value	72.318767		
Training data	71.457000	0.011916	4387.171322
Crude Monte Carlo	64.582600	0.106973	389.793717
Cartesian prop. stratified MC	64.466247	0.108582	101.007273
Cartesian opt. stratified MC	63.837264	0.117279	77.794681
Spherical prop. stratified MC	64.555056	0.107354	185.059751
Spherical opt. stratified MC	64.125270	0.113297	173.615493

Table 5.10: Results of experiments using the NICE model for the temp_humi dataset. All columns except 'True value' represent estimators based on the given samples.

has a higher variance than the cartesian stratification across all statistics. On I_3, where this difference is the biggest, it is about 1.5 times higher.

The Real NVP model performs comparably to the NICE model in the first two statistics. It performs approximately 3 times better on I_3, therefore it is just around 3 times worse than the training data, when using proportional spherical stratification. It is also worth noting, that the stratified estimators all have notably lower relative error than the Crude MC estimator on I_2 and I_3. Relative variance of the stratified estimators vary here from statistic to statistic, but the difference is the highest on I_3, where the situation is similar to the NICE model.

5.2.2 Artificial dataset

The experiments on the two-dimensional artificial dataset presented the biggest differences between the two tested flow models. Their results are presented: for the NICE model in Table 5.13 and for Real NVP in Table 5.14 and Figure 5.15. We can see that, especially on I_2 and I_3, NICE did not manage to learn the probability

