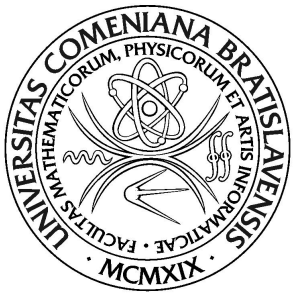

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Application of Boltzmann generator to study of structural rearrangements in small Lennard-Jones cluster

Diploma thesis

Bc. Andrej Uhliarik

Study program: Solid State Physics
Field of study: Physics

Supervisor: **prof. Ing. Roman Martoňák, DrSc.**
Research Group for Computational Materials Science
Department of Experimental Physics

Bratislava 2021

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Andrej Uhliarik
Study programme:	Solid State Physics (Single degree study, master II. deg., full time form)
Field of Study:	Physics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Application of Boltzmann generator to study of structural rearrangements in small Lennard-Jones cluster

Annotation: The work will explore the applicability of Boltzmann generator to study of structural rearrangements in small Lennard-Jones cluster.

Aim: The goal of the work is to explore the applicability of the recently proposed approach called Boltzmann generator [Noe2019] to study of structural rearrangements in small Lennard-Jones cluster. The method is based on finding a neural-network mapping from the configurational space x to a new so-called latent space z where the potential energy surface (PES) becomes simpler and much less rough. By sampling from the latent space it is possible to generate statistically uncorrelated configurations from the Boltzmann distribution, bypassing the problem of rare events. We plan to apply the scheme to the well-known problem of structural rearrangements in 2D Lennard-Jones cluster with 7 atoms. We will use the freely available library Tensorflow [tensorflow].

Literature: [Noe2019] Frank Noé, Simon Olsson, Jonas Köhler, Hao Wu, Science 365, 1001 (2019)
[tensorflow] www.tensorflow.org

Keywords: Boltzmann generator, machine learning, neural networks, Lennard-Jones cluster

Supervisor:	prof. Ing. Roman Martoňák, DrSc.
Department:	FMFI.KEF - Department of Experimental Physics
Head of department:	prof. Dr. Štefan Matejčík, DrSc.
Assigned:	21.11.2019
Approved:	22.11.2019

prof. RNDr. Peter Kúš, DrSc.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Andrej Uhliarik
Študijný program: fyzika tuhých látok (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: fyzika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Application of Boltzmann generator to study of structural rearrangements in small Lennard-Jones cluster
Aplikácia Boltzmannovho generátora na štúdium štruktúrnych preskupení v malom Lennard-Jonesovom klastri

Anotácia: Cieľom práce je preskúmať použiteľnosť Boltzmannovho generátora na štúdium štruktúrnych preskupení v malom Lennard-Jonesovom klastri.

Cieľ: Cieľom práce je preskúmať použiteľnosť nedávno navrhnutého prístupu nazvaného Boltzmannov generátor [Noe2019] na skúmanie štruktúrnych preskupení v malom Lennard-Jonesovom klastri. Metóda je založená na nájdení zobrazenia pomocou neurónovej siete (NN) z konfiguračného priestoru x do tzv. latentného priestoru z, v ktorom je povrch potenciálnej energie (PES) jednoduchší a podstatne menej drsný. Vzorkovaním z latentného priestoru je možné generovať štatisticky nekorelované konfigurácie z Boltzmannovho rozdelenia, čím sa vyhneme problému zriedkavých udalostí. Metódu plánujeme aplikovať na známy problém štruktúrnych preskupení v 2D Lennard-Jonesovom klastri so 7 atómami. Použijeme voľne dostupnú knižnicu Tensorflow [tensorflow].

Literatúra: [Noe2019] Frank Noé, Simon Olsson, Jonas Köhler, Hao Wu, Science 365, 1001 (2019)
[tensorflow] www.tensorflow.org

Kľúčové slová: Boltzmannov generátor, strojové učenie, neurónové siete, Lennard-Jonesov klaster

Vedúci: prof. Ing. Roman Martoňák, DrSc.
Katedra: FMFI.KEF - Katedra experimentálnej fyziky
Vedúci katedry: prof. Dr. Štefan Matejčík, DrSc.

Dátum zadania: 21.11.2019

Dátum schválenia: 22.11.2019 prof. RNDr. Peter Kúš, DrSc.
garant študijného programu

.....
študent

.....
vedúci práce

Acknowledgments

I am very grateful to my supervisor prof. Martoňák for his guidance during my studies, his trust and all the valuable discussions we had and advice he gave me. I would also like to thank my fellow students of solid state physics for the superb time spent together. I am especially thankful to Ondrej Bilý who has been my closest companion throughout the studies at the FMFI.

I extend gratitude to my family for their support and to my dear fiancée Paťka for all her love, understanding and courage to be by my side for all days to come.

Abstract

In the present thesis, we examine a new method of enhanced sampling called the Boltzmann generator, which was introduced in 2019. Unlike other methods, this one is based on machine learning, in particular, it uses a deep generative model to sample the Boltzmann (canonical) distribution. In the first part of the thesis, we discuss the theoretical background of the method. We have refactored the original code and adapted it to TensorFlow 2.0. We replicated results in the original article and examined the effect of changed solvent density on the solvated bistable dimer problem. Finally, we used the Boltzmann generator for a completely new system and studied rearrangements of a small Lennard-Jones cluster in two dimensions. We were able to train the Boltzmann generator for this system and using it we obtained transition (rearrangement) paths that correspond to the ones described in the existing literature.

Abstrakt

V tejto práci skúmame novú metódu na vylepšenie vzorkovania konfiguračného priestoru, ktorá sa nazýva Boltzmannov generátor. Táto metóda bola vynájdená v roku 2019. Na rozdiel od iných známych metód je založená na strojovom učení, využíva konkrétne hlboký generatívny model na vzorkovanie Boltzmannovho (kanonického) rozdelenia. V prvej časti práce sa zaoberáme teoretickými základmi tejto metódy. Prepracovali sme kód, ktorý bol použitý autormi pôvodného článku, a prispôsobili sme ho na použitie knižnice TensorFlow s novou verziou 2.0. Zreprodukovali sme výsledky zmieneného článku a preskúmali sme účinok zmeny hustoty rozpúšťadla na systém dvojstabilného diméru v rozpúšťadle. Nakoniec sme použili Boltzmannov generátor na úplne nový systém a skúmali sme preusporiadania Lennard-Jonesovho klastra v dvoch dimenziách. Podarilo sa nám natrénovať Boltzmannov generátor na tento systém a s jeho použitím sme zreprodukovali spôsoby preusporiadania klastra, ktoré sú opísané v literatúre.

Contents

Introduction

In 1929 Paul Dirac, one of the greatest physicists of that period, declared [1]:

> The underlying physical laws necessary for the mathematical theory of a large part of physics and the whole of chemistry are thus completely known, and the difficulty is only that the exact application of these laws leads to equations much too complicated to be soluble. It therefore becomes desirable that approximate practical methods of applying quantum mechanics should be developed, which can lead to an explanation of the main features of complex atomic systems without too much computation.

Even almost a hundred years later, this quote still represents the state of the art in the solid state physics. We are able to write down sufficiently precise rules, given by quantum mechanics, which describe behavior of the macroscopic systems based on their microscopic constituents. However, we are not able to *solve* these equations and thus extract the information we want.

As we know from statistical mechanics, every measurable quantity A of a macroscopic system can be expressed as an expectation value of the function a of the system microscopic state:

$$A = \mathbb{E}[a] = \int a(\mathbf{x}, \mathbf{k}) p(\mathbf{x}, \mathbf{k}) \; d\mathbf{x} d\mathbf{k}.$$

where $\mathbf{x}$ is the vector containing all system generalized coordinates, $\mathbf{k}$ contains all their conjugate momenta and $p(\mathbf{x}, \mathbf{k})$ is the probability with which the microstate is realized in the relevant ensemble. For example, for canonical ensemble, p denotes the microstate probability in the Boltzmann distribution.

While the momentum-related part can be almost always easily calculated, this does not hold for the configuration-related part of the equation. For the NVT (canonical) ensemble, the probability $p(\mathbf{x})$ is given by the Boltzmann distribution

$$p(\mathbf{x}) = \frac{e^{-U(\mathbf{x})/k_B T}}{Z} \qquad Z = \int e^{-U(\mathbf{x})/k_B T} d\mathbf{x}$$

where $U(\mathbf{x})$ is the potential energy of the configuration, k_B is the Boltzmann constant, T is the system temperature and Z is the partition function. The first problem is the precise calculation of $U(\mathbf{x})$ requiring solving a complicated many-body Schroedinger equation. Nevertheless, we are going to ignore this obstacle in the thesis and assume $U(\mathbf{x})$ to be known. Instead, we are going to focus on another problem which is related to sampling $p(\mathbf{x})$. The complication is that the hypersurface of the potential energy U for dense systems, like liquids or solids, is *very rough* – numerous local minima are separated by very high barriers. As a consequence, the Boltzmann distribution is extremely complicated.

Therefore, sampling the Boltzmann distribution is a very difficult task. The basic methods such as Molecular dynamics or Markov chain Monte Carlo generally sample the configuration space only slowly and need a long simulation time to escape the local minima of the potential energy. Therefore, numerous techniques of *enhanced sampling* are used.

We discuss some of them together with the problem of configuration-space sampling in section 1.

One such method was introduced in 2019 and is named *the Boltzmann generator* (BG) [2]. It is based on a subset of machine learning called *deep learning*. The Boltzmann generator uses a deep neural network which learns sampling of the Boltzmann distribution during the training process. This neural network represents the heart of the BG. Because of this, we provide a brief introduction into deep learning in section 2.

The BG learns an *invertible* mapping F from the real to so-called *latent* space where samples from the Botzmann distribution are distributed according to a simple normal distribution. After that, we can sample the Boltzmann distribution by sampling the normal one in the latent space and mapping the samples back to the real (configuration) space.

The BG is mainly trained using two methods:

- *Training by example:* we show the BG a set of samples from the Boltzmann distribution and teach it to sample similar ones. This set can be obtained, for example, with a Monte Carlo or an isothermal Molecular dynamics simulation.

- *Training by energy:* we sample the simple latent probability distribution, transform samples with F^{-1} and train the BG so that sample probabilities correspond to the ones given by the Boltzmann distribution (which are defined by sample energies).

We discuss the idea of the BG together with its theoretical base from the perspective of statistical mechanics and mathematical statistics in section 3.

We refactored the original code of the BGs and made some improvements to it. Our implementation of the BGs is discussed in section 4. The next goal was to replicate results of the original article for some toy models – this is the main topic of section 5.

After we accomplished that, we applied the BGs to a new system comprised of a small Lennard-Jones cluster. We mainly studied the rearrangements of the cluster and various mechanisms of their realization, see section 6. Finally, a summary of the results as well as a look into the future of the method are given in the conclusions section.

1 Difficulties in sampling Boltzmann distribution

1.1 Basic principles

Statistical mechanics (SM) provides us with tools that help us determine certain characteristics of a macroscopic physical system, based on its microscopic constituents. Let us think about a system at a temperature T. In sense of the Hamiltonian mechanics, the microscopic state of the system can be described by a vector of *generalized coordinates* $\mathbf{x}$ together with a vector of corresponding *conjugate momenta* $\mathbf{k}$.[1] For example, for a system of particles, vectors $\mathbf{x}$ and $\mathbf{k}$ would contain all atom coordinates and momenta, respectively, i.e.:

$$\mathbf{x} = (x_{1x}, x_{1y}, x_{1z}, x_{2x}, \ldots, x_{Nz})^\top \quad \mathbf{k} = (k_{1x}, k_{1y}, k_{1z}, k_{2x}, \ldots, k_{Nz})^\top, \tag{1}$$

where N is the number of particles in the system, x_{ij} denotes the $j \in \{x, y, z\}$ coordinate of the i-th particle and k_{ij} denotes the momentum of i-th particle in the direction of $j \in \{x, y, z\}$ axis.

All measurable system quantities A can be calculated as expectation values of corresponding functions $a(\mathbf{x}, \mathbf{k})$ of the system microscopic state:

$$A = \mathbb{E}[a] = \int a(\mathbf{x}, \mathbf{k}) p(\mathbf{x}, \mathbf{k}) \; d\mathbf{x} d\mathbf{k}, \tag{2}$$

where $p(\mathbf{x}, \mathbf{k})$ is the probability with which the microscopic state (*microstate*) is realized. For example, in the kinetic theory of gases, temperature is tied to the average kinetic energy $E_k(\mathbf{k})$. In the canonical ensemble, where a temperature T is fixed together with number of particles N and a system volume V, the probability of finding a system in its microstate $(\mathbf{x}, \mathbf{k})$ is given by the well-known Boltzmann distribution:

$$p(\mathbf{x}, \mathbf{k}) = \frac{1}{\tilde{Z}} e^{-\beta E(\mathbf{x}, \mathbf{k})}, \tag{3}$$

where $\beta = 1/k_B T$ is the inverse of product of temperature and the Boltzmann constant, $E(\mathbf{x}, \mathbf{k})$ is the energy of state $(\mathbf{x}, \mathbf{k})$ and $\tilde{Z}$ is the canonical partition function given by

$$\tilde{Z} = \int e^{-\beta E(\mathbf{x}, \mathbf{k})} \; d\mathbf{x} d\mathbf{k}, \tag{4}$$

where the integral is calculated over the whole available phase space.

The total energy of the system $E(\mathbf{x}, \mathbf{k})$ is given by the sum of potential energy E_p and kinetic energy E_k:

$$E(\mathbf{x}, \mathbf{k}) = E_p(\mathbf{x}) + E_k(\mathbf{k}), \tag{5}$$

For brevity, from now on we will join two indices of elements in the vectors $\mathbf{x}$ and $\mathbf{k}$, namely the number of the particle and axis, into a single one (x_i/k_i). Kinetic energy can be almost always expressed as a sum of per-particle kinetic energies. In other words,

[1] In the Hamiltonian mechanics, generalized coordinates are usually denoted as q_i and corresponding momenta as p_i. We use different notation here as we keep p reserved for the probabilities.

there are no products $k_i k_j$ where $i \neq j$ in the formula for $E_k(\mathbf{k})$. Moreover, for the kinetic energy quadratic in k_i, the momentum-related part of the $\tilde{Z}$ can be easily calculated

$$\begin{aligned}\tilde{Z} &= \int e^{-\beta(E_p(\mathbf{x})+E_k(\mathbf{k}))} d\mathbf{x} d\mathbf{k} = \int e^{-\beta E_p(\mathbf{x})} e^{-\beta E_k(\mathbf{k})} d\mathbf{x} d\mathbf{k} \\ &= \int e^{-\beta E_p(\mathbf{x})} d\mathbf{x} \int e^{-\beta E_k(\mathbf{k})} d\mathbf{k} = Z \int \exp\left(-\beta \sum_i k_i^2/2m_i\right) d\mathbf{k} \\ &= Z \prod_i \int \exp(-\beta k_i^2/2m_i) dk_i = Z \prod_i \sqrt{\frac{2\pi m_i}{\beta}}.\end{aligned}$$

Because of this, calculations of momentum-related quantities $a(\mathbf{k})$ are quite simple – when eq. (3) is used in the eq. (2), the numerator of the $p(\mathbf{k})$ can be factorized into a product of exponential functions, a formula for the denominator ($\tilde{Z}$) was just given and the parts corresponding to the vector of positions $\mathbf{x}$ simply cancel out.

On the other hand, since the formula for the *potential* energy $E_p(\mathbf{x}) \equiv U(\mathbf{x})$ of the configuration $\mathbf{x}$ is *much* more complicated, calculations of coordinate-related quantities are in general very complex. In the remainder of the thesis, we are going to deal only with this kind of problem.

Thus, motivated by the previous statements, we can ignore the momentum in eq. (2) and write:

$$A = \mathbb{E}[a] = \int a(\mathbf{x}) p(\mathbf{x}) d\mathbf{x}, \tag{6}$$

with the Boltzmann distribution given by

$$p(\mathbf{x}) = \frac{1}{Z} e^{-\beta U(\mathbf{x})} \qquad Z = \int e^{-\beta U(\mathbf{x})} d\mathbf{x}. \tag{7}$$

In fact, expectation values of some key properties are everything we want to know about a macroscopic system. For very large systems, like the objects with which people interact, we tend to identify these expectation values with the "true" values of the properties. Our question can be for instance, what is the probability of a protein being folded at a given temperature? Simple calculation of such expectation values would mean significant progress in many areas of modern science – we would be able to design new medicament molecules for drugs *in silico*, design new materials with desired qualities, etc.

However, for real systems, there are two difficulties. The first one comes with evaluating the potential energy function $U(\mathbf{x})$, as its exact computation for complex molecular systems requires solving the Schroedinger equation for nuclei and electrons. The surface of the function $U(\mathbf{x})$ in the configuration space $\mathbf{x}$ is called the *potential energy surface* (PES). There are numerous methods how to approximate PES using the Born-Oppenheimer approximation, which employ various approaches. These methods are also known as *ab initio* calculations since their input does not contain empirical information.

For instance, there is one such numerical method called *density-functional theory* (DFT) [5], which is, however, similarly to other *ab initio* calculations, computationally quite expensive. Therefore, mostly semi-empirical force fields are used, which are valid only for a small group of systems. From this point, we are not going to discuss this problem and we will consider $U(\mathbf{x})$ to be known throughout the thesis.

The second issue is that even for a known PES $U(\mathbf{x})$, calculations of expectation values (6) are difficult to evaluate. This is caused by the fact that probable configurations with high $p(\mathbf{x})$ constitute only a very small part of the configuration space $\mathbb{R}^n$ of dense systems. The vast majority of configurations $\mathbf{x} \in \mathbb{R}^n$ have very high energies and thus are realized with a very low probability.

Another difficulty lies in the partition function Z which takes the role of a normalization constant in the Boltzmann distribution. It is integral over the whole configuration space. For macroscopic systems, this has enormous dimension – of order $\mathcal{O}(10^{23})$ and more. Evaluating this kind of integral is far beyond our abilities.

Then, the question that remains is, how to deal with these issues? The most straightforward approach to calculate the expectation value (6), would be to construct a high-dimensional periodical mesh in the configuration space and try to approximate (6) with:

$$A = \mathbb{E}[a] \approx \frac{1}{N} \sum_{\mathbf{x}_i \in \text{mesh}} a(\mathbf{x}_i) p(\mathbf{x}_i), \tag{8}$$

where N is the number of points in the mesh. However, there are astronomically many ways how to organize atoms of the system and thus also possible configurations $\mathbf{x}$. Therefore, we are not able to calculate sums like this one. Even if we tried to perform such a calculation, the proportion of probable configurations, which play the most important role in the *true* value of A, would be close to zero in our mesh.

The solution would be to sample configurations from the distribution $p(\mathbf{x})$ and estimate A by

$$A = \mathbb{E}[a] \approx \frac{1}{N} \sum_{\mathbf{x}_i \sim p(\mathbf{x})} a(\mathbf{x}_i), \tag{9}$$

where N is the number of sampled configurations $\mathbf{x}_i$. If we could sample the Boltzmann distribution, we would be done. However, more frequently we are able to sample only some distribution $q(\mathbf{x})$ that is similar to the Boltzmann distribution. We can write

$$A = \mathbb{E}[a] = \int a(\mathbf{x}) p(\mathbf{x}) d\mathbf{x} = \int a(\mathbf{x}) \frac{p(\mathbf{x})}{q(\mathbf{x})} q(\mathbf{x}) d\mathbf{x}. \tag{10}$$

This expectation value can be approximated by

$$\mathbb{E}[a] \approx \frac{1}{N} \sum_{\mathbf{x}_i \sim q(\mathbf{x})} \frac{a(\mathbf{x}_i) p(\mathbf{x}_i)}{q(\mathbf{x}_i)}, \tag{11}$$

where samples $\mathbf{x}_i$ are sampled from the distribution $q(\mathbf{x})$ resembling the $p(\mathbf{x})$. This algorithm is called the *importance sampling*, as we sample configurations from the important regions more frequently than from the unimportant ones.

However, there is still one difficulty present in the last equation – we cannot calculate $p(\mathbf{x})$ for we do not know the partition function Z. Therefore, let us generalize the algorithm. Imagine that we are able to sample the distribution $q(\mathbf{x})$, but when given the sample $\mathbf{x}_i$, we are able to calculate $p(\mathbf{x}_i)$ (or $q(\mathbf{x}_i)$, or both) only up to a constant factor C_p (C_q). In other words, we are able to calculate $p_u(\mathbf{x})$ ($q_u(\mathbf{x})$, where index u stands for unnormalized) but not the $p(\mathbf{x}) = C_p p_u(\mathbf{x})$ ($q(\mathbf{x}) = C_q q_u(\mathbf{x})$).

When we use this in the eq. (10), we get:

$$\mathbb{E}[a] = \int a(\mathbf{x}) \frac{C_p p_u(\mathbf{x})}{C_q q_u(\mathbf{x})} q(\mathbf{x}) d\mathbf{x},$$

which can be divided by a smart one, which uses the fact that $p(\mathbf{x})$ is normalized ($\int p(\mathbf{x})d\mathbf{x} = 1$):

$$\begin{aligned}
\mathbb{E}[a] &= \int a(\mathbf{x}) \frac{C_p p_u(\mathbf{x})}{C_q q_u(\mathbf{x})} q(\mathbf{x}) d\mathbf{x} \Bigg/ \int \frac{C_p p_u(\mathbf{x})}{C_q q_u(\mathbf{x})} q(\mathbf{x}) d\mathbf{x} \\
&= \frac{C_p}{C_q} \int a(\mathbf{x}) \frac{p_u(\mathbf{x})}{q_u(\mathbf{x})} q(\mathbf{x}) d\mathbf{x} \Bigg/ \frac{C_p}{C_q} \int \frac{p_u(\mathbf{x})}{q_u(\mathbf{x})} q(\mathbf{x}) d\mathbf{x} \\
&= \int a(\mathbf{x}) \frac{p_u(\mathbf{x})}{q_u(\mathbf{x})} q(\mathbf{x}) d\mathbf{x} \Bigg/ \int \frac{p_u(\mathbf{x})}{q_u(\mathbf{x})} q(\mathbf{x}) d\mathbf{x} \,.
\end{aligned}$$

Ratio $p_u(\mathbf{x})/q_u(\mathbf{x})$ can be denoted as $w(\mathbf{x})$ as it takes the role of a statistical weight:

$$\mathbb{E}[a] = \int a(\mathbf{x}) w(\mathbf{x}) q(\mathbf{x}) d\mathbf{x} \Bigg/ \int w(\mathbf{x}) q(\mathbf{x}) d\mathbf{x} \tag{12}$$

which can be approximated by

$$\mathbb{E}[a] \approx \frac{\sum_i a(\mathbf{x}_i) w(\mathbf{x}_i)}{\sum_i w(\mathbf{x}_i)}, \tag{13}$$

where samples $\mathbf{x}_i$ are sampled from the distribution $q(\mathbf{x})$ and factor $1/N$ from the numerator and the denominator are canceled out. This is called the *self-normalizing importance sampling*.

At this point, the only thing we miss is a way how to sample distribution $q(\mathbf{x})$ similar to the Boltzmann distribution.

1.2 Molecular dynamics and Markov chain Monte Carlo

As we already explained, efficient sampling of the Boltzmann distribution is of high practical significance. However, there are only a few distributions for sampling which we can construct numerical algorithms. This includes, for example, a uniform distribution on an interval or a multivariate normal distribution.

Unfortunately, the Boltzmann distributions are usually very complicated. The source of this complexity is the intricacy of the potential-energy surface $U(\mathbf{x})$. For dense systems such as liquids or solids PES consists of numerous minima of various depths separated by very high barriers that arise for example when some particles are too close to each other. While the equilibrium properties are determined by the minima, barriers are deciding for the kinetics (transitions). One must keep in mind that the area occupied by the minima is very low compared to the whole configuration space.

Therefore, generating *independent* samples from the Boltzmann distribution in one pass ("one shot") is unfeasible. This means that we are not able to construct a machine that would produce independent configuration distributed according to (7) each time we asked it to do so.

In order to overcome this problem, a few methods have been invented. These do not sample from the Boltzmann distribution in "one shot" but are trajectory-based instead. They produce a trajectory where consecutive configurations are separated just a little in the configuration space and, thus, they are correlated. Only if we take each N-th sample ($N \gg 1$) into account, we can consider them to be independent. Nevertheless, it can be shown that these methods visit individual configurations with a probability (7) in a limit of an infinite number of steps (trajectory length). We present here the two most common methods. The following text is based on [6, 7].

Markov chain Monte Carlo

The first method is Markov Chain Monte Carlo (MCMC) introduced by Metropolis et al. in 1953 [8, 9]. Discrete steps are used here rather than a continuous time. During each step the current configuration $\mathbf{x}$ is changed a little so that $\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \Delta\mathbf{x}$, where $\Delta\mathbf{x}$ is randomly chosen. This new configuration is called a *trial* or *proposal* move. The trial move is accepted with a probability of:

$$P_{\text{acc}} = \min\left\{1, \exp\left(-\frac{U(\mathbf{x}_{\text{new}}) - U(\mathbf{x}_{\text{old}})}{k_B T}\right)\right\}, \tag{14}$$

i.e. configuration with lower energy is always accepted, while configuration with higher energy is accepted with a probability given by the Boltzmann factor.

This algorithm can be used to compute expectation values, as in eq. (9):

$$\mathbb{E}[a] \approx \frac{1}{N}\sum_{i=1}^{N} a(\mathbf{x}_i), \tag{15}$$

where N is the number of realized steps and $\mathbf{x}_i$ is the configuration after i-th step. One must keep in mind that even if $\mathbf{x}_{\text{new}}$ was not accepted, the step has been executed – in such case the trajectory $\{\mathbf{x}_i\}_{i=1}^{N}$ will contain the configuration $\mathbf{x}_{\text{old}}$ twice in a row.

There are various ways how to construct the trial moves. One of them is, for instance, to move *one* particle to some point in a cube around its original position:

$$\begin{aligned} x_{ix}^{\text{new}} &= x_{ix}^{\text{old}} + (\xi_x - 0.5)\Delta \\ x_{iy}^{\text{new}} &= x_{iy}^{\text{old}} + (\xi_y - 0.5)\Delta \\ x_{iz}^{\text{new}} &= x_{iz}^{\text{old}} + (\xi_z - 0.5)\Delta, \end{aligned}$$

where i is the index of the particle, ξ_x, ξ_y, ξ_z are random numbers from the interval $[0; 1]$ and Δ, which is the length of the cube side, is called the *step size* as it controls the length of a typical move. Since this method of constructing the proposal moves does change position of only a single particle at the time, we usually use the term "step" for a pass over all system particles.

Another method for trial-move construction is to sample the new configuration from an isotropic multivariate normal distribution with the origin in the old configuration:

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \Delta\mathbf{x} \qquad \Delta\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \sigma^2 I_n). \tag{16}$$

Here, σ has the role of the step size.

Molecular dynamics

The second method is Molecular dynamics (MD). The idea here is very simple – to numerically integrate Newton equations of motion for particles (atoms/molecules). A convenient choice is, for example, to use the *velocity Verlet* algorithm [10, 11]:

$$\mathbf{r}_i(t+\Delta t) = \mathbf{r}_i(t) + \Delta t \mathbf{v}_i(t) + \frac{(\Delta t)^2}{2m_i}\mathbf{F}_i(t)$$
$$\mathbf{v}_i(t+\Delta t) = \mathbf{v}_i(t) + \frac{\Delta t}{2m_i}[\mathbf{F}_i(t) + \mathbf{F}_i(t+\Delta t)],$$

where $\mathbf{r}_i(t)$ and $\mathbf{v}_i(t)$ are position and velocity of the i-th particle, respectively, $\mathbf{F}_i(t)$ is the force acting on this particle, m_i is its mass and Δt is the time step. These equations instruct us on how to calculate time evolution of the initial configuration $\mathbf{x}_{\text{init}}$.

According to the *ergodic hypothesis*, if we assume the limit of an infinite time, a time average obtained by the MD is the same as an ensemble average in the microcanonical ensemble. This can be expressed as:

$$\mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})}[a] = \lim_{t\to\infty} \frac{1}{t}\int_0^t a(\mathbf{x}(t'))\mathrm{d}t', \tag{17}$$

where $a(\mathbf{x})$ is the quantity whose average is being calculated and $p(\mathbf{x})$ is the configuration probability in the microcanonical ensemble.

In order to sample the canonical ensemble (Boltzmann distribution), we have to implement one of so-called thermostats in our MD simulation which forces the system to change its $E_{\text{tot}} = E_{\text{kin}} + E_{\text{pot}}$ according to the Boltzmann distribution, see e.g. [7, pp. 140 – 158] or [12].

Problem of rare events

In nature, important events such as chemical reactions in chemistry, structural phase transitions in condensed matter physics, or protein folding in biophysics require coordinated movement of many particles in the system. This means that they are realized on a much longer time scale than the microscopic motion of the system.

From a theoretical point of view, this is caused by the fact that (first-order) phase transitions represent switching from one free-energy minimum (also called **metastable state**) to the other one and this requires crossing the free-energy barrier. Therefore, a certain time is needed so that there is such fluctuation present in the system which enables the crossing of the barrier. Because of this, we refer to this kind of transitions as **rare events**.

When running an MD simulation, we are integrating the equations of motion. This means that the integration time step Δt must be approximately one order of magnitude smaller than the period of oscillation of the fastest degree of freedom in the system. Therefore, observation of a single rare event may require $10^8 - 10^{12}$ time steps and may even become inaccessible to the simulation. This phenomenon is called the *time-scale gap* in the MD.

In the MCMC simulations, we often experience a similar problem. In principle, if we are able to construct long but smart proposal moves (with high acceptance ratio), we can

switch from one metastable state to the other easily. However, if we cannot construct such moves, the long step comes with a considerable probability of producing a high-energy configuration which has a very low probability of acceptance. As a result, the system tends to stay in the same (initial) configuration. Therefore, we have to use a short step size in the simulation and crossing a barrier requires, similarly to the MD, multiple coordinated movements.

To sum up, the main problem of these methods is a slow sampling of the configuration space and a long time that is needed to cross a free-energy barrier and move from one metastable state to another. A considerable effort was made in order to overcome this obstacle. We discuss some of the methods that are used to solve this problem in the next section.

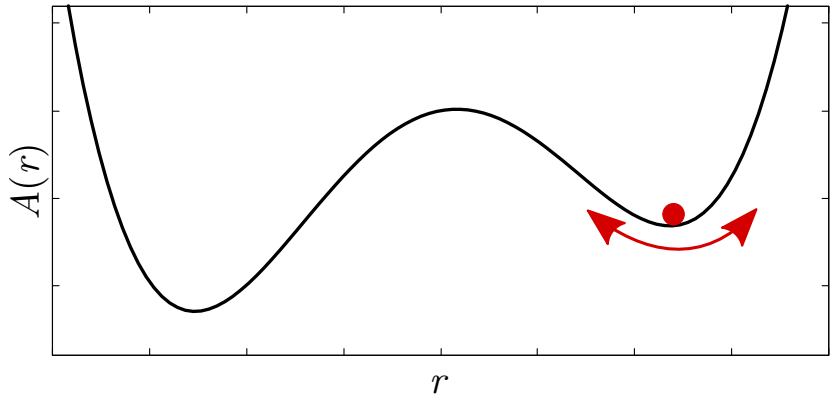

Figure 1: Configuration $\mathbf{x}$ represented by a red ball is in one of the two free energy minima. During both MCMC and MD simulations, $\mathbf{x}$ performs mostly only small oscillations near the minimum (red arrows). Transition to the second minimum requires coordination of many consecutive moves and thus occurs only rarely – such transitions are called *rare events*.

1.3 Enhanced sampling techniques

Numerous techniques have been invented, both for the MD and MCMC, which help us solve the rare-event problem and promote sampling outside of a single metastable state. One can even say that research of these *enhanced sampling* methods has become an area on its own. What these techniques have in common is that they use a non-physical element to force the system to leave a metastable state. They are especially used to compute free-energy landscapes along some kind of variable $r(\mathbf{x})$. These are called *collective variables* or *reaction coordinates*. They are usually scalars or vectors with a low dimensionality and depend on the positions of many atoms. They can also be used to describe the position of the system in some transitions (e.g. chemical reaction) A $\rightarrow$ B.

At this point, we are going to derive a formula for the free-energy profile (FEP) along a reaction coordinate r. From SM we know that for free energy A holds:

$$A = -k_B T \ln Z,$$

where Z is the partition function. Let us now divide this single state of the system into many sub-states labeled by value of the collective variable r, i.e. all microstates $\mathbf{x}$ with the

same value of $r(\mathbf{x})$ belong here. The landscape of free energy along a reaction coordinate r is then given by:

$$A(r) = -k_B T \ln Z(r) \qquad \text{where} \qquad Z(r) = \int \exp\left(-\beta U(\mathbf{x})\right) \delta(r(\mathbf{x}) - r) \mathrm{d}\mathbf{x}.$$

Therefore, adding smart zero using the partition function of the whole system Z, we get:

$$A(r) = -k_B T \ln Z(r) + k_B T \ln Z - k_B T \ln Z = -k_B T \ln \frac{Z(r)}{Z} + \text{const.},$$

and after noticing that $Z(r)/Z$ is probability $p(r)$ of the whole system to be in the state with value of the reaction coordinate $r(\mathbf{x}) = r$, we can write:

$$A(r) = -k_B T \ln p(r) + \text{const.}, \tag{18}$$

which is the desired formula for the free-energy profile.

There are many methods for enhanced sampling. However, for brevity, we are going to discuss only three of them. The first method is called the **Parallel tempering** or *replica exchange* MCMC sampling [13–16]. It is probably the simplest method of enhanced sampling. Multiple replicas ($i \in \{1, \ldots, N\}$) of the system run in parallel, each in its current state $\mathbf{x}_i$, but they have different temperatures T_i. After a given number of MCMC steps, states of two replicas are exchanged ($\mathbf{x}_i \leftrightarrow \mathbf{x}_j$) with a probability given by Metropolis criterion:

$$P_{\mathrm{PT}}(\mathbf{x}_i \leftrightarrow \mathbf{x}_j) = \min\left\{1, \frac{\exp\left(-\frac{E_j}{kT_i} - \frac{E_i}{kT_j}\right)}{\exp\left(-\frac{E_i}{kT_i} - \frac{E_j}{kT_j}\right)}\right\} = \min\left\{1, \exp\left[(E_i - E_j)\left(\frac{1}{kT_i} - \frac{1}{kT_j}\right)\right]\right\},$$

where $E_i \equiv E(\mathbf{x}_i)$. Since replica at a higher temperature is more probable to escape the local metastable state, this way, by exchanging configurations, the replica at a lower (physical) temperature is forced to sample states outside the local minimum of the free energy as well.

The second method is called the **Umbrella sampling** [17, 18] and it was the first one to be developed. Simulation is divided into separate "windows" ($i \in \{1, \ldots, N\}$), and for each of them an additional term called *bias potential* is added to the $U(\mathbf{x})$:

$$U^{(i)}(\mathbf{x}) = U(\mathbf{x}) + V^{(i)}_{\mathrm{bias}}\left(r(\mathbf{x}), r_0^{(i)}\right). \tag{19}$$

Bias potential is usually low at $r_0^{(i)}$ and raises when the collective variable moves away from this value. The most frequent choice has the quadratic form

$$V^{(i)}_{\mathrm{bias}} = \frac{1}{2} k_{\mathrm{bias}}\left(r(\mathbf{x}) - r_0^{(i)}\right)^2. \tag{20}$$

This way MD or MCMC simulation is biased towards sampling configurations with values of collective variable $r(\mathbf{x})$ close to the center $r_0^{(i)}$ of the current (i-th) window.

Typically, values $r_0^{(i)}$ are set equidistantly in some relevant interval $[r_{\min}; r_{\max}]$. Final configuration of the $(i-1)$-th window is used as the initial configuration for the i-th window

and there is some burn-in period in the simulation so that the system can accommodate (relax) itself to the new potential $U^{(i)}(\mathbf{x})$. Often, the umbrella-sampling simulation runs both forwards and backwards – after the simulation in the last window, the whole sequence of windows is repeated but now in the reverse order. This way $r_0^{(i)}$ is changed as follows: $r_{\min} \to N$ windows $\to r_{\max} \to N$ windows in the reverse order $\to r_{\min}$.

After the simulations are performed, results from individual windows have to be correspondingly merged to reconstruct the *unbiased* probability distribution $p(r)$. This can be done for instance with *Weighted histogram analysis method* (WHAM) [19, 20].