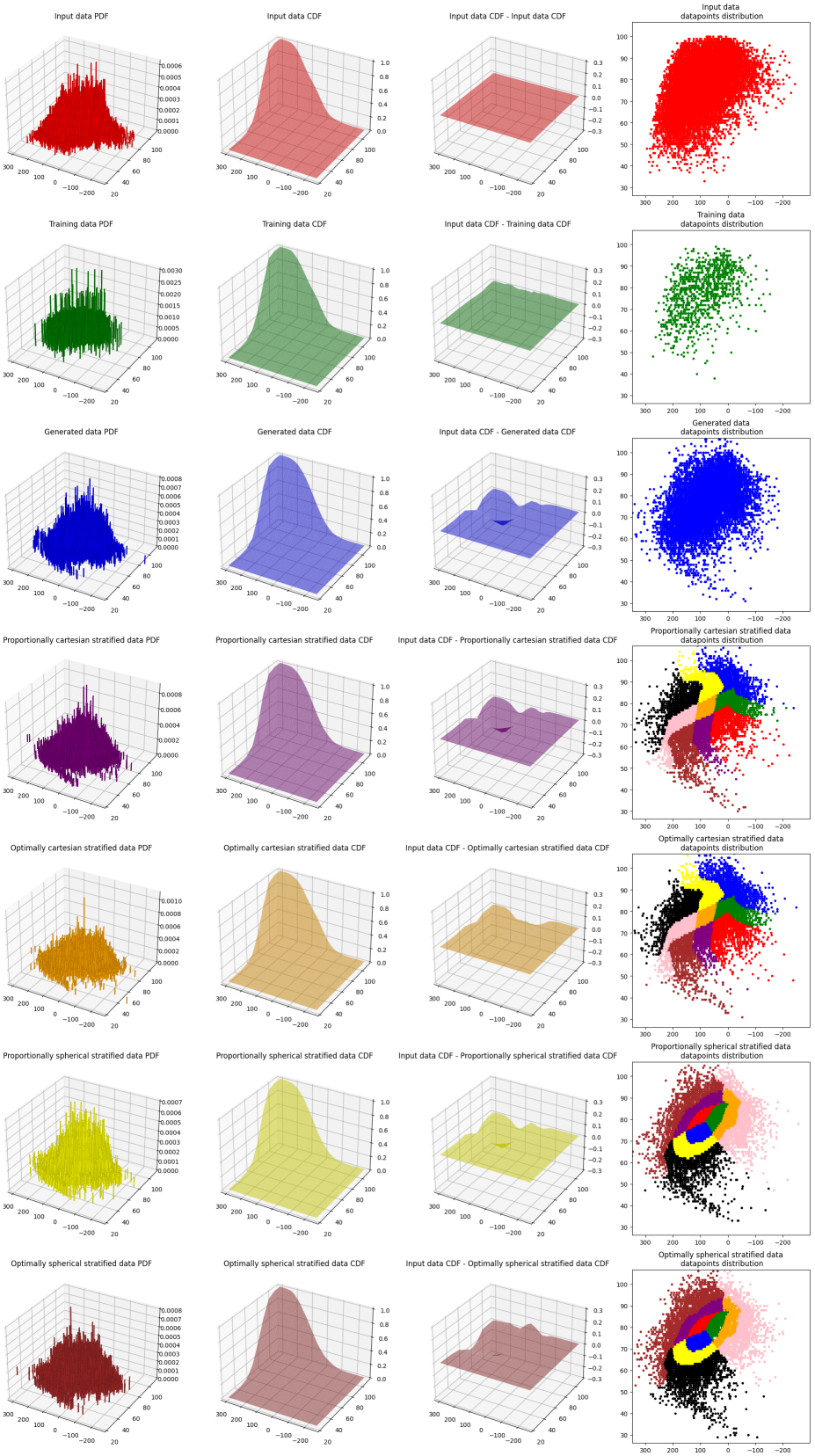


Figure 5.12: Plots of all generated data for temp_humi dataset with the Real NVP model.

	I_1		
	Value	Relative error	Variance
True value	0.549382		
Training data	0.549000	0.000695	0.247847
Crude Monte Carlo	0.560900	0.020965	0.024632
Cartesian prop. stratified MC	0.568557	0.034903	0.008872
Cartesian opt. stratified MC	0.575400	0.047359	0.006752
Spherical prop. stratified MC	0.573357	0.043640	0.009200
Spherical opt. stratified MC	0.573224	0.043397	0.005685
	I_2		
	Value	Relative error	Variance
True value	167.509073		
Training data	167.400000	0.000651	5.681443
Crude Monte Carlo	160.976900	0.038996	0.558231
Cartesian prop. stratified MC	163.443244	0.024272	0.149696
Cartesian opt. stratified MC	162.911032	0.27450	0.131394
Spherical prop. stratified MC	163.629263	0.023162	0.177309
Spherical opt. stratified MC	162.576337	0.029448	0.145820
	I_3		
	Value	Relative error	Variance
True value	72.318767		
Training data	71.457000	0.011916	4387.171322
Crude Monte Carlo	67.152000	0.071444	393.546964
Cartesian prop. stratified MC	69.415042	0.040152	98.756838
Cartesian opt. stratified MC	68.848940	0.047980	78.330523
Spherical prop. stratified MC	69.471147	0.039376	161.809280
Spherical opt. stratified MC	68.976329	0.046218	117.609624

Table 5.11: Results of experiments using the Real NVP model for the temp_humi dataset. All columns except 'True value' represent estimators based on the given samples.

distribution. Its estimators vary greatly from the data and its variance is very big.

On the other hand, Real NVP obtained much better results. On I_1 and I_3, the model managed to beat the training set in all strategies. On I_1, the optimal cartesian stratified estimator achieves the lowest relative error, which is also about 50 times lower than the training data error. It also performs very well on I_3, where it is just slightly worse than the optimal spherical stratified estimator, but still about 19 times smaller than the training data error. Those two optimal allocation strategies also have the lowest variance, about 75 times lower than the training set and approximately 7.5 times lower than the Crude MC estimator.

Those observations are confirmed by Figure 5.15. Plots generated by the Real NVP model look similar to the original dataset. The plot of the difference between the distributions is almost flat in all of the rows.

	I_1		
	Value	Relative error	Variance
True value	0.136125		
Training data	0.146000	0.072544	0.124809
Crude Monte Carlo	0.424400	2.117723	0.024431
	I_2		
	Value	Relative error	Variance
True value	111.969184		
Training data	112.548000	0.005169	0.906180
Crude Monte Carlo	-151.749700	2.355281	683.660982
	I_3		
	Value	Relative error	Variance
True value	12.671942		
Training data	13.640000	0.076394	528.528929
Crude Monte Carlo	878.518800	68.327874	188370.316316

Table 5.13: Results of experiments using the NICE model for the art2d dataset. All columns except 'True value' represent estimators based on the given samples.