Most frequently, we want to use $p(r)$ for calculation of the free-energy profile along the collective variable r, see eq. (18). The same can also be done by applying the *Bennett acceptance ratio* (BAR) [21] to each two consecutive windows:

$$\Delta A_{ij} = A_j - A_i = -k_B T \ln \frac{Z_j}{Z_i} = k_B T \frac{\langle \exp[-\beta U^{(i)}(\mathbf{x})] \rangle_j}{\langle \exp[-\beta U^{(j)}(\mathbf{x})] \rangle_i}, \tag{21}$$

where $\langle \cdot \rangle_i$ stands for the average in the i-th window. This calculates the free-energy difference between these windows (states), which can then be interpreted as $\Delta A_{ij} = A\left(r_0^{(j)}\right) - A\left(r_0^{(i)}\right)$. Although this method is much simpler than WHAM, it is less accurate since it compares always only two windows. The more accurate generalization of this method, which calculates the whole free-energy profile at the same time, is the *multistate* Bennett acceptance ratio (MBAR) [22].

All of these methods require a convenient choice of the number of windows and size of the energy constant k_{bias}. If this constant is too low, the bias potential will not be strong enough to force the system to sample configurations with $r(\mathbf{x})$ close to the $r_0^{(i)}$ and the system will stay in one of the local minima instead. On the other side, if the constant is too high (or the number of windows too small), there will not be a sufficient overlap between the windows and results will be inaccurate.

The third method is called **Metadynamics** (MetaD) [23–26]. Similarly to the Umbrella sampling, a bias potential is added to the real potential:

$$U_{\text{MetaD}}(\mathbf{x}) = U(\mathbf{x}) + V^{\text{bias}}(r(\mathbf{x})). \tag{22}$$

In this case, $V(r)$ is adjusted *continuously* during the simulation:

$$V^{\text{bias}}_{t+\Delta t}(r) = V^{\text{bias}}_t(r) + w \exp\left[-\frac{(r - r(t))^2}{2\sigma^2}\right], \tag{23}$$

where w and σ are parameters of the bias potential. This way, the potential is *locally raised* for the current configuration which drives the system to leave it. When the system is moving inside the free-energy minimum, it gradually fills it with the bias potential and finally enters another metastable state.

When we achieve the point when the simulation can move between the metastable states freely, i.e. $p(r)$ is uniform and the minima are filled with the bias potential; from eq. (18) we can see that this implies

$$p(r) = \text{const.} \Rightarrow A_{\text{biased}}(r) = -\ln p(r) = \text{const.} = A(r) + V^{\text{bias}} \Rightarrow A(r) = -V^{\text{bias}} + \text{const.} \tag{24}$$

and thus we can calculate the FEP from the accumulated bias potential.

Note that all these methods utilize some collective variable $r(\mathbf{x})$. As it turns out, they are very sensitive to the choice of the $r(\mathbf{x})$ and very inaccurate results can be obtained by these techniques if the reaction coordinate does not characterize the transition appropriately.

In 2019 a completely new method for solving the sampling problem was proposed, called the *Boltzmann generator*, which is the main subject of the thesis. This technique does not resemble any of the methods known up to this point and is based on machine learning. Therefore, we are going to explain the basics of this field in the following section so that the reader has a sufficient understanding to comprehend the Boltzmann generators.

2 Deep learning

2.1 Machine learning

In order to better understand deep learning, we start with a discussion of a much broader field – *machine learning* (ML). There are many books that provide an introduction to this area; this section is mainly based on [27]. As suggested by its name, the key aspect of machine learning is that "a machine (computer) is able to learn". Tom Mitchell came with a rather formal but very accurate definition explaining what this means [28, p. 2]:

> A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

There are many tasks T which science, and in general people, try to solve using computers. Some of them are more suitable for ML algorithms (we also call them *models*) than others. For example, one could try to develop an ML model for which input would be the name of the user (e.g. James/Jane) and the task of the program would be to greet the user using "Hi James/Jane!". In the beginning, the program would print random output but with experience, it would learn to greet the user in the desired way. Naturally, it is substantially easier to code such a program without the use of ML, since the solution will have no more than a few lines of code in any modern programming language.

In general, traditional programs perform well when part of the reality which we are trying to describe by a program has the following properties:

1. its *state* is easy to describe and
2. there is a well-defined set of *rules* that apply to these states.

Since this may not be easy to grasp, let us illustrate these principles with two sample problems. The first problem, for which we could use a computer, is to win a chess game against a well-trained chess player. The second problem is to differentiate between dogs and cats in pictures taken by a digital camera.

"State" in the former problem can be easily described – full information consists of positions of $2 \times 16 = 32$ chess pieces on 2D chessboard, i.e. $32 \times 2 = 64$ integers from the interval $[1; 8]$. However, the situation in the latter problem is more complicated. If pictures have, for instance, size 200 x 200 pixels with color described by the RGB model, then each picture is represented by $200 \times 200 \times 3 = 120\,000$ integers from interval $[0; 255]$, which is by 4 orders of magnitude more than in the former case.

Regarding the "rules", we can see that chess pieces can move in a strictly defined way, which can be easily translated to a computer program. On the other hand, even an adult person may have to think for a while in order to define how a dog differs from a cat. Describing such rules not in natural human language but at the level of pixels might be considered downright unfeasible.

In other words, traditional computer programs can easily surpass people in tasks that require only a little knowledge about the world but lots of "monotonous" operations. However, these programs perform poorly when solving problems that require what we

call *intuition* – a large amount of information about the world around us that all humans have and take as granted but struggle to logically describe. This is a subset of problems that are suited for ML-based algorithms.

Problems that are nowadays solved by ML models are usually named by defining how the algorithm should process a given *example* (also called *datapoint*). By an example we commonly mean an n-dimensional vector $\mathbf{x} \in \mathbb{R}^n$ that is a set of *features* x_i. We refer to a set of all examples that are used in training (i.e. all examples the model can *experience*) as *training dataset*. Because of this, we want the program to learn mapping $y = f(\mathbf{x})$ where y is the desired output for the example $\mathbf{x}$.

Let us clarify this by listing some of the most common types of tasks solved by ML:

- **Classification** is a type of task where we want our algorithm to be able to classify datapoints to some well-defined groups. For example, we may want it to identify what animal is on the picture – either a cat or a dog, as we already explained above. In this case, our algorithm would be the mapping $f : \mathbb{R}^n \to \{1, 2\}$ where 1 stands for a dog and 2 for a cat. Another example of a classification problem is the recognition of hand-written digits. We may alter this task by requiring the algorithm to produce probability mass for all categories instead of outputting just the most probable category.

- **Regression** is asking the program to predict some numerical value y given the datapoint $\mathbf{x}$.[2] This means that we want our prediction $\hat{y}$ to be as close to *real* y as possible. Usually we want the prediction (estimator) $\hat{y}$ to have some properties, for instance in *linear* regression we are trying to find a vector of *weights* $\mathbf{w}$ and a so-called *bias* b so that we optimize $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$. In principle, regression is very similar to classification – they differ only by the type of the output and perspective from which we define the task.

- **Transcription** resides in extracting some text/number from the pictures. Common examples are reading house numbers from street view images [29] or extraction of a text from a scanned PDF file so that a user can search through it.

- **Sampling** – while the first three tasks have some similarities, this one is much more different. In sampling, we show a model set of examples and we want it to produce other examples that are similar to the ones we began with. This requires the model to learn the properties of the set of examples. For example, we could show the ML model some pictures of human faces in order to be able to produce other pictures, which would preferably not resemble those in the starting set but would still be considered credible by an uninvolved observer.

- **Density estimation** represents learning of the probability density function (PDF) $p(\mathbf{x})$ by whose sampling it is possible to produce the set of examples $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}$ present in the training dataset. In this case, the ML model has to learn where in the hyperspace $\mathbb{R}^n$ examples are likely to occur and where they are not.

[2] Of course, y does not have to be limited to just single number $y \in \mathbb{R}$. Instead it can be a vector $\mathbf{y} \in \mathbb{R}^m$.

Naturally, this list of typical tasks is far from being exhaustive. We present mainly those that are somehow important in the context of this thesis. For a more complete overview of tasks ordinarily solved by ML models see [27, pp. 98 – 101].

Furthermore, we are going to discuss the basic principles of machine learning. Please consider this to be an overview rather than a detailed explanation. Reading this should provide some insight for the reader before starting working with Boltzmann generators.

Basics of the ML can be, at the time of writing, best understood by taking part in one of the many available online ML courses for beginners. Answers to subtler questions can be found in review-style books and research articles.

Supervised vs. unsupervised machine learning

While reading previous lines, the reader may have noticed that, in principle, typical ML tasks can be divided into two categories:

1. either for each example $\mathbf{x}$ from the training dataset there is a known *label* or *target* y which we want our model to be able to determine from $\mathbf{x}$ – this is the case of classification (we know whether there is a dog or a cat at the picture), regression ($y^{(i)} = f(\mathbf{x}^{(i)})$ is known for all examples) or transcription (one can read house numbers or text in a file and thus prepare targets $y^{(i)}$ before training)

2. or such targets are not known (density estimation, where $y = p(\mathbf{x})$ is *a priori* unknown) or even do not exist in principle (sampling).

The former category is called supervised and the latter unsupervised learning. The naming originates from the view that by providing targets y, we are teaching the ML model what to do and therefore supervising its learning. On the other side, when we do not provide labels, the model must learn what to do "by itself".

When one refers to the training dataset in the context of supervised training, they refer to the set of pairs example-target $\{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{N}$. In the following paragraph, we will illustrate the basic principles of the ML on supervised learning as it is easier to understand. However, unsupervised learning is more important in the context of this thesis.

Evaluation of model performance

In order to teach a model, we must have a training dataset as well as a formulated *aim* of the learning process, since learning is never the *goal* of ML algorithm but only a *tool*. Therefore, we usually define a certain performance *metric* by which the model is assessed.

For classification, we may use *accuracy* as such metric, i.e. the proportion of examples for which the model correctly determines their category (label). Analogically, we could use error rate, which is just another formulation of the same quality. The error rate is an example of so-called 0-1 loss, as for a single example it can only acquire two values: 0 if the example is classified correctly and 1 if it's not.

For other tasks, such as regression, discrete 0-1 loss is not suitable. Instead, since both true targets y and predicted targets $\hat{y}$ are continuous, we need some kind of a continuous

performance metric. For example, we could use the mean squared error (MSE), which is defined as

$$\text{MSE} = \frac{1}{N}\sum_i (\hat{y}_i - y_i)^2, \tag{25}$$

where N is the number of assessed datapoints, $\hat{y}_i$ is the model's prediction of target for example $\mathbf{x}_i$ and y_i is the true value of its target.

While coming up with *some* way of measuring the model performance is usually quite straightforward, figuring out the metric that best describes our expectations of model behavior may be a complicated matter. The appropriate definition of the performance metric is very important as it turns out that it has a *substantial* influence on the training process and thus also on the model performance. Think, for example, about these questions:

- In case we are doing animal classification and our model classifies a dog as a shark, it is a serious mistake; but if a dog is classified as a wolf, it is much less serious. Should we take this into consideration?

- If our ML model should separate ordinary e-mails from spam, then the accuracy is probably not the best metric, since leaving spam e-mails among ordinary ones is less serious than when an expected e-mail ends up in spam (impact of these decisions on a user can differ a lot). This should probably be taken into account. But how many times is the latter error case worse than the former one? Is it three, or perhaps five times worse?

- In general, what should be penalized more? When a model is wrong more often but only by a little or when it is wrong rarely but mistakes are more considerable?

Since questions like these arise in all ML applications, choosing the best performance metric can take an appreciable amount of time/energy.

In most ML implementations the performance metric is defined in a way that we want it to be *minimized* and it is usually called *error* or *loss* and denoted J.

Underfitting and overfitting

In practice, we want our model to perform as well as possible in real-world conditions. Due to the very nature of ML applications, the training dataset can not contain *all relevant* examples $\mathbf{x}^{(i)}$ but only a portion of them. For example, it is not possible to include all pictures of dogs and cats into our training dataset, simply because this would represent an enormous amount of data and moreover, new pictures are taken every day.

This means that our model $\hat{y} = f(\mathbf{x})$ will have to make predictions for different inputs $\mathbf{x}$ than it experienced during the training. In other words, if we imagine hyperspace $\mathbb{R}^n$ containing the training dataset, the model does have to perform well not only in points $\mathbf{x}^{(i)}_{\text{train}}$, but also *between* them. This ability is called *generalization*.

Because of this, the aim of the training is not to optimize performance on the training dataset, but rather on previously *unseen* inputs. In order to accomplish it, available dataset $\{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{N}$ is typically divided into three groups:

- *Training* dataset is used for training of the ML model – it is seen by the model during the training phase and the model can learn from it.
- *Validation* dataset is used for estimating model performance on unseen inputs. It is seen by the model during the training, but the model does not learn from it. It may be, however, used in setting model parameters.[3]
- *Test* dataset is used for evaluation of model performance after the training. Ideally, it should contain all kinds of inputs the model may experience in real usage. The same naturally applies to other datasets as well. In literature, it is also sometimes used as a synonym for the validation dataset.

The training dataset is normally a few times larger than validation and test datasets. The general rule of thumb is to divide the whole dataset into these groups in the ratio 60:20:20.

During the training, we usually monitor training error (on training dataset) as well as validation error (on validation dataset, also called test/generalization error). Naturally, the model always performs better on the training dataset, from which it can learn, than on the validation dataset, where it has to generalize. Therefore, the training error is always lower than the test error. While a decrease in the former one is usually accompanied by a decrease in the latter one, there are some circumstances under which this may not be true. This is typically caused by the fact that the model somehow starts to memorize the training dataset and fails to interpolate between its datapoints. An effort for the best possible generalization can be translated to pursuing two goals [27, p. 109]:

1. Making the training error small.
2. Making the gap between training and test error small.

These two goals correspond to two central challenges in ML: *underfitting* and *overfitting*.[4] Underfitting means that the ML model is not able to have a sufficiently low training error. Overfitting means that the model is able to have a low train error but suffers from a too large gap between training and test error. We can control the model's tendency to underfit/overfit by adjusting its *capacity*. Capacity is the amount of complexity the model is able to cover/learn.

This is illustrated in fig. 2. Imagine that we want to fit some datapoints by a polynomial function. We want our fit $\hat{y} = f(x)$ to perform as well as possible for the *test* dataset, i.e. also on examples not available during the training. If we instead wanted the best possible performance on available datapoints, it would not be an ML problem but rather an *optimization* problem.

As can be seen, linear fit can not capture the complexity of the relationship between $x^{(i)}$ and $y^{(i)}$ and we are experiencing underfitting. The quadratic function does reproduce this relationship well and will also perform well on unseen inputs. A higher polynomial

[3] Specifically its *hyperparameters*, which are explained later.

[4] Challenge of balancing between underfitting and overfitting is also known as "bias-variance tradeoff". This name arises from a more mathematical view of the issue.

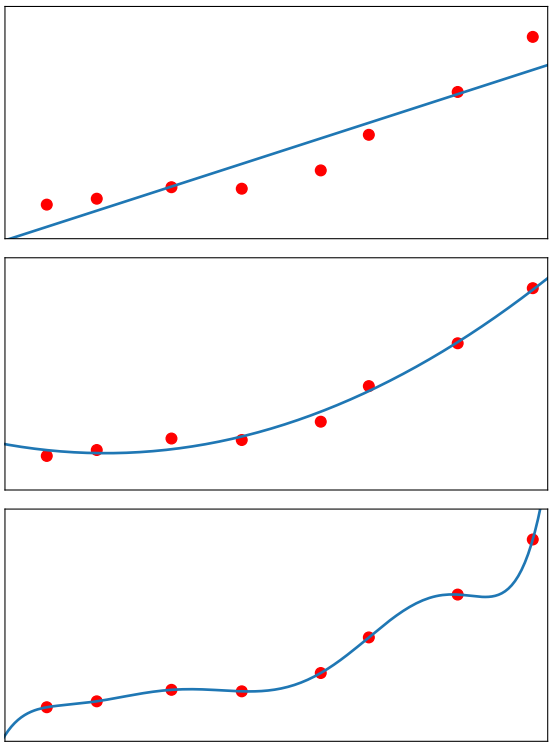

Figure 2: Fitting of datapints with polynomials of different capacity (blue lines). Data (red dots) have been produced by adding Gaussian noise to a quadratic function. *Top:* Fit by the linear function resulting in underfitting. *Center:* Fit by the quadratic function which best describes underlying function $y^{(i)} = f(x^{(i)})$. *Bottom:* Fit by the higher-order polynomial resulting in overfitting. Idea by [27, p. 111].

is able to exactly pass through all training datapoints, but this will result in overfitting as it will not perform well on unseen inputs, especially in the area of extrapolation.

Decreasing the model capacity is not the only way how to prevent overfitting. There is a whole class of such methods which is named *regularization* [27, p. 117]: "*Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.*" Regularization techniques include for example:

- *Weight decay* – model is penalized if its trainable weights get too high. This is done because models with lower weights are generally simpler. This way we can force the model to try and find simpler solutions which have a lower tendency to overfit. Recall the example of an ML model learning the linear regression. Weight decay could be applied by changing the loss function J to

  $$J_{\text{reg}} = J + \lambda \mathbf{w}^\top \mathbf{w},$$

  with which we are penalizing size of $\|\mathbf{w}\|^2$. With parameter λ we can control how large must the decrease in error J be so that it is worth increasing $\|\mathbf{w}\|$ for the model.

- *Dropout* – if the learned function f is composed of multiple sub-functions so that $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$, dropout means that randomly chosen part of information

is intentionally lost in the middle of calculating $f(\mathbf{x})$. This is done by setting some features of $\mathbf{x}$ to zero after applying $f^{(1)}$, $f^{(2)}$ or $f^{(3)}$.

- *Data augmentation* – simply scaling up the training dataset, either by including new examples or by doing some processing to the existing ones (e.g. zooming/rotating the images, slightly changing the colors, etc.).

- *Early stopping* – stopping the training at the moment when the validation error starts to grow, even if the training error still decreases.

(Hyper)parameters

When talking about parameters of the ML model, we distinguish between *parameters* and *hyperparameters*. Hyperparameters are properties of the model which are set "by the programmer" and, roughly speaking, which we do not intend to change during the training. On the other hand, parameters of the model are those variables that are trained, i.e. adjusted during the training.[5] Frequently, we use $\boldsymbol{\theta}$ to denote the vector of all model's parameters.

For example, in the fitting problem in fig. 2, we tried to fit the data using standard polynomial

$$f(x) = \sum_{i=0}^{N} a_i x^i.$$

Here the order of the polynomial N is a hyperparameter and $\{a_i\}_{i=0}^{N}$ is a set of parameters.

Maximum likelihood estimation

The last concept of ML we want to talk about is the most frequently used recipe for constructing a loss function J. Let us for a while ignore the overfitting problem. On this assumption, we can say that in supervised learning, we want to adjust model parameters $\boldsymbol{\theta}$ so that we maximize the probability of model returning the *true* label $y^{(i)}$ when it is given example $\mathbf{x}^{(i)}$ from the training dataset. This means we are trying to find the optimal vector of model parameters $\boldsymbol{\theta}_{\text{ML}}$ for which [27, pp. 129 – 130]

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{N} p_{\text{model}}(y^{(i)}|\mathbf{x}^{(i)};\boldsymbol{\theta}), \tag{26}$$

where N is the size of training dataset and $p_{\text{model}}(y^{(i)}|\mathbf{x}^{(i)};\boldsymbol{\theta})$ is the conditional probability of model returning label $y^{(i)}$ given the input $\mathbf{x}^{(i)}$ and model parameters $\boldsymbol{\theta}$.

The problem with this equation is the presence of the product of probabilities. Many of these can be close to zero and, therefore, the computation of the product can be very prone to underflow (numerical rounding of very small numbers to zero). The idea, which can be used to solve this, is to apply a logarithm to the product. This does not change the

[5] However, one may always design some kind of loop so that hyperparameters are changed as well and then select the best set of them. This way we can say the model "learned" the best hyperparameters. Therefore, differentiating between parameters and hyperparameters is not always strict.

argmax and transforms the product to a sum, which is much more suitable for numeric calculations:

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \ln p_{\mathrm{model}}(y^{(i)}|\mathbf{x}^{(i)};\boldsymbol{\theta}). \tag{27}$$

Now, let us take advantage of the fact that argmax does not change when we multiple the function by $1/N$:

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} \ln p_{\mathrm{model}}(y^{(i)}|\mathbf{x}^{(i)};\boldsymbol{\theta}). \tag{28}$$

However, this is equivalent to calculating argmax of the following expectation value:

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x},y\sim\hat{p}_{\mathrm{data}}} \left[\ln p_{\mathrm{model}}(y|\mathbf{x};\boldsymbol{\theta})\right], \tag{29}$$

where $\hat{p}_{\mathrm{data}}$ is empirical distribution defined by the training dataset. As we stated above, in ML we usually define the learning goal as minimizing a loss function J. Nevertheless, finding argmax in the last equation is the same as minimizing loss function:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},y\sim\hat{p}_{\mathrm{data}}} \left[\ln p_{\mathrm{model}}(y|\mathbf{x};\boldsymbol{\theta})\right]. \tag{30}$$

This is in fact the cross-entropy between distributions $\hat{p}_{\mathrm{data}}$ and p_{model}, which is by far the most common loss function used in classification problems.

At this point, we are going to discuss a subset of machine learning – so-called *deep learning*. For completeness, there are ML algorithms that do not belong to the area of deep learning, e.g. support vector machines (SVM), (k-)nearest neighbor(s) regression, decision trees and k-means clustering.

2.2 Deep feedforward networks

Let us talk once again about the chess problem described in section 2.1. As we mentioned, the most natural way to describe the state of the chess game is to write down positions of all pieces in $16 \times 2 \times 2 = 64$-dimensional vector of integers $\mathbf{x} \in \mathbb{N}^{64}$ where $x_i \in \{1, \ldots, 8\}$. However, there are also other, less natural, forms of state description. For instance, we could use coordinates of square centers instead of their indices and thus change integers for floats from the interval [0.5; 7.5]. If we wanted to bring our description even closer to the real world, we could use *exact* positions of the chess pieces, and, last but not least, it would be possible to use polar coordinates.

It is easy to see that a more complicated notation of the state results in less understandable and valuable information. In data science, we call this a *representation* – the way of storing information about the example. In other words, representation gives meaning to the numbers x_i in the datapoint vector $\mathbf{x}$.

Besides the selection of the learning algorithm and performance metric, representation is another choice with a fundamental impact on the success of an ML model. Choosing representation with the best outcome is far from being trivial. Especially due to the fact that representations most natural to humans do not tend to be optimal for computers.

Significant progress has been made in machine learning after models started to be able to learn the optimal representation. Models with this ability initiated the formation of a new area called *deep learning*. Ability to learn representation is what differentiates often interchanged terms "artificial intelligence", "machine learning" and "deep learning" [27, fig. 1.4 and 1.5]:

- *Artificial intelligence* (AI) is the broadest concept. In general, AI algorithms do not have to be able to learn from experience. This term is used to describe all non-natural intelligent behavior. For example, a chess-playing computer program is a form of AI, even if it is as good in the moment of completion as after 100 games, i.e. it still follows rules encoded by a person in the development process.

- *Machine learning* (ML) algorithms (models) are capable of *learning* and constitute a subset of AI. In the broadest sense, the ML model takes a hand-designed set of features ($\mathbf{x}$; i.e. uses hand-designed representation) as input and learns to assign the output (y). The necessity to choose representation is a significant limitation of these algorithms.

- *Representation learning* is subset of ML. This term is less common than the other ones mentioned here. Representation-learning models have an additional layer of abstraction. Besides evaluating the features of an example, they can learn how to extract these *relevant* features from the input.

- *Deep learning* is, again, a subset of the previous class of algorithms. The attribute *deep* describes the model's ability to cover several levels (layers) of abstraction in extracting information from the input. Let us illustrate this on the model classifying images of grayscale hand-written digits [30]. At the input level, imagine there is a vector of brightness levels for each pixel. The first layer is able to identify edges (boundaries between black and white areas) of the image. The second layer processes information about edges and identifies geometric objects (e.g. circles or lines). The last layer decides which number is in the picture using information from the previous layer, which is much simpler than making a decision using raw pixel data.

Deep learning models represent a state of the art in machine learning. Nowadays they are used for countless applications – identification of objects at images, speech/text recognition, recommendations at e-shops, social media and web search engines, prediction of prices, and much more. This way they have a large impact on our lives and also on society as a whole [27, 31].

The backbone of the deep learning is one specific architecture of the model, called *deep feedforward network* of *multilayer perceptron* (MLP). This architecture, with some modifications, is used in all application fields mentioned above. An example of a deep feedforward network can be seen in fig. 3.

Architecture

MLP consists of several layers, represented by individual columns in the figure. From the logical point of view, each layer represents one level of abstraction on the way from the

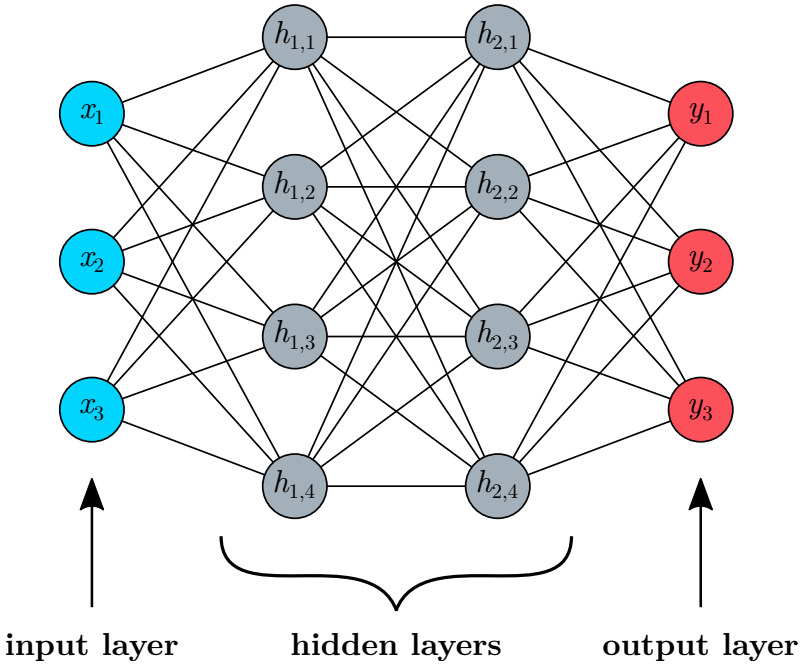

Figure 3: Example of a deep feedforward network. It has 4 layers, of which 2 are hidden. Each circle represents a unit and the lines between units symbolize passing information – this is always done from left to right. Looking at the picture, it is obvious why this architecture uses attribute *network*. They are also called *neural networks* since the units resemble neurons in the human brain.

inputs to the outputs. The number of layers determines the *depth* of the model, hence the name *deep* learning. Each layer comprises a standalone function $f^{(i)}$ and takes the previous layer as its input. The layer consists of multiple units (circles in the figure) and their number represents the *width* of the layer.

In our example, information is passed from the left to the right. Therefore, the leftmost layer is called an *input layer* and is nothing more but a placeholder for the input. Naturally, its width must match the size of the input vector $\mathbf{x}$. The last (rightmost) layer is called the *output layer* as its output is the output of the whole transformation

$$\begin{aligned}\hat{\mathbf{y}} &= f(\mathbf{x}) \\ &= f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x}))).\end{aligned}$$

Layers in the center are called *hidden layers* simply because their output is visible only for other layers of f. These models are called *feedforward* because the information is always passed further forward, never backward to the previous layers.[6]

Each unit in the hidden/output layer represents a simple function. It has two attributes (variables) which are used to produce its input – a vector of *weights* $\mathbf{w}$ and a scalar b called *bias*. For example, the output of one of the units in the first hidden layer is given by

$$h = g(\mathbf{w}^\top \mathbf{x} + b). \tag{31}$$

[6]However, there is an exception to this rule – *recurrent neural networks* (RNNs).

Here $\mathbf{x}$ is the input layer, which is, in other words, the output of the previous layer. The vector of weights and $\mathbf{x}$ are multiplied using the standard dot product producing a scalar. Should we stop after adding b, a layer of such units would represent an *affine* transformation. It is not difficult to figure out that, due to this, the whole model would be an affine transformation of $\mathbf{x}$. Since we want to be able to reproduce also non-linear relations, we apply a non-linear function α, called an *activation function*.

The most widely used activation function is a co-called *rectified linear unit* (ReLU), which is also recommended as a default one, if there is not a more motivated choice. It is defined as

$$\mathrm{ReLU}(x) = \max\{0, x\}. \tag{32}$$

There are lots of other activation functions used in MLPs. Let us only mention a hyperbolic tangent (tanh) and a sigmoid function.

To sum up, transformations of individual layers in our example in fig. 3 can be mathematically described as:

$$\begin{aligned}
\mathbf{h}^{(1)} = f^{(1)}(\mathbf{x}) &= \alpha^{(1)}(\mathbf{W}^{(1)\top}\mathbf{x} + \mathbf{b}^{(1)}) \\
\mathbf{h}^{(2)} = f^{(2)}(\mathbf{h}^{(1)}) &= \alpha^{(2)}(\mathbf{W}^{(2)\top}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}) \\
\hat{\mathbf{y}} = f^{(3)}(\mathbf{h}^{(2)}) &= \alpha^{(3)}(\mathbf{W}^{(3)\top}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}),
\end{aligned}$$

where we organized weights of units in the layer to a matrix, biases to a vector and allowed a different activation function for each layer. Equivalently, if we denote

$$\mathbf{x} \equiv \mathbf{h}^{(0)} \qquad \text{and} \qquad \hat{\mathbf{y}} \equiv \mathbf{h}^{(3)}, \tag{33}$$

we can write[7]

$$\mathbf{h}^{(i)} = \alpha^{(i)}(\mathbf{W}^{(i)\top}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)}). \tag{34}$$

As we have already mentioned, parameters of the network are often denoted as a single vector $\boldsymbol{\theta}$. In this case, $\boldsymbol{\theta}$ would contain elements of all weight matrices $\mathbf{W}^{(i)}$ and bias vectors $\mathbf{b}^{(i)}$. One can easily calculate that our example network has 51 trainable parameters, i.e. $\boldsymbol{\theta} \in \mathbb{R}^{51}$.

Backpropagation – algorithm for gradient-based learning

Equation (34) describes how information propagates forwards (from input to output) in a deep network. The question however is; *how* can such a deep network learn?

As we already described, learning of ML model means minimizing the loss J. In majority of applications, total loss function can be calculated as an expectation value for a per-example loss function $L(y, \mathbf{x}, \boldsymbol{\theta})$, which is the function of the true label y, the input $\mathbf{x}$ and model parameters $\boldsymbol{\theta}$. Note that

$$L(y, \mathbf{x}, \boldsymbol{\theta}) \equiv L(y, \hat{y}(\mathbf{x}, \boldsymbol{\theta})) \equiv L(y, \hat{y}), \tag{35}$$

so we can write

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\mathrm{data}}} \left[L(y, \mathbf{x}, \boldsymbol{\theta}) \right] \equiv \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\mathrm{data}}} \left[L(y, \hat{y}) \right], \tag{36}$$

[7] Layers described by eq. (34) are often called *dense*, as each unit in the layer is connected to all of the units in the previous layer.

where $\hat{p}_{\text{data}}$ is the empirical distribution given by the training dataset.

The most precise way to calculate eq. (36) is to use the whole dataset:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},y\sim\hat{p}_{\text{data}}}\left[L(y,\mathbf{x},\boldsymbol{\theta})\right] = \frac{1}{N}\sum_{i=1}^{N} L(y^{(i)},\mathbf{x}^{(i)},\boldsymbol{\theta}). \tag{37}$$

In fact, the last equation gives us a recipe how to teach a deep network – take the whole training dataset, transform it through the network to obtain a set of predicted labels, compare them with the true ones and calculate $J(\boldsymbol{\theta})$ using eq. (37) and then update the parameters with

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon\mathbf{g}, \tag{38}$$

where $\leftarrow$ is used to denote assignment (i.e. $=$ in a majority of programming languages), ε is the so-called *learning rate* and $\mathbf{g}$ is the gradient of loss with respect to model parameters:

$$\mathbf{g} = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}). \tag{39}$$

This algorithm is called *the gradient descent*.

However, the problem is that training datasets are typically very large. Therefore, the calculation of predicted labels $\{\hat{y}^{(i)}\}_{i=1}^{N}$ for the whole dataset is computationally expensive. The solution is to use a trick called the *stochastic gradient descent* (SGD).

Key principle of the SGD is to use just a subset of the training dataset to approximate $J(\boldsymbol{\theta})$ instead of the full dataset. We call this subset a *batch*.[8] Let us denote the number of examples it contains as M. Typically $1 \ll M \sim 10^2 - 10^4 \ll N$. Thus:

$$J(\boldsymbol{\theta}) \approx \frac{1}{M}\sum_{i=1}^{M} L(y^{(i)},\mathbf{x}^{(i)},\boldsymbol{\theta}). \tag{40}$$

When we substitute eq. (40) for (39), we get estimate of the gradient $\mathbf{g}$, which can be used in the learning step (38). In practice, learning using the SGD results in doing less precise steps in the space of model parameters when compared to the gradient descent. Nevertheless, these steps can be done much faster since evaluating eq. (40) is by orders of magnitude less computationally expensive than the calculation of (37).

The stochasticity of the SGD arises from the fact that the batch is sampled from the training dataset randomly. Therefore, two separate SGD runs from the same starting point $\boldsymbol{\theta}_{\text{init}}$ for the same number of steps result in different final sets of parameters $\boldsymbol{\theta}_{\text{final}}$.

The previous lines answered the question, how to efficiently approximate the value of $J(\boldsymbol{\theta})$. However, we still need to know what is the optimal way to calculate its gradient with respect to the vector of parameters $\boldsymbol{\theta}$. In the majority of deep-learning implementations this is done using an algorithm called the *backpropagation* [32], commonly abbreviated as "backprop".

In order to explain this algorithm, let us begin with a simple observation. When computing a (partial) derivative of a nested function $z = f(f(f(w)))$, obtained by applying f three times in a row, one can use the well-known *chain rule* of derivatives:

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}, \qquad \text{where} \quad x = f(w) \quad y = f(x) \quad z = f(y). \tag{41}$$

[8] Also called *minibatch* in the literature.

This can be written as

$$\frac{\partial z}{\partial w} = f'(y)f'(x)f'(w) \tag{42}$$
$$= f'(f(f(w))f'(f(w))f'(w). \tag{43}$$

As can be seen from this equation, when calculating derivatives of the nested functions, one has to evaluate f at the same point multiple times. For instance, in this example we needed to evaluate $f(w)$ twice when calculating the $\partial z/\partial w$.[9] It is easy to imagine, that with *deep* function compositions this effect will be even stronger. Therefore, there are two possible approaches:

1. save all the partial results (in this case values of variables x and y) and thus trade off higher memory demands for lower computing time, or

2. do not save partial results but calculate them each time they are needed and this way trade off higher computing time for lower memory demands.

For modern computers, saving even a million floating-point variables, which results in a few megabytes of occupied memory, is nothing to be bothered about and thus the former option is always the better one, as it can increase the speed of learning by orders of magnitude.

The backpropagation algorithm takes advantage of this fact and views the deep network as a directed acyclic computational graph beginning at its outputs and finishing at the input. This way, it manages to visit each node (unit) only once when calculating the gradient (39). It is explained in the algorithm 1 [27, p. 209].[10]

Algorithm 1: The backpropagation algorithm

Input: l – number of layers excluding the input layer

After the forward propagation, compute the gradient on the output layer:

$\mathbf{g} \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} \frac{1}{M} \sum_{j=1}^{M} L(\hat{y}^{(j)}, y^{(j)})$;

Iterate through the network layers:

for $i = l, l-1, \ldots, 1$ **do**

Use gradient on the layer outputs to compute gradient with respect to the input $\mathbf{a}^{(i)} = \mathbf{W}^{(i)\top}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)}$ of the activation function α:

$\mathbf{g} \leftarrow \nabla_{\mathbf{a}^{(i)}} J = \mathbf{g} \odot \alpha'(\mathbf{a}^{(i)})$

where $\odot$ is the Hadamard (element-wise) product;

Compute gradient on weights and biases:

$\nabla_{\mathbf{b}^{(i)}} J = \mathbf{g}$;

$\nabla_{\mathbf{W}^{(i)}} J = \mathbf{g}\ \mathbf{h}^{(i-1)\top}$;

Calculate gradient with respect to the output of the layer $\mathbf{h}^{(i-1)}$:

$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(i-1)}} J = \mathbf{W}^{(i)}\ \mathbf{g}$;

[9] Note that this result is not limited to nesting the *same* function f.

[10] For explanation of gradient calculations see e.g. https://web.stanford.edu/class/cs224n/readings/gradient-notes.pdf

As can be seen, using the backpropagation algorithm, the error/loss is propagated from the output *back* to the model parameters (therefore its name). Those can be then updated using $\nabla_{\boldsymbol{\theta}} J$, as suggested by eq. (38).