	I_1		
	Value	Relative error	Variance
True value	0.136125		
Training data	0.146000	0.072544	0.124809
Crude Monte Carlo	0.139000	0.021120	0.011969
Cartesian prop. stratified MC	0.136914	0.005795	0.005786
Cartesian opt. stratified MC	0.136083	0.000308	0.002143
Spherical prop. stratified MC	0.137114	0.007263	0.004567
Spherical opt. stratified MC	0.135751	0.002750	0.000998
	I_2		
	Value	Relative error	Variance
True value	111.969184		
Training data	112.548000	0.005169	0.906180
Crude Monte Carlo	110.229900	0.015534	0.097233
Cartesian prop. stratified MC	110.507451	0.013055	0.019111
Cartesian opt. stratified MC	110.407590	0.013947	0.018577
Spherical prop. stratified MC	110.655165	0.011736	0.037203
Spherical opt. stratified MC	110.328969	0.014649	0.029047
	I_3		
	Value	Relative error	Variance
True value	12.671942		
Training data	13.640000	0.076394	528.528929
Crude Monte Carlo	12.674900	0.000233	53.330074
Cartesian prop. stratified MC	12.561556	0.008711	15.691618
Cartesian opt. stratified MC	12.613713	0.004595	7.585851
Spherical prop. stratified MC	12.800580	0.010151	13.139814
Spherical opt. stratified MC	12.614430	0.004539	7.312766

Table 5.14: Results of experiments using the Real NVP model for the art2d dataset. All columns except 'True value' represent estimators based on the given samples.