There are some algorithms that enhance the SGD. Namely, they gradually change the learning rate ε and often utilize some kind of *momentum* – inertia of the learning steps in the direction, which leads to a decrease in the $J(\boldsymbol{\theta})$ during multiple successive steps. This class of algorithms includes AdaGrad [33], RMSProp, Adam [34] and others [27, pp. 302 – 307].

In conclusion, a single learning iteration of a deep ML model proceeds as follows:

1. A batch of examples $\mathbf{x}$ is randomly chosen from the training dataset and is processed by the network to produce corresponding predictions y. This is called the forward propagation.

2. The loss $J(\boldsymbol{\theta})$ is calculated as an average of the per-example loss $L(y, \hat{y}(\mathbf{x}, \boldsymbol{\theta}))$ over the batch.

3. The loss is backpropagated through the network and its derivative with respect to each parameter θ_i is calculated. After that learning step is done, using either equation (38) or some kind of more sophisticated algorithm – model parameters are slightly adjusted in order to decrease the loss J.

2.3 Deep generative models

In this section, we are going to discuss a specific group of deep networks called *generative models*. As their name suggests, they are used for the generation of new samples (i.e. for sampling) from a certain distribution. In general, sampling is a much more difficult task than, for example, regression or classification. While these two represent some kind of data compression (targets are usually low-dimensional when compared to inputs), generative models must learn to produce full-sized examples that are close to those in the training dataset.

Although sampling is a difficult task, it can be utilized in many areas. In most cases, the target distribution $p^*(\mathbf{x})$ is not known exactly but is approximated by the empirical distribution of the training dataset $\hat{p}_{\text{data}}(\mathbf{x})$. For instance, a generative model can be used to produce images of buildings for a computer game, which is much easier than designing numerous building images manually.

Currently, there are a few different ways how to approach the sampling problem; each of them based on a different principle. However, all of them use some form of stochasticity. This is important for if we want to produce various not-identical samples, we need a random factor that decides *which* example from the set of available examples will be produced.

An easily understandable overview of the generative models was given in [35]. A more exhaustive review, which also explains the mathematical bases of these models, was recently given in [36].

Energy-based models

These models are inspired by physics. They are based on the fact that every PDF $p(\mathbf{x})$ can be written as

$$p(\mathbf{x}) = \frac{e^{-E(\mathbf{x})}}{\int e^{-E(\mathbf{x})} d\mathbf{x}}.$$

The model learns to assign low energies E to probable examples $\mathbf{x}$ and high energies to the improbable ones. A typical example of an energy-based model is the *Boltzmann machine*. These models are quite simple but scale poorly when data are of a higher dimension and sampling is not as straightforward as for other methods.

Variational autoencoders

Autoencoders are models whose goal is to produce output that is as similar to the input as possible. Naturally, learning a simple identity function would not be a problem. However, autoencoders always contain an obstacle that makes reconstructing input in the model output difficult.

This can be achieved, for instance, by symmetric architecture with the middle part having a significantly lower width than the input $\mathbf{x}$. This way autoencoder consists of the *encoder* f, which has to efficiently encode input $\mathbf{x}$ into low-dimensional code $\mathbf{c}$, and the *decoder* f^{-1}, which is the exact inverse to the encoder and has to reconstruct $\mathbf{x}$ from the $\mathbf{c}$ as precisely as possible.

After the training, the decoder can be used for sampling, when provided with codes $\mathbf{c}$. However, since we do not understand the representation of codes, we don't know what codes should be provided in order to sample examples similar to the training dataset.

The solution is to force the encoder to use codes with some wanted properties. Variational autoencoders achieve this by forcing the encoder to use such codes so that their distribution resembles normal Gaussian distribution. This way, after the training, sampling can be done easily by sampling codes $\mathbf{c} \sim \mathcal{N}(0, I_n)$ from the multivariate normal distribution and transforming them with the decoder.

Generative adversarial networks

Generative adversarial networks, or for short, GANs and use a completely different approach. GANs consist of two networks: a *discriminator* and a *generator*. Discriminator $D : \mathbb{R}^n \to [0; 1]$ estimates the probability that certain example $\mathbf{x}$ comes from the training dataset. The generator takes a random vector $\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})$ as an input and produces example $\mathbf{x}$ similar to the training dataset.

During the training phase, these two play a game in which they compete against each other. The discriminator gets better in distinguishing between real and fake examples. It is learned both on the training dataset (*real* examples) and the samples from the generator (*fake* examples). The generator learns to produce examples, that are classified as real by the discriminator and learns solely from its feedback (it does not have direct access to the training dataset).

GANs represent a state of the art in producing images, e.g. human faces. However, they are very difficult to learn and struggle to understand some concepts, such as the

number of objects. This may result, quite amusingly, in producing also faces with three eyes.

Autoregressive likelihood models

These are based on the so-called chain rule of the probability, which states that probability of vector variable $\mathbf{x} = (x_1, \ldots, x_n)^\top$ can be expressed as:

$$p(\mathbf{x}) = p(x_1, \ldots, x_n) = \prod_{i=1}^{n} p(x_i | x_{i-1}, \ldots, x_1) \tag{44}$$

i.e. PDF of variable x_i depends only on values of $x_{i-1}, \ldots, x_1$. These models are very good in density estimation. However, since sampling is a sequential process, it is slow when compared to other methods.

Normalizing flows

The last currently used approach of sampling a PDF $p_{\mathbf{x}}(\mathbf{x})$ is to use a simple latent probability distribution $p_{\mathbf{z}}(\mathbf{z})$ (e.g. a normal Gaussian distribution) and train mapping $f : \mathbb{R}^n \to \mathbb{R}^n$ so that when samples from the $p_{\mathbf{z}}(\mathbf{z})$ are mapped through f, their distribution is given by the target distribution $p_{\mathbf{x}}(\mathbf{x})$. As with other methods, the target distribution is usually not known exactly. Instead, it is approximated by the empirical distribution of the training dataset.

This procedure is based on the *change of variable formula*:

$$p_{\mathbf{x}}(\mathbf{x}) = \left| \frac{\partial \mathbf{z}}{\partial \mathbf{x}} \right| p_{\mathbf{z}}(\mathbf{z}) = \left| \frac{\partial f^{-1}}{\partial \mathbf{x}} \right| p_{\mathbf{z}}(\mathbf{z}) = \left| \frac{\partial f}{\partial \mathbf{z}} \right|^{-1} p_{\mathbf{z}}(\mathbf{z}) = \frac{p_{\mathbf{z}}(\mathbf{z})}{\det(\mathbf{J}_f)}. \tag{45}$$

Transformation f has to be invertible and must have a tractable determinant of its Jacobian matrix $\det(\mathbf{J}_f)$.

Normalizing flows represent the most important class of deep generative models in the context of this thesis given that one of them is used in the Boltzmann generators. We will discuss them in more detail later.

3 Boltzmann generators

3.1 The idea

As we explained in the previous section, a prominent problem of the MD and MCMC methods is the simulation of rare events. Traditional enhanced sampling techniques facilitated solving of this problem but still have some downsides. Most importantly, these methods are very sensitive to the choice of a reaction coordinate (collective variable).

In 2002, Zhu *et al.* [37, 38] came with a novel approach to deal with the problem of rare events, called the *reference potential spatial warping algorithm* (REPSWA). It is based on the fact that physics of the system cannot depend on the set of coordinates that are used to describe its state. Therefore, the idea is to use a transformation F so that the potential-energy surface in the coordinate space $\mathbf{z} = F(\mathbf{x})$ is simpler than in the original one.

Namely, we would like to design transformation F that brings minima closer to each other and lowers the height of barriers that separate them, without changing the partition function and thus the physics of the system. This way, the Boltzmann distribution proportional to $\exp[-\beta U(\mathbf{z})]$ would be much easier to sample using the trajectory-based methods as the MD and MCMC. Whereas authors succeeded in designing such transformation for linear polymers, which is a very limited group of all systems, creating such transformation for a more complicated general system is downright unfeasible.

Nevertheless, this obstacle can be solved by utilizing deep neural networks. Let us recall that deep NN are in fact very complicated functions (transformations), which are able to, for instance, map grayscale images of handwritten digits to their values. Using them to learn the convenient transformation was first time proposed by Noé *et al.* in 2019 [2]. They named this new method as *the Boltzmann generator*.

In the very center of the Boltzmann generator (BG) there is a transformation $F : \mathbf{z} \to \mathbf{x}$, which is a deep neural network. This transformation maps vectors $\mathbf{z}$ from the so-called *latent* space to the *real* configuration space of vectors $\mathbf{x}$. As already explained in section 1, the vector $\mathbf{x}$ contains full information about the microstate of the physical system, e.g. a full set of particle coordinates

$$\mathbf{x} = (x_{1x}, x_{1y}, x_{1z}, x_{2x}, \ldots, x_{Nz})^\top, \tag{46}$$

where N is the number of particles.

F learns such mapping so that when we sample latent vectors $\mathbf{z}$ from a simple distribution and transform them through F resulting vectors $\mathbf{x}$ are distributed according to the Boltzmann distribution (7). We use a latent distribution that can be sampled in one pass, i.e. we are able to easily produce a sequence of independent samples from this distribution. Therefore, if F learns the mapping we described, we will be able to sample the Boltzmann distribution in one pass, which solves the problem of rare events.

The process of sampling the Boltzmann distribution with the BG is illustrated in the fig. 4. In the first step, we sample a simple latent probability distribution. We use multivariate normal distribution as the latent distribution throughout the thesis. However, other simple PDFs can be used as well (e.g. Cauchy or log-normal distribution). In the

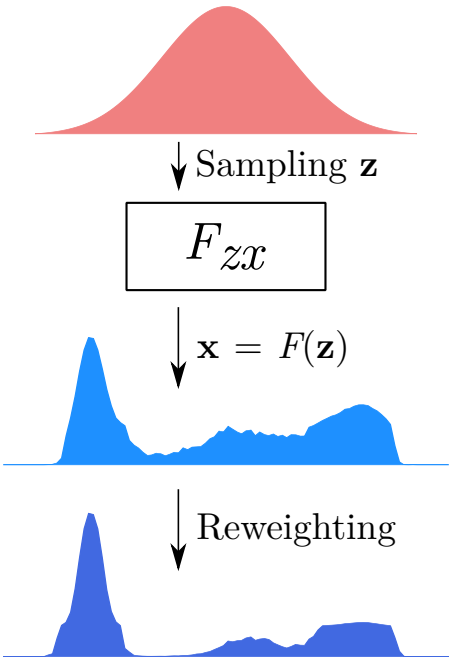

Figure 4: Schematic illustration of the sampling process of the Boltzmann generator. 1. Samples from the simple latent distribution are sampled. 2. These samples are transformed through F, which learns during the training how to map latent vectors $\mathbf{z}$ so that produced probability distribution of $\mathbf{x}$ is close to the Boltzmann one. 3. Samples are reweighted with proper statistical weights in order to get the Boltzmann distribution.

second step, latent samples are transformed through F to obtain samples $\mathbf{x}$ in the configuration space. We want the produced distribution $p_X(\mathbf{x})$ to resemble the Boltzmann distribution. However, in most cases $p_X(\mathbf{x})$ is just *similar* to the target Bolt. dist. Therefore, another step is employed in order to improve precision – samples from $p_X(\mathbf{x})$ are used with statistical weights

$$w(\mathbf{x}) = \frac{\exp[-\beta U(\mathbf{x})]}{p_X(\mathbf{x})}, \tag{47}$$

as done in self-normalized importance sampling (13), which was already explained in section 1.1. The resulting distribution is even closer to the Boltzmann distribution.

The transformation F is trained with two methods, which are in practice used simultaneously. These are:

1. **Training by example:** If we know some important (high-probability) microstates of the system, we can train F so that we promote their sampling. These can be, for example, configurations from individual metastable states, obtained by MD or MCMC simulations. If F is *invertible*, we can map them from the real space back to the latent space ($\mathbf{z} = F^{-1}(\mathbf{x})$) and teach F in the way that these latent samples $F^{-1}(\mathbf{x})$ occupy the region near the origin of the latent space, which is frequently sampled due to the Gaussian normal distribution $p_Z(\mathbf{z})$. This does *not* require the knowledge of Boltzmann probabilities of used states $\mathbf{x}$.

2. **Training by energy:** The second method uses sampled latent vectors $\mathbf{z} \sim p_Z(\mathbf{z})$

from the normal distribution that are transformed to the real space through F. We can then evaluate their energies $U(F(\mathbf{z}))$ and train F so that these energies are low, which means samples will have high Boltzmann probability.

Although the main goal of the Boltzmann generators is to solve the sampling problem, they can also be used for calculating free-energy profiles along the reaction coordinates, finding new metastable states as well as transition paths between them. This will be discussed later. In the following section, we are going to provide more rigorous mathematical fundamentals for the method.

3.2 Theoretical background

This section will be mainly based on the part *Materials and methods* of the BG article [2]. We will also use the same notation where possible. As we already explained, mapping F has to be invertible in order to use training by example. As will be shown, F must have also another feature – determinant of the Jacobian matrix of its transformation must be easily calculable. Let us, for now, postpone the discussion of the question *how* can such transformation be created. We deal with this problem in section 3.3.

Since F is a deep neural network, mapping $\mathbf{z} \to \mathbf{x}$ depends on its parameters. Let us denote the vector of these parameters as $\boldsymbol{\theta}$. In order to avoid confusion, we will use F_{zx} to denote transformation $F : \mathbf{z} \to \mathbf{x}$ and F_{xz} for $F^{-1} : \mathbf{x} \to \mathbf{z}$. Thus:

$$\begin{aligned} \mathbf{x} &= F_{zx}(\mathbf{z}; \boldsymbol{\theta}) \\ \mathbf{z} &= F_{xz}(\mathbf{x}; \boldsymbol{\theta}). \end{aligned} \tag{48}$$

Corresponding Jacobian matrices are:

$$\begin{aligned} \mathbf{J}_{zx}(\mathbf{z}) &= \left[\frac{\partial F_{zx}(\mathbf{z}; \boldsymbol{\theta})}{\partial z_1}, \ldots, \frac{\partial F_{zx}(\mathbf{z}; \boldsymbol{\theta})}{\partial z_n} \right] \\ \mathbf{J}_{xz}(\mathbf{x}) &= \left[\frac{\partial F_{xz}(\mathbf{x}; \boldsymbol{\theta})}{\partial x_1}, \ldots, \frac{\partial F_{xz}(\mathbf{x}; \boldsymbol{\theta})}{\partial x_n} \right], \end{aligned} \tag{49}$$

where n is the dimension of the physical system, i.e. $\mathbf{x}, \mathbf{z} \in \mathbb{R}^n$.

As it is known from the calculus, the determinant of the Jacobian matrix describes how the transformation changes the size of a volume element. As we will use such determinants frequently, let us denote

$$\begin{aligned} R_{zx}(\mathbf{z}) &= |\det \mathbf{J}_{zx}(\mathbf{z})| \\ R_{xz}(\mathbf{x}) &= |\det \mathbf{J}_{xz}(\mathbf{x})|. \end{aligned} \tag{50}$$

If we apply the *change of variables formula* to the probability distributions in the latent and real space, we can see that these are linked to each other. In particular:

$$\begin{aligned} p_X(\mathbf{x}) &= p_Z(\mathbf{z}) R_{zx}^{-1}(\mathbf{z}) \\ &= p_Z(F_{xz}(\mathbf{x})) R_{xz}(\mathbf{x}) \end{aligned} \tag{51}$$

$$\begin{aligned} p_Z(\mathbf{z}) &= p_X(\mathbf{x}) R_{xz}^{-1}(\mathbf{x}) \\ &= p_X(F_{zx}(\mathbf{z})) R_{zx}(\mathbf{z}), \end{aligned} \tag{52}$$

where $p_X(\mathbf{x})$ and $p_Z(\mathbf{z})$ are probability distributions in the real and latent space, respectively. Let us also define the *reduced* (or dimensionless) energy $u(\mathbf{x})$:

$$u(\mathbf{x}) \equiv \frac{U(\mathbf{x})}{k_B T} = \beta U(\mathbf{x}), \tag{53}$$

where $U(\mathbf{x})$ is the potential energy of the system, T is its temperature, and k_B is the Boltzmann constant.

So far we used p_Z to denote probability density of the normal distribution in the latent space and p_X for PDF resulting from mapping samples from p_Z through F_{zx}, see eq. (51). From now on, we will distinguish between the "exact" probabilities and generated probabilities. Namely, μ_Z will be the Gaussian normal distribution in the latent space and $\mu_X \propto \exp[-u(\mathbf{x})]$ the Boltzmann distribution in the real space. In other words, the aim is to achieve

$$\frac{1}{Z_Z} e^{-\frac{1}{2}\mathbf{z}^2/\sigma_Z^2} = \mu_Z \xrightarrow{F_{zx}} \mu_X = \frac{1}{Z_X} e^{-u(\mathbf{x})}, \tag{54}$$

where σ_Z is the standard deviation of the latent normal distribution and Z is the partition function (which also has the role of a normalization constant). We can choose an arbitrary value for σ_Z; we choose $\sigma_Z = 1$, which is the simplest option. However, F never *exactly* learns transformation (54). What we really get is a *generated* probability distribution q_X. The same applies vice versa, if we map μ_X through F_{xz}, we do not get μ_Z but instead a different distribution q_Z. Thus

$$\begin{aligned} \mu_Z &\xrightarrow{F_{zx}} q_X \\ \mu_X &\xrightarrow{F_{xz}} q_Z. \end{aligned} \tag{55}$$

Derivation of loss functions

At this point we are going to derive loss functions that should be minimized in order to train the transformation F. Our starting point will be the *Kullback–Leibler* (KL) *divergence*, which is used as a measure of difference between two probability densities in mathematical statistics. It is also used in numerous ML applications. KL divergence between distributions p and q is defined as

$$\begin{aligned} D_{\mathrm{KL}}(p\|q) &= \int p(\mathbf{x}) \left[\ln p(\mathbf{x}) - \ln q(\mathbf{x})\right] d\mathbf{x} \\ &= -H_p - \int p(\mathbf{x}) \ln q(\mathbf{x}) d\mathbf{x}, \end{aligned} \tag{56}$$

where we used the definition of entropy H in the information theory: $H_p = -\int p(\mathbf{x}) \ln p(\mathbf{x}) d\mathbf{x}$. The *physical* entropy S is related to H by $S = k_B H$. Note that the KL divergence is not symmetric, i.e. $D_{\mathrm{KL}}(p\|q) \neq D_{\mathrm{KL}}(q\|p)$.

Let us start the derivation of a loss used in the training by energy with the KL divergence between μ_Z and q_Z which we want to minimize:

$$D_{\mathrm{KL}}(\mu_Z\|q_Z) = -H_Z - \int \mu_Z(\mathbf{z}) \ln q_Z(\mathbf{z};\boldsymbol{\theta}) d\mathbf{z} \quad \text{where} \quad H_Z = -\int \mu_Z(\mathbf{z}) \ln \mu_Z(\mathbf{z}) d\mathbf{z}.$$

When we use the second line of eq. (52), we get

$$\begin{aligned} D_{\mathrm{KL}}(\mu_Z \| q_Z) &= -H_Z - \int \mu_Z(\mathbf{z}) \ln \left[\mu_X(F_{zx}(\mathbf{z};\boldsymbol{\theta}))R_{zx}(\mathbf{z};\boldsymbol{\theta})\right] d\mathbf{z} \\ &= -H_Z - \int \mu_Z(\mathbf{z}) \left[\ln \mu_X(F_{zx}(\mathbf{z};\boldsymbol{\theta})) + \ln R_{zx}(\mathbf{z};\boldsymbol{\theta})\right] d\mathbf{z} \end{aligned}$$

and after we express μ_X using eq. (54):

$$\begin{aligned} D_{\mathrm{KL}}(\mu_Z \| q_Z) &= -H_Z - \int \mu_Z(\mathbf{z}) \left[\ln(1/Z_X) - u(F_{zx}(\mathbf{z};\boldsymbol{\theta})) + \ln R_{zx}(\mathbf{z};\boldsymbol{\theta})\right] d\mathbf{z} \\ &= -H_Z + \ln Z_X - \int \mu_Z(\mathbf{z}) \left[-u(F_{zx}(\mathbf{z};\boldsymbol{\theta})) + \ln R_{zx}(\mathbf{z};\boldsymbol{\theta})\right] d\mathbf{z} \\ &= -H_Z + \ln Z_X + \mathbb{E}_{\mathbf{z}\sim\mu_Z(\mathbf{z})} \left[u(F_{zx}(\mathbf{z};\boldsymbol{\theta})) - \ln R_{zx}(\mathbf{z};\boldsymbol{\theta})\right]. \end{aligned}$$

However, since H_Z and Z_X are constants in trainable parameters $\boldsymbol{\theta}$ of F, minimizing $D_{\mathrm{KL}}(\boldsymbol{\theta})$ is the same as minimizing the loss function

$$\boxed{J_{\mathrm{KL}}(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{z}\sim\mu_Z(\mathbf{z})} \left[u(F_{zx}(\mathbf{z};\boldsymbol{\theta})) - \ln R_{zx}(\mathbf{z};\boldsymbol{\theta})\right]}, \tag{57}$$

which is **the Kullback-Leibler (KL) loss** minimized in training by energy. Note that while the first term of the KL loss promotes sampling of low-energy states, the second one penalizes shrinking the space in transformation F_{zx}, i.e. when this mapping focuses on the single region in $\mathbf{x}$-space too much. This way the second term forces F_{zx} to increase entropy of the produced PDF $q_X(\mathbf{x})$.

It can be shown that the KL loss is, up to a constant, equal to the free energy of the system. We will start with the definition of entropy H_X of the produced distribution q_X:

$$\begin{aligned} H_X &= -\int_{\mathbf{x}} q_X(\mathbf{x}) \ln q_X(\mathbf{x}) d\mathbf{x} \\ &= -\int_{\mathbf{z}} q_X(F_{zx}(\mathbf{z})) \ln q_X(F_{zx}(\mathbf{z})) R_{zx}(\mathbf{z}) d\mathbf{z} \\ &= -\int_{\mathbf{z}} \mu_Z(\mathbf{z}) \ln q_X(F_{zx}(\mathbf{z})) d\mathbf{z}, \end{aligned}$$

where the second row was obtained by changing the integration variable and the last one using the eq. (52). If we now use the same eq. for the q_X, we get:

$$\begin{aligned} H_X &= -\int_{\mathbf{z}} \mu_Z(\mathbf{z}) \ln \left[\mu_Z(\mathbf{z}) R_{zx}^{-1}(\mathbf{z})\right] d\mathbf{z} \\ &= \left(-\int_{\mathbf{z}} \mu_Z(\mathbf{z}) \ln \mu_Z d\mathbf{z}\right) + \mathbb{E}_{\mathbf{z}\sim\mu_Z(\mathbf{z})} \left[\ln R_{zx}(\mathbf{z})\right] \\ &= H_Z + \mathbb{E}_{\mathbf{z}\sim\mu_Z(\mathbf{z})} \left[\ln R_{zx}(\mathbf{z})\right]. \end{aligned}$$

When we compare the last equation with (57), we can see that

$$J_{\mathrm{KL}} = \langle u \rangle_{q_X} - H_X + H_Z = \frac{1}{k_B T} \left[\langle U \rangle_{q_X} - T S_X\right] + H_Z = \frac{A}{k_B T} + H_Z, \tag{58}$$

where $S_X = k_B H_X$ and A is the free energy of the system with the probability of microstates given by $q_X(\mathbf{x})$. There are two points worth noticing regarding this equation. First, if two BGs use the same latent distribution, they have the same H_Z and thus J_{KL} measures the free energy up to the same constant. This way we can determine the free-energy difference between two systems by using two unlinked BGs – for example, each one trained on configurations from a different metastable state. Second, eq. (58) means that we can view minimizing of J_{KL} in the training as the process during which the system approaches thermodynamic equilibrium and during which, as is well known from physics, the free energy decreases.

The KL divergence is a starting point also in the derivation of a loss used in the training by example. This time we will express the KL divergence between μ_X and q_X:

$$\begin{aligned} D_{\mathrm{KL}}(\mu_X \| q_X) &= \int \mu_X(\mathbf{x})[\ln \mu_X(\mathbf{x}) - q_X(\mathbf{x};\boldsymbol{\theta})]d\mathbf{x} \\ &= -H_X - \int \mu_X(\mathbf{x}) \ln q_X(\mathbf{x};\boldsymbol{\theta}) d\mathbf{x} \\ &= -H_X - \int \mu_X(\mathbf{x}) \ln \left[\mu_Z(F_{xz}(\mathbf{x};\boldsymbol{\theta})) R_{xz}(\mathbf{x};\boldsymbol{\theta})\right] d\mathbf{x} \\ &= -H_X - \int \mu_X(\mathbf{x}) \left[\ln(1/Z_Z) - \frac{1}{2}\|F_{xz}(\mathbf{x};\boldsymbol{\theta})\|^2 + \ln R_{xz}(\mathbf{x};\boldsymbol{\theta})\right] d\mathbf{x} \\ &= -H_X + \ln Z_Z + \mathbb{E}_{\mathbf{x}\sim\mu_X(\mathbf{x})}\left[\frac{1}{2}\|F_{xz}(\mathbf{x};\boldsymbol{\theta})\|^2 - \ln R_{xz}(\mathbf{x};\boldsymbol{\theta})\right], \end{aligned}$$

where we used eq. (51) to obtain the third line and (54) with the fact that $\sigma_Z = 1$ for the fourth line. Nevertheless, the problem is that we cannot sample μ_X – this is what we are trying to achieve with the BG. The solution is to approximate μ_X by samples obtained from the MCMC or thermostatted MD simulation. We will denote the empirical distribution of the training dataset as $\rho(\mathbf{x})$. Thus, after accepting this approximation and omitting terms constant in $\boldsymbol{\theta}$, we get:

$$\boxed{J_{\mathrm{ML}}(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}\sim\rho(\mathbf{x})}\left[\frac{1}{2}\|F_{xz}(\mathbf{x};\boldsymbol{\theta})\|^2 - \ln R_{xz}(\mathbf{x};\boldsymbol{\theta})\right]}. \tag{59}$$

Note that this is in fact an expectation value of the negative log-likelihood of sampling configurations from the training dataset:

$$J_{\mathrm{ML}}(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x}\sim\rho(\mathbf{x})}\left[\ln q_X(\mathbf{x};\boldsymbol{\theta})\right]. \tag{60}$$

Therefore, minimizing J_{ML} maximizes the probability of sampling configurations from $\rho(\mathbf{x})$ (recall the paragraph "Maximum likelihood estimation" in section 2.1). Because of this, J_{ML}, which is used in training by example, is called **the maximum likelihood (ML) loss**.

There is also a third loss used in the training of BG – **the reaction coordinate (RC) loss**. This loss uses user-defined RC $r(\mathbf{x})$, where $r : \mathbb{R}^n \to \mathbb{R}$. The RC loss is defined as the negative entropy of the RC distribution $p(r(\mathbf{x}))$ with $\mathbf{x} \sim q_X(\mathbf{x})$:

$$J_{\mathrm{RC}} = -H_{\mathrm{RC}} = \int p(r(\mathbf{x})) \ln p(r(\mathbf{x}))\ dr(\mathbf{x})$$

$$\boxed{J_{\mathrm{RC}} = \mathbb{E}_{\mathbf{x}\sim q_X(\mathbf{x})}\left[\ln p(r(\mathbf{x}))\right]}. \tag{61}$$

Thus minimizing this loss maximizes the entropy of the RC distribution, i.e. promotes sampling of configurations along the RC. Note that there is an essential difference between this loss and the first two losses – the RC loss is not based on minimizing the KL divergence between real and target distribution. Therefore, minimizing J_{RC} does not teach the system to sample the Boltzmann distribution. On the contrary, in most cases, this means shifting q_X away from the μ_X. This is, however, not an issue, since this bias can be usually easily corrected by reweighting the samples.

The total loss which is minimized in the training is the linear combination of the three introduced loss functions:

$$\boxed{J = w_{\mathrm{KL}}J_{\mathrm{KL}} + w_{\mathrm{ML}}J_{\mathrm{ML}} + w_{\mathrm{RC}}J_{\mathrm{RC}}}, \tag{62}$$

where $w_{\mathrm{KL}}, w_{\mathrm{ML}}$ and w_{RC} are weights of the losses.

The last formula we need to derive is the one for the statistical weight $w(\mathbf{x})$ introduced in eq. (47). Note that we can multiply and divide these weights arbitrarily as $w \to \alpha w$ for all weights does not have any effect (see eq. (13)). For the weight w holds:

$$w(\mathbf{x}) = \frac{e^{-u(\mathbf{x})}}{q_X(\mathbf{x})} = \frac{e^{-u(\mathbf{x})}}{\mu_Z(F_{xz}(\mathbf{x}))R_{xz}(\mathbf{x})} = Z_Z\frac{\exp[-u_X(\mathbf{x})]}{\exp[-u_Z(F_{xz}(\mathbf{x}))]R_{xz}(\mathbf{x})},$$

where we denoted $u(\mathbf{x}) \equiv u_X(\mathbf{x})$ in order to avoid confusion. After using identities well known at his point, w can be expressed as:

$$w(\mathbf{x}) \propto \exp\left[-u_X(\mathbf{x}) + u_Z(F_{xz}(\mathbf{z})) - \ln R_{xz}(\mathbf{x})\right] \tag{63}$$

or, equivalently, if we express $\mathbf{x}$ as a result of sampling in the latent space:

$$w(F_{zx}(\mathbf{z})) \propto \exp\left[-u_X(F_{zx}(\mathbf{z})) + u_Z(\mathbf{z}) + \ln R_{zx}(\mathbf{z})\right]. \tag{64}$$

These weights are also used when the BG is employed to calculate the free-energy profile (also known as the *potential of mean force* – PMF) along the RC: A normalized histogram $p(r(\mathbf{x}))$ is created, with samples $\mathbf{x} \sim q_X(\mathbf{x})$ weighted using (63) and after that PMF can be calculated as $A(r) = -k_BT\ln(p(r))$.

3.3 Real NVP transformations

From the previous text it is clear that transformation F of the BG, which is being trained, must have the following properties:

1. it must be easily *invertible*
2. and must have easily *calculable determinant of Jacobian matrix*.

The latter requirement is more restrictive than the former one.

There is a class of deep generative models with above mentioned features called the **normalizing flows** [39], which have been introduced in section 2.3. It is important that

transformations with these properties are *composable*. In other words, if two transformations f_1, f_2 are invertible and their determinants $|\det \mathbf{J}_{f_{1,2}}|$ are tractable, then

$$\begin{aligned}(f_2 \circ f_1)^{-1} &= f_1^{-1} \circ f_2^{-1} \\ |\det \mathbf{J}_{f_2 \circ f_1}| &= |\det \mathbf{J}_{f_2}| \cdot |\det \mathbf{J}_{f_1}|,\end{aligned} \tag{65}$$

and thus composition $f_2 \circ f_1$ is also invertible and has tractable $|\det \mathbf{J}|$, since multiplication is a fast operation.

Many types of normalizing flows utilize this idea. They provide a template for constructing f which can be then stacked multiple times to get the desired complexity of the transformation $F = f_N \circ \ldots \circ f_1$.

This is also the case of *real-valued non-volume preserving transformations* (Real NVP) [40], which are used by the Boltzmann generator. Real NVP transformations are based on their predecessor the *Non-linear Independent Component Estimation* (NICE) [41]. The main downside of the NICE transformations, which was solved by Real NVP, is that they do not scale the volume – they have $|\det \mathbf{J}| = 1$.

The main idea of the Real NVP is to split the input vector $\mathbf{x}$ into two vectors (so-called *channels*) $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2)$ and do only trivially invertible operations between these channels, such as addition and multiplication. Particularly, a single Real NVP transformation $f(\mathbf{x}) = f(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{y}_1, \mathbf{y}_2) = \mathbf{y}$ can be mathematically expressed as:

$$\begin{aligned}\mathbf{y}_1 &= \mathbf{x}_1 \\ \mathbf{y}_2 &= \mathbf{x}_2 \odot \exp\left[S(\mathbf{x}_1)\right] + T(\mathbf{x}_1),\end{aligned} \tag{66}$$

where $\odot$ is the Hadamard product, i.e. element-wise multiplication, exponentiation is also done element-wise and S, T are transformations $\mathbf{x}_1 \in \mathbb{R}^m \to \mathbb{R}^{n-m} \ni \mathbf{x}_2$, where n is the dimension of the (physical) system. Obviously, S stands for *scaling* and T for *translation*. There are many ways to split vector $\mathbf{x}$ into the channels and this choice is not considered to be crucial. In the BG, channels contain odd and even elements, respectively, i.e.

$$\mathbf{x}_1 = (x_1, x_3, \ldots)^\top \quad \mathbf{x}_2 = (x_2, x_4, \ldots)^\top.$$

This is motivated by the fact that for the BGs vector $\mathbf{x}$ is defined by (46) and we want to mix highly correlated coordinates of the same atoms.[11] Nevertheless, for theoretical purpose, indices of vector elements are only a matter of choice and thus we will use a simpler method of splitting the input into channels in this section:

$$\mathbf{x}_1 = \mathbf{x}_{1:m} \equiv (x_1, \ldots, x_m)^\top \qquad \mathbf{x}_2 = \mathbf{x}_{m+1:n} \equiv (x_{m+1}, \ldots, x_n)^\top,$$

where we use $\mathbf{x}_{i:j}$ to denote vector slices.

The key aspect of the Real NVP transformation is that S and T can be arbitrarily complex and the transformation will be still easily invertible with a tractable determinant of Jacobian matrix. Indeed, the inverse transformation to (66) can be easily calculated:

$$\begin{aligned}\mathbf{x}_1 &= \mathbf{y}_1 \\ \mathbf{x}_2 &= (\mathbf{y}_2 - T(\mathbf{y}_1)) \odot \exp\left[-S(\mathbf{y}_1)\right].\end{aligned} \tag{67}$$

[11] The latter mentioned option of splitting the vector $\mathbf{x}$ into (approximately) two halves would result in each channel containing half of particles, which might not be optimal.

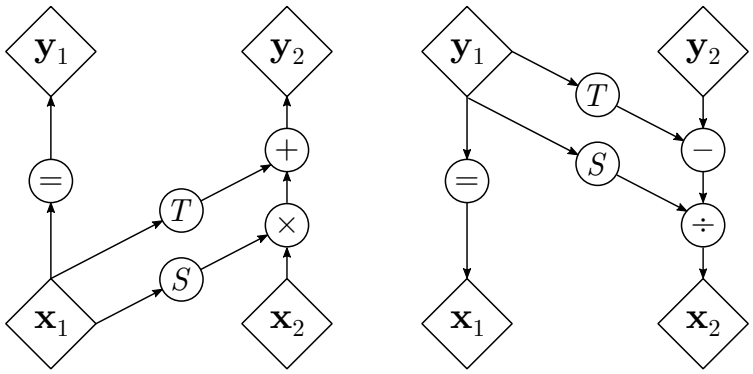

Figure 5: Forward (left) and backward (right) Real NVP transformation. Input vector $\mathbf{x}$ is divided into two channels and only one of them undergoes the transformation. Functions S and T can be arbitrarily complex; they do not have to be invertible nor have a tractable determinant. Adapted from [40].

Note that we did not need to invert S and T. The Jacobian matrix of the forward transformation (66) has the following form:

$$\mathbf{J}_f = \frac{\partial \mathbf{y}}{\partial \mathbf{x}^\top} = \begin{pmatrix} I_m & \mathbf{0} \\ \frac{\partial \mathbf{y}_1}{\partial \mathbf{x}_1} & \operatorname{diag}\left(\exp\left[S(\mathbf{x}_1)\right]\right) \end{pmatrix}. \tag{68}$$

Note that this matrix is triangular. Therefore, its determinant is simply a product of diagonal elements:

$$|\det \mathbf{J}_f| = \prod_i \exp\left[S_i(\mathbf{x}_1)\right] \quad \Rightarrow \quad \ln|\det \mathbf{J}_f| = \sum_i S_i(\mathbf{x}_1), \tag{69}$$

which is, again, easily calculated, even if we do not know $|\det S(\mathbf{x}_1)|$. Furthermore, the log-determinant of the inverted transformation is equal to:

$$\ln|\det \mathbf{J}_{f^{-1}}| = \ln|\det \mathbf{J}_f|^{-1} = \ln \prod_i \exp\left[-S_i(\mathbf{y}_1)\right] = -\sum_i S_i(\mathbf{y}_1). \tag{70}$$

This concludes our proof that the Real NVP transformation is easily invertible and has a tractable determinant.

However, the single Real NVP transformation alters only one channel of the input vector $\mathbf{x}$. Therefore, as we want both channels to undergo a transformation, two sequential Real NVP transformations with swapped channels are used as an elementary unit called a Real NVP *block* (also *layer*), see fig. 6. Therefore, the Real NVP layer has four transformations S_1, S_2, T_1, T_2. Formulae for both forward and backward transformations of the layer, together with determinants of their Jacobians, could be derived without difficulty using equations (65), (66), (67), (69) and (70).