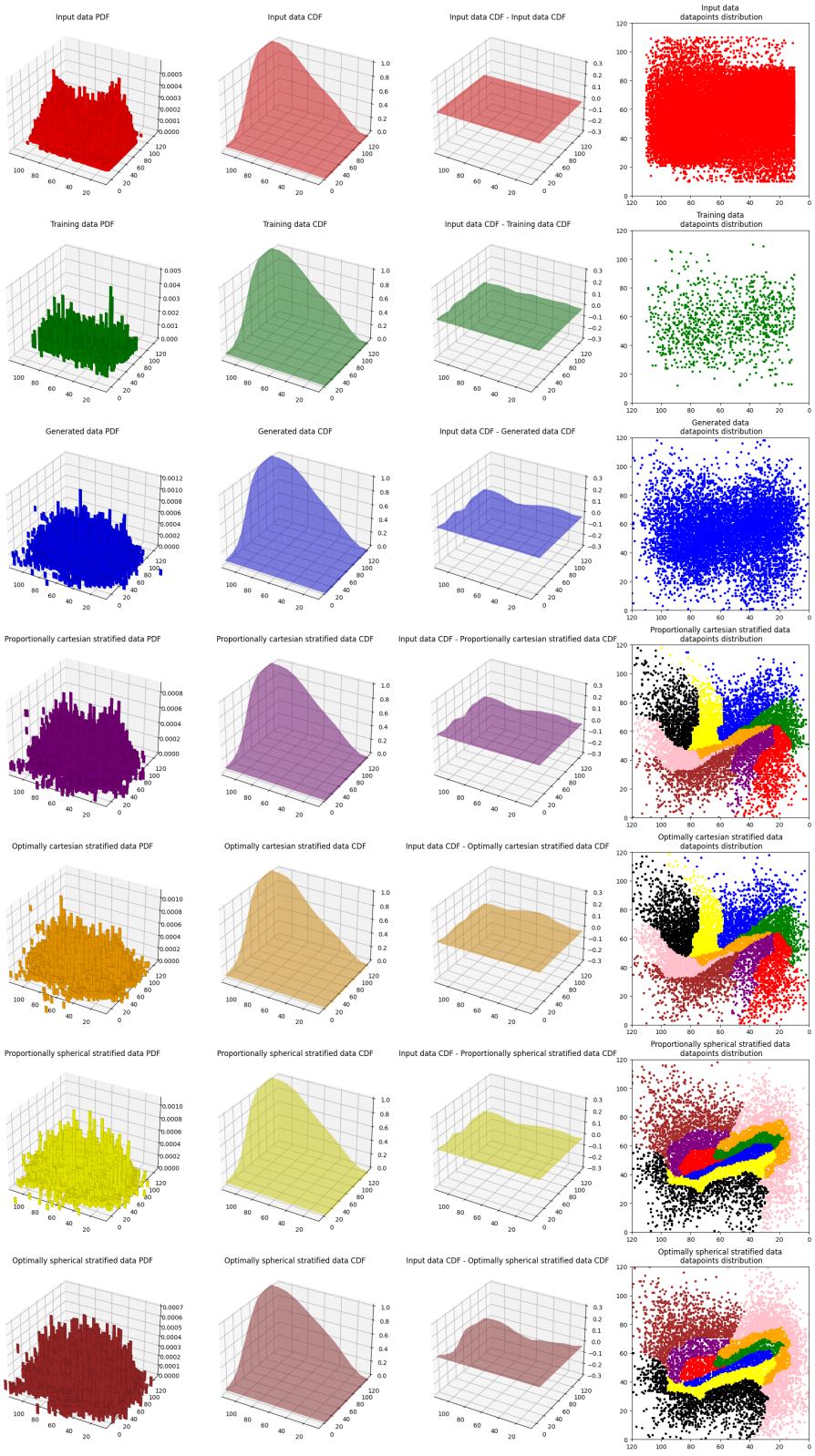


Figure 5.15: Plots of all generated data for art2d dataset with the Real NVP model.

Chapter 6

Conclusions

We tested discrete flow models NICE and Real NVP with five very different datasets. Apart from the humidity dataset, for which we did not use Real NVP, on all the other data the best result of NICE was outperformed by the best result of Real NVP. The biggest difference was in the artificial, two-dimensional dataset, where NICE failed to learn any significant properties of the data, meanwhile Real NVP performed better than the data it was learning from. Those results show that, at least for those datasets, Real NVP is more powerful than NICE. Adding multiplication to the coupling layer in (2.5) seems to enhance the learning capabilities of the model or at least make it learn much faster.

The stratification also proved to be effective, not only in reducing the variance by several orders of magnitude, but also in improving the quality of estimation. Stratified estimators almost always outperformed the Crude Monte Carlo estimator. Optimal estimators do not seem to have an advantage in the estimation quality over the proportional estimators, but they are worth using as they have significantly smaller variance. Their disadvantage is that they tend to produce plots that do not resemble the original data, even though they behave similarly according to the calculated statistics. That was the case with the temperature and humidity datasets, as the optimal cartesian stratification tends to allocate high budget to the endpoints of the distribution, as it grows rapidly at those intervals.

In the two-dimensional distributions, the cartesian stratification performed usually slightly better than the spherical stratification. The cartesian stratification is additionally much easier to implement in many dimensions, whereas the spherical stratification gets complicated with three or more dimensions.

One way to improve the results produced here is to find a better way of allocating the probabilities to strata, which were distributed equally in all our experiments. That can solve some small problems, like the shape of the plot for optimal allocation, but it could possibly also improve the overall quality of the generated samples.

Bibliography

[1] Frederik Michel Dekking, Cornelis Kraaikamp, Hendrik Paul Lopuhaä, and Ludolf Erwin Meester. *A Modern Introduction to Probability and Statistics*. Springer Texts in Statistics. Springer London, 2005.

[2] Laurent Dinh, David Krueger, and Yoshua Bengio. NICE: non-linear independent components estimation. In *3rd International Conference on Learning Representations, ICLR*, 2015.

[3] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real NVP. In *5th International Conference on Learning Representations, ICLR*, 2017.

[4] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems, NeurIPS*, pages 2672–2680, 2014.

[5] Will Grathwohl, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. FFJORD: free-form continuous dynamics for scalable reversible generative models. In *7th International Conference on Learning Representations, ICLR*, 2019.

[6] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR*, 2015.

[8] Diederik P. Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems, NeurIPS*, pages 10236–10245, 2018.

[9] Paweł Lorek, Rafał Nowak, Rafał Topolnicki, Tomasz Trzciński, and Maciej Zięba. Reducing estimation uncertainty using normalizing flows and stratification. Unpublished material.

[10] Neal Noah Madras. *Lectures on Monte Carlo Methods*. Fields Institute for Research in Mathematical Sciences Toronto: Fields Institute monographs. American Mathematical Society, 2002.

[11] George Marsaglia. Choosing a Point from the Surface of a Sphere. *The Annals of Mathematical Statistics*, 43(2):645 – 646, 1972.

[12] Per Pettersson and Sebastian Krumscheid. Adaptive stratified sampling for non-smooth problems. *CoRR*, abs/2107.01355, 2021.

[13] Aäron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. In *Proceedings of the 33nd International Conference on Machine Learning, ICML*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1747–1756, 2016.