Note that defining these four transformations $S_{1,2}$, $T_{1,2}$ corresponds to describing behavior of the whole layer, as they are the only variable part of it. In practice, S and T transforms are implemented as deep feedforward networks. By default, ReLU is used as an activation function for T networks and hyperbolic tangent for S networks. However, no

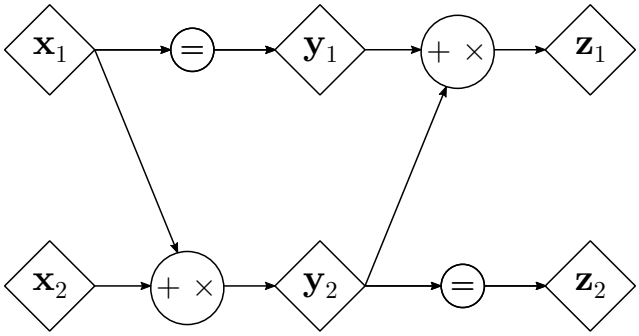

Figure 6: A Real NVP block (layer) composed of two Real NVP transformations. The channel which remains untouched in one transformation is modified in the second one. Adapted from [40].

activation is used in the output layer so that the output can span over both positive and negative numbers (recall that, for instance, the output of ReLU is always non-negative).

The transformation F of the BG consists of several Real NVP blocks in a row. Therefore, the vector $\boldsymbol{\theta}$ of BG's trainable parameters contains parameters of all S and T deep NNs, which are used in the Real NVP layers.

When the BG is trained, batches of examples $\mathbf{x}$ and latent vectors $\mathbf{z}$ are sampled from the training dataset and the normal latent distribution, respectively. These batches are transformed by F_{xz}/F_{zx}, i.e. when looking at fig. 7, $\mathbf{x}$-batch proceeds upwards and $\mathbf{z}$-sample downwards. Subsequently, the ML loss is evaluated using vectors $\mathbf{z} = F_{xz}(\mathbf{x})$ and KL loss, together with RC loss, are calculated using $\mathbf{x} = F_{zx}(\mathbf{z})$. After that, the gradient of total loss (62) is computed, as it was explained in section 2.2, with respect to the each element of $\boldsymbol{\theta}$. This way, the transformation F_{xz} is trained (and thus also F_{zx}, since it is defined by the very same set of the parameters).

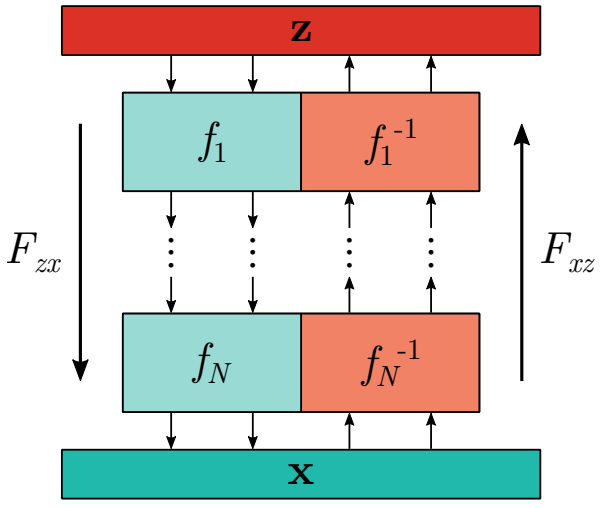

Figure 7: Transformations F_{zx} and F_{xz} of the Boltzmann generator. They consist of several Real NVP layers, which are represented by f_i. Naturally, both transformations depend on each other and have the same set of parameters $\boldsymbol{\theta}$. Samples undergo transformations split into two channels; this is illustrated by pairs of the arrows. Adapted from [2].

4 Our implementation of the Boltzmann generators

Problems regarding the original code

Authors of the Boltzmann generators have published a code used in their research article at web portal Zenodo [4]. It uses the Python programming language to implement the BGs together with auxiliary functionalities and the Jupyter Notebook to run the simulations. Although working correctly, we found out some issues regarding the code. Namely:

- It uses an ML library called TensorFlow (TF) [42]. However, it works only with version 1 and not with the newer TF 2.
- It does not follow PEP coding standards[12] for Python programs.
- It includes many functionalities that proved to be dead ends or there already exist better alternatives. For example, the code contains an implementation of NICE layers. Furthermore, it contains functions/methods with duplicate functionality.
- It contains only a small documentation. Moreover, the naming of variables and other objects is sometimes different from the one used in the BG article [2].

Because of these issues, the original code used by the authors is not very beginner-friendly. It took us a long time to understand it. Therefore, we have decided to build on this code and create our own fork of the repository.

We have started by adapting the code to the TF version 2. The key problem lied in the implementation of the loss functions. Let us give an example – the original definition of the ML loss:

```
def loss_ML_normal(y_true, y_pred):
    return -self.log_likelihood_z_normal(std=std)
```

It is not important to fully understand the code, one should just note that the function does *not* use the input parameters `y_true` and `y_pred`. On the contrary, it calls the method `log_likelihood_z_normal` which uses BG layers to calculate the loss. Therefore, the computational graph is not connected between the model outputs `y_pred` and the calculated loss, which causes the program to fail when running under TF 2.

We solved this by changing the way losses are implemented. We have created a new module `losses.py` where each loss (ML, KL and RC) is represented by its own class. The losses are then added to the model using the method `tf.keras.Model.add_loss`.

After that, we continued with the refactoring of the code. We removed everything that was not needed in the context of our thesis so that the code was as lightweight as possible. By reducing the code complexity, we intended to lower the time needed to get familiar with it. Readability of the code was further enhanced with comments, documentation and renaming of some objects.

[12] https://www.python.org/dev/peps/pep-0008/

Brief introduction to our code

The resulting code, which we use in the thesis, is freely accessible at GitHub.[13] In the following text, we are going to provide a concise introduction to the code in our repository. For more information, we refer the reader to documentation present in the code.

The main class is the BoltzmannGenerator (further as `bg`; previously named "EnergyInvNet"), defined in the `boltzmann_generator.py`, which implements all BG features described in the previous section. Its main methods are:

- `bg.train` encapsulates all training methods – by energy, by example or using the RC entropy. It is able to handle any combination of these methods, i.e. any linear combination of the weights in eq. (62). This method executes desired number of training iterations, while each of them comprises of sampling a batch of examples $\mathbf{x}$ from the training dataset, sampling latent vectors $\mathbf{z}$ from the normal distribution, transforming these batches with F_{xz}/F_{zx}, calculating the total loss and, finally, updating the model parameters $\boldsymbol{\theta}$. If a validation dataset is provided, a validation loss is calculated as well.

- `bg.transform_zx_with_jacobian` (the same also for $\mathbf{x} \to \mathbf{z}$) takes a batch of latent vectors as an input and returns the batch of transformed vectors $\mathbf{x} = F_{zx}(\mathbf{z})$ together with $\ln R_{zx}(\mathbf{z})$ for each vector in the input batch.

- `bg.sample` samples the latent normal distribution, transforms samples through F_{zx} and returns:

  1. Generated sample of latent vectors.
  2. Corresponding vectors $\mathbf{x} = F_{zx}(\mathbf{z})$ from the real configurational space.
  3. Energies $\frac{1}{2}\|\mathbf{z}\|^2$ in the latent space.
  4. Reduced energies $u(\mathbf{x})$ in the real space.
  5. The logarithm of samples' statistical weights $w(\mathbf{x})$.

In order to create an instance of the BoltzmannGenerator, one must specify the number and type of layers to be used in the BG in form of a string. Furthermore, the so-called *energy model* has to be provided, which will be discussed later. In our code, only Real NVP layers are implemented and they are represented by the letter "R".[14] For example, `BoltzmannGenerator("RRRR", energy_model)` creates a BG with four Real NVP layers. Additional layers, which split vectors into the channels and then merge them again, are augmented automatically.

The energy model represents the physical system that we want to sample by BG. Its name originates from the fact that it provides a function for the reduced potential energy $u(\mathbf{x})$. More specifically, the energy model must be an object with:

[13] https://github.com/auhliarik/boltzmann-generators

[14] However, it is possible to easily add new layers in the future. They must have methods `connect_xz`, `connect_zx` used in the creation of the model and properties (methods marked with decorator `@property`) `log_det_Jxz`, `log_det_Jzx` if the layer transformation scales the volume elements.

- Integer attribute `dim`, which describes a physical dimension of the system. In other words, $\mathbf{x} \in \mathbb{R}^{\mathrm{dim}}$.
- Function `energy` which returns array of energies $u(\mathbf{x})$ given the sample of configurations $\mathbf{x}$. Both input and output must have the form of a NumPy[15] array.
- Function `energy_tf` which does the same as the previous one, but input and output are instances of `tf.Tensor`.

Moreover, TensorFlow must be able to calculate the gradient of the last-mentioned function with respect to its inputs since it is used in the calculation of $\nabla_{\boldsymbol{\theta}} J$ during the training. This restricts the functions that can be used in the definition of `energy_tf`. The first available option is to use only functions from the TF library, which can be automatically differentiated. The second option is to call arbitrary functions but, in that case, the gradient must be provided by hand. This will have to be taken into consideration when external software, such as LAMMPS [43], will be used for calculation of energies.[16]

Algorithm 2: Algorithm for calculation of the RC entropy used in the BG training

Input: $r_{\min}, r_{\max}$ – bounds of the RC value
Input: $\mathbf{X}$ – batch matrix of vectors $\mathbf{x}$, i.e. one line for each configuration
Input: $r(\cdot)$ – function that calculates value of the RC
$N \leftarrow 11$;
$\sigma_{\mathrm{RC}} \leftarrow (r_{\max} - r_{\min})/N$;
$\mathbf{g} \leftarrow$ vector of N equidistant points in the interval $[r_{\min}; r_{\max}]$;
Calculate the value of the RC for each configuration in a batch:
$\mathbf{r} \leftarrow r(\mathbf{X})$;
Place centers of unnormalized Gaussians with $\sigma = \sigma_{\mathrm{RC}}$ to points $\mathbf{r}$ and evaluate them at points $\mathbf{g}$:
$\mathbf{G} \leftarrow \exp\left[-(\mathbf{g}^\top - \mathbf{r})^2/2\sigma_{\mathrm{RC}}\right]$;
where $\mathbf{G}$ has one line for each configuration in the batch and N columns that hold values of the Gaussian in points $\mathbf{g}$.
Normalize each row of $\mathbf{G}$, i.e. each Gaussian:
$\mathbf{G} \leftarrow$ `normalize_rows`$(\mathbf{G})$;
Sum the columns – values of all Gaussians at individual points of $\mathbf{g}$:
$\mathbf{p} \leftarrow$ `sum_columns`$(\mathbf{G})$;
Normalize the histogram we obtained:
$\mathbf{p} \leftarrow \mathbf{p}/\|\mathbf{p}\|$;
Calculate the entropy of this histogram (probability mass function):
$H \leftarrow$ `sum`$(\mathbf{p} \ln \mathbf{p})$;
Output: H

[15] https://numpy.org/doc/stable/
[16] In this case, one may use that $\nabla_i u(\mathbf{x}) = -\mathbf{F}_i(\mathbf{x})/k_B T$, where F_i is the force acting on the i-th particle.

In the beginning of the training, samples $\mathbf{x}$ obtained by sampling using the BG are mostly random vectors. Because of this, samples may have very large energies and there is a risk of numeric overflows. Therefore, the following regularization is applied to the reduced energy $u(\mathbf{x})$ when calculating the KL loss (57) [3]:[17]

$$u_{\mathrm{reg}}(\mathbf{x}) = \begin{cases} u(\mathbf{x}) & u(\mathbf{x}) < E_{\mathrm{high}} \\ E_{\mathrm{high}} + \ln(u(\mathbf{x}) - E_{\mathrm{high}} + 1) & E_{\mathrm{high}} \leq u(\mathbf{x}) < E_{\mathrm{max}} \\ E_{\mathrm{high}} + \ln(E_{\mathrm{max}} - E_{\mathrm{high}} + 1) & E_{\mathrm{max}} < u(\mathbf{x}) \end{cases} \tag{71}$$

where E_{high}, E_{max} are constants. In other words, identity function switches to the logarithm after E_{high} and E_{max} determines the upper bound for the potential-energy values.

Implementation of some algorithms

Herein we would like to discuss two algorithms that we took over from the original code, but we want to familiarize the reader with their implementation for better understanding and usage of the BGs.

Algorithm 3: Bootstrapped free-energy calculation

Input: $\mathbf{X}$ – matrix of vectors $\mathbf{x}$ sampled from the BG, i.e. one line for each configuration
Input: $\mathbf{w}$ – vector of statisical weights correspoding to configurations in $\mathbf{X}$
Input: N – number of configurations in $\mathbf{X}$, i.e. number of its lines
Input: $r(\cdot)$ – function that calculates value of the RC
Input: $r_{\mathrm{min}}, r_{\mathrm{max}}$ – bounds of the RC value
Input: N_b – number of bootstrapped samplings to be performed

Create a 2D array for storing free-energy profiles:
$\mathbf{A} \leftarrow$ empty 2D array ;
for $i = 1, \ldots, N_b$ **do**
  Randomly sample from $\mathbf{X}$ with allowed resampling of configurations:
  $\mathbf{S} \leftarrow N$ samples from $\mathbf{X}$;
  Select weights corresponding to samples in $\mathbf{S}$ from $\mathbf{w}$:
  $\mathbf{w}_S \leftarrow$ `select_correspoding_weights`$(\mathbf{w}, \mathbf{S})$;
  Calculate values of RC for the sample:
  $\mathbf{r} = r(\mathbf{S})$;
  Split interval $[r_{\mathrm{min}}; r_{\mathrm{max}}]$ into bins and create a weighted histogram of r:
  $\mathbf{h} \leftarrow$ `create_weighted_histogram`$(\mathbf{r}, \mathbf{w}_S, r_{\mathrm{min}}, r_{\mathrm{max}})$;
  $\mathbf{h} \leftarrow$ `normalize_histogram`$(\mathbf{h})$;
  Calculate the free energy profile (up to factor $k_B T$):
  $\mathbf{A}$.`append_line`$(-\ln \mathbf{h})$;
Return mean value and standard deviation in each bin:
Output: `calculate_mean_of_columns`$(\mathbf{A})$
Output: `calculate_error_of_columns`$(\mathbf{A})$

[17] This is called the "*linlogcut*" in the code.

The first one is the calculation of the entropy along the RC. Naturally, the first idea could be to use a function provided by one of the Python libraries. However, as we already explained, since the RC entropy is used in the RC loss, TF must be able to calculate its gradient, which does not hold for the external functions. Therefore, the authors of the BG use a less precise but faster and simpler algorithm for the calculation of the RC entropy, which comprises only functions from the TF library. The algorithm is based on the kernel density estimation and is described in alg. 2.

The second algorithm is the *bootstrapped* free-energy calculation. Bootstrapping is a statistical technique that uses random sampling with replacement. In this case, it is used to estimate the error of the calculated FEP.

5 Testing the BGs on toy models and on solvated-dimer problem

Both during the development phase and after its end, we tested our code using very simple models to assure that our implementation was working correctly. 2D models are particularly suitable for this purpose as their low dimensionality allows us to easily visualize results and thus understand what is going on and also promptly reveal possible errors. We will use the following notation for the 2D systems:

$$\mathbf{x} = (x_1, x_2)^\top \equiv (x, y)^\top.$$

The simplest model we used was the bivariate normal distribution. If covariance matrix of this distribution is

$$\mathbf{\Sigma} = \begin{pmatrix} \sigma_X^2 & \rho\sigma_X\sigma_Y \\ \rho\sigma_X\sigma_Y & \sigma_Y^2 \end{pmatrix},$$

where ρ is the correlation between X and Y, then the corresponding reduced energy can be expressed as

$$u(\mathbf{x}) = \frac{1}{2(1-\rho^2)} \left[\frac{x^2}{\sigma_X^2} - 2\rho \frac{xy}{\sigma_X\sigma_Y} + \frac{y^2}{\sigma_Y^2} \right]. \tag{72}$$

Note that this potential does not even have any barriers and has only one, global, minimum. Training dataset can be produced simply by direct sampling of the distribution, for instance using the NumPy software package. We used this model to make sure that the basic functionalities of the BG (sampling, training, etc.) work the way we expect them to.

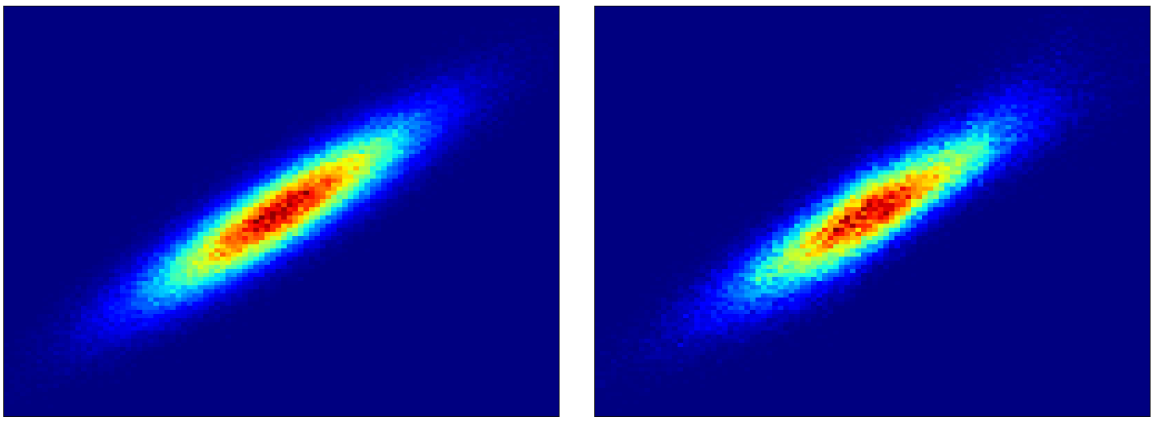

Figure 8: Training the BG to sample the bivariate normal distribution. *Left:* 2D histogram of the training dataset. *Right:* histogram of samples obtained using the BG after the training.

5.1 Double-well model

After becoming convinced that our BGs are fully functional, we replicated results of the BG article [2] regarding 2D toy models. The first model like this is called the *double well*,

since it is defined by the potential with two minima:

$$U(x,y) = \frac{1}{4}ax^4 - \frac{1}{2}bx^2 + cx + \frac{1}{2}dy^2, \tag{73}$$

with $k_BT = 1$, i.e. $u(\mathbf{x}) = U(\mathbf{x})$. This is the same potential as the one used in the BG article [3]. Although authors of the article state the same formula for the potential energy as we do, however, they use in reality a different set of parameters a, b, c, d than they claim. We use $a = 4$, $b = 12$ and $c = d = 1$, which are the parameters truly used in the article. This discrepancy between *stated* and *real* formulae and parameters of the potential energy is present in the supplementary materials of the BG article for all models we will discuss in this section. The landscape of the double-well potential can be seen in fig. 9.

Note that since $u(\mathbf{x}) = u_1(x) + u_2(y)$ the Boltzmann probability distribution can be factorized to two terms – one in x and one in y. Therefore, when FEP along the $x \equiv x_1$ is calculated, the entropy term, depending only on y, is constant and thus $A(x) = u_1(x) + \text{const}$.

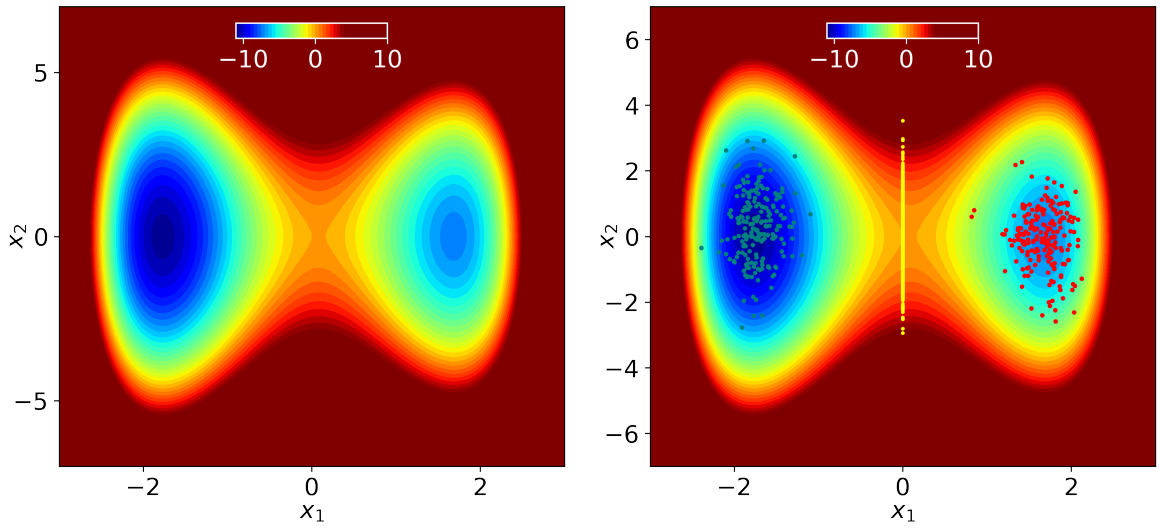

Figure 9: *Left:* Landscape of the 2D double-well potential. *Right:* Training dataset used in the training of the BG. It includes states from the left (blue) and right (red) minimum of the potential. Transition states (orange) are showed only for illustration purposes and are *not* used in the training.

The training set was generated with two *disconnected* MCMC simulations. Proposal steps are generated by sampling a multivariate normal distribution as was already explained in section 1.2. This method of creating the acceptance moves is used also for all other models discussed in the thesis. The normal distribution is isotropic with the standard deviation of $\sigma_{\text{metro}} = 0.1$.

Each simulation starts in one of the two potential-energy minima and runs for 10 000 steps, of which 1000 are saved. This way, the training dataset with 2000 configurations is created, see fig. 9. The training dataset does not contain any configurations from the transition (middle) part of the configuration space. Once again, supplementary materials

of the BG article incorrectly state that 1000 training configurations are used. Therefore, we are not going to further point to this kind of inconsistencies unless we want to emphasize them.

We use the BG with 4 Real NVP layers (further marked as "R^4"). All S and T transformations in these layers have $l_{\text{hidden}} = 2$ hidden layers with $n_{\text{hidden}} = 100$ neurons each. It becomes apparent that the best way to start the training of the BG is to use *solely* training by example. This helps the BG to focus on the important parts of the configuration space. As of now, this recipe seems to have general applicability and is used for all models in the thesis.

After this initial training stage, we proceed in the same way as the authors of the BG article and train the BG multiple times using four different training schedules for the higher stages of the training, as described in table 3. Results obtained this way are shown in figure 10.

Mark	Iterations	Batch size	w_{ML}	w_{KL}	w_{RC}
ML	400	2000	1	0	0
KL	400	2000	0	1	0
ML + KL	400	2000	1	1	0
ML + KL + RC	200	2000	1	1	1
	200	2000	0.01	1	5

Table 1: Different training methods used to train the BG for the double-well problem. Number in the second column represents the number of the training iterations. The last three columns contain weights of individual losses. In *all* cases the training starts with 200 iterations with $w_{\text{ML}} = 1,\ w_{\text{KL}} = w_{\text{RC}} = 0$ using batch size 500, which is the first *stage* and it is not present in the table. Parameters stated in the table have been used in the second stage. The method "ML + KL + RC" consists in total of three stages but the overall number of training iterations is the same. The learning rate is 10^{-3} for all methods during the whole training. Constants $E_{\text{high}},\ E_{\text{max}}$ of the energy regularization (71) are set so high that the logarithmic part of the function is never activated during the training.

The low dimension of the problem helps us visualize the results that constitute a very educative insight into the BGs. The first row of the figure represents the case when the training by energy is used in the second stage as well as in the first one, i.e. only this single method is used throughout the training. In this case, the only way that potential energy comes into the training is that it was used in the MCMC simulations which produced the training dataset. Therefore, since the training dataset consists of two *disconnected* simulations and training by energy is not used, *the BG cannot learn the ratio of Boltzmann probabilities for configurations coming from different simulations* (metastable states).

This results in splitting the latent space into two halves, each one occupied by states from the different potential-energy minimum. When we calculate the free-energy profile without reweighting the samples (the third column), we get the same free energy in the two minima, which agrees with the presented ideas. Nevertheless, this can be solved by assigning the statistical weights to the samples, although one can not rely on this when dealing with more complicated models.

To sum up, when w_{ML} is set to a high value during the training, F is forced to forget the Boltzmann probabilities of the configurations and learns to produce them with their probability in the training dataset instead. In general, these two are not the same if the training dataset consists of multiple disconnected simulations. In our opinion, this fact is not positive neither negative as it can be utilized in the training but also be a source of problems.

Results in the second row have been obtained by putting emphasis on the training by energy (the KL loss) in the training. Although the KL loss (57) contains an entropy term that penalizes mapping large portions of the latent space into small volumes in the real space, this effect is not strong enough to prevent the BG from doing so. As was already observed by the authors of the BG, training by energy has a tendency to focus too much on the region with the lowest potential energy. This can be seen also in fig. 10, where the configurations from the left (lower) energy minimum occupy the region around the origin of the latent space. The resulting free-energy profile before the reweighting estimates higher free energy in the right minimum than the real value. Reweighting is able to solve this issue but the error of the FEP in the right minimum is still higher than the one obtained by training solely with an example.

A convenient combination of the ML and KL losses performs generally much better than the two previous methods. However, for this system, its results are comparable to the first method. Furthermore, without using the reaction coordinate in the training, we are not able to reconstruct the whole FEP between the metastable states. We are able to succeed in this goal only if we use also the RC loss with RC being set to the x coordinate, i.e. $r(\mathbf{x}) = x_1$. This promotes sampling along the RC, as it increases the RC entropy.

The authors of the BG article were interested in the *latent interpolations* between the configurations from the training dataset, since it is used in other fields [44]. The latent interpolations for the BGs are carried out in the following way:

1. Two random configurations $\mathbf{x}_1$, $\mathbf{x}_2$ from the training dataset are used – each from a different metastable state of the system.

2. These two configurations are mapped to the latent space: $\mathbf{z}_1 = F_{xz}(\mathbf{x}_1)$, $\mathbf{z}_2 = F_{xz}(\mathbf{x}_2)$.

3. After that, the path of the linear interpolation is constructed in the latent space:

$$\Gamma_Z = \mathbf{z}_1 + \lambda(\mathbf{z}_2 - \mathbf{z}_1) \quad \text{where} \quad \lambda \in [0; 1]. \tag{74}$$

4. This linear latent path is then mapped back to the real space: $\Gamma_X = F_{zx}(\Gamma_Z)$.

It turns out that the latent interpolations result in low-energy transition paths between the metastable states, see fig. 11. At the moment, we are not aware of any theory that would be able to describe this behavior. The only information that we have is an observation that when some normalizing-flow architectures are used for generating human faces, individual latent variables z_i tend to represent well-defined features of the faces. In other words, changing one of the variables z_i, while keeping values of the other ones, often results in the face changing, for example, its extent of a smile or skin color [40, 44]. We will discuss latent interpolations in more detail later.

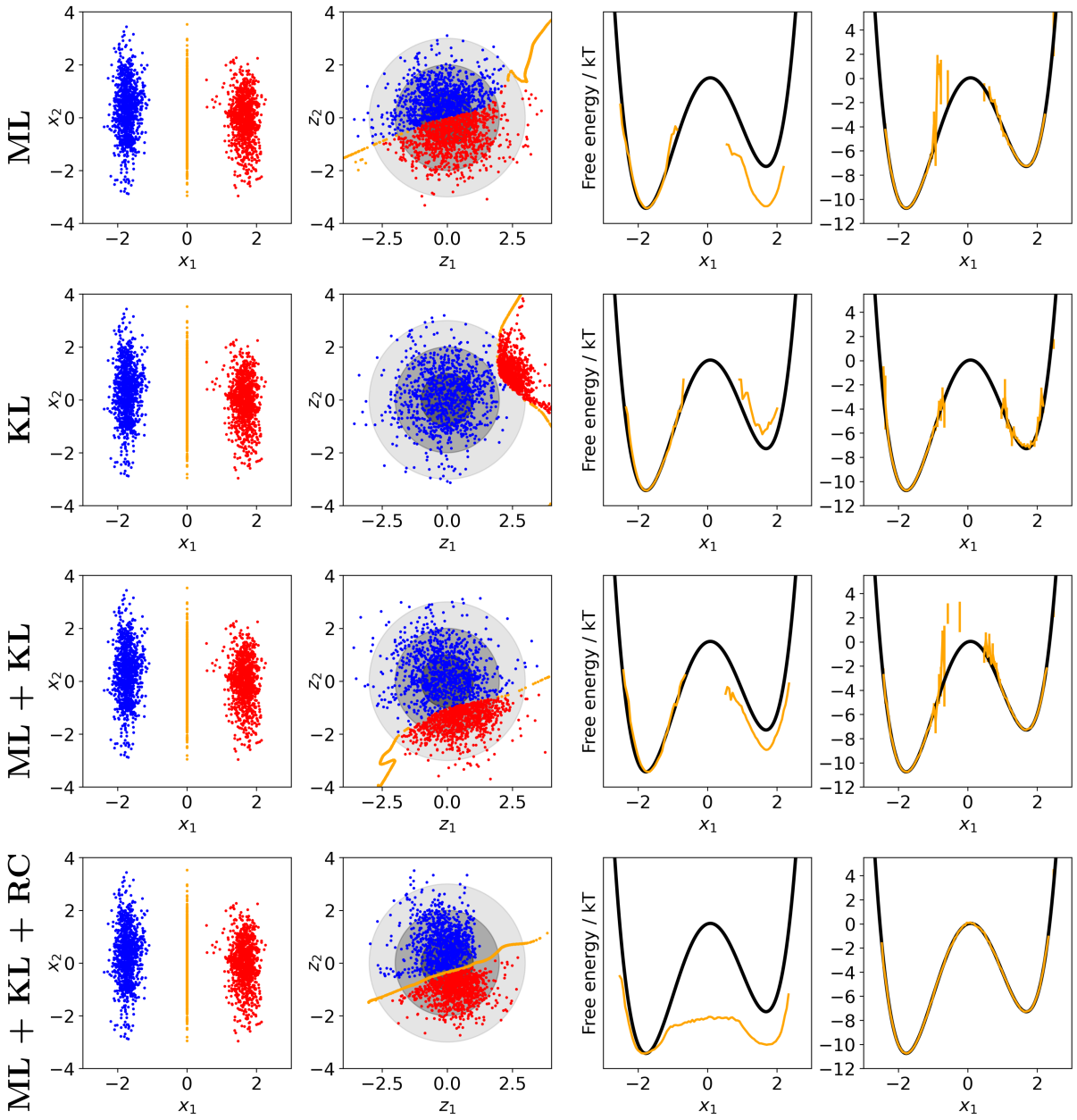

Figure 10: Results obtained after training the BG using different methods described in table 3. *First column:* The training dataset (blue and red dots) together with transition states (orange) that were not used in training and are present only for illustrative purposes. This is the same for all methods. *Second column:* Visualization of the latent space – the training dataset and transition states mapped from the real to the latent space with F_{xz}. Circles represent areas with $(1,\ 2,\ 3) \times \sigma_Z$ of the latent normal distribution. *Third column:* Free-energy profile obtained as $-\ln p(r)$ where $r(\mathbf{x}) = x_1$ and $p(r)$ is the histogram of r obtained *without* using the statistical weights (63) for the samples $\mathbf{x}$. *Fourth column:* FEP obtained in the same way as for the previous column but with samples reweighted using the mentioned statistical weights. Black and orange lines represent true and calculated FEP, respectively. Errors are obtained by bootstraping (see algorithm 3) and indicated by orange vertical bars. Result of the FEP calculation is not shown for bins whose total statistical weight is lower than the preset cutoff. Note that this is the only place of the thesis where we present FEPs calculated without the reweighting. In the following text, all FEPs will be calculated *with* reweighting as it is the standard way.

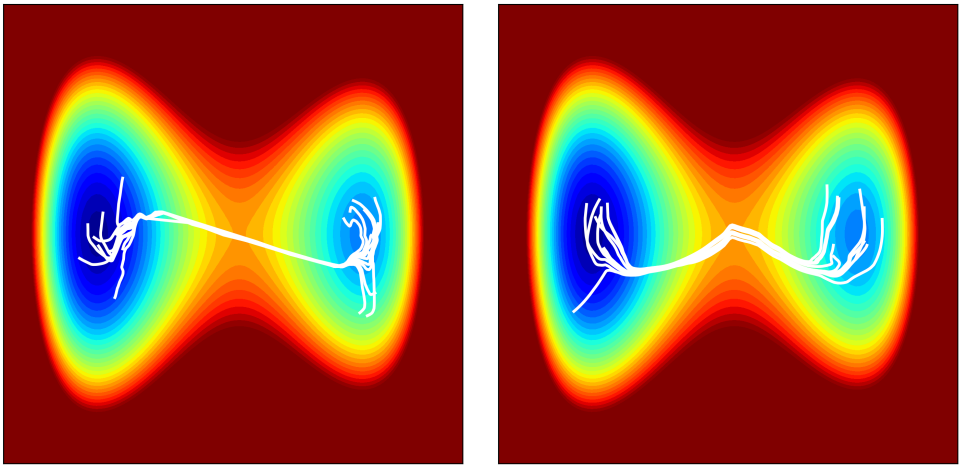

Figure 11: Ten paths (white) obtained by the latent interpolations between random configurations from two metastable states of the double-well potential. *Left:* BG trained without using the RC entropy. *Right:* BG trained using the RC loss.

5.2 Mueller potential

The second 2D toy-model potential used in the BG article is the *Mueller potential* [45, 46], which is used as a benchmark for the transition paths finding algorithms. It is defined as:

$$u(x,y) = \alpha \sum_{i=1}^{4} A_i \exp\left[a_i(x-\hat{x}_i)^2 + b_i(x-\hat{x}_i)(y-\hat{y}_i) + c_i(y-\hat{y}_i)^2\right], \tag{75}$$

with $\alpha = 1$ and other constants given in table 2.

i	A_i	a_i	b_i	c_i	$\hat{x}_i$	$\hat{y}_i$
1	-200	-1	0	-10	1	0
2	-100	-1	0	-10	0	0.5
3	-170	-6.5	11	-6.5	-0.5	1.5
4	15	0.7	0.6	0.7	-1	1

Table 2: Parameters of the Mueller potential (75).

The landscape of this potential is more complicated than the previous one, see fig. 12. It has three minima with a narrow "valley" that separates them. Therefore, finding a low-energy transition path is a demanding task.

The training dataset was produced by merging configurations from two MCMC simulations which have run in leftmost and rightmost minimum, respectively. Each simulation used $\sigma_{\text{metro}} = 0.1$ and was 1000 steps long while saving each tenth one. Therefore, the whole training dataset consists of 200 configurations. Transition states from the central minimum of the PES are displayed only for illustration and were, similarly to the double well, not used in the training. The MCMC simulation that starts in the right minimum tends to discover the central minimum and stay there for a certain number of steps. When

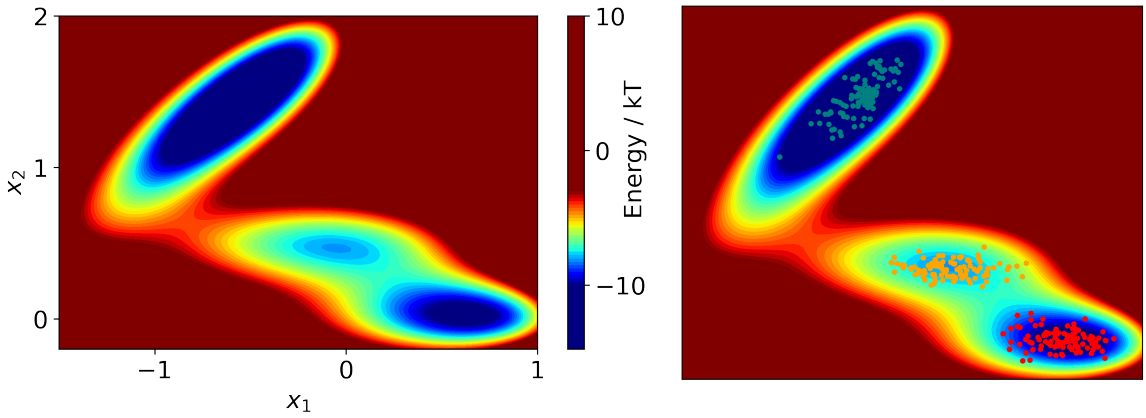

Figure 12: *Left:* landscape of the used 2D Mueller potential. *Right:* the training dataset that comprises configurations from leftmost (blue) and rightmost (red) minima of the potential. Configurations from the central minimum (orange) are shown only for illustration and are not used in training.

this happened, we discarded the simulation since we did not want the training dataset to contain these configurations.

We used the following parameters of the BG: R^5 architecture with $l_{\mathrm{hidden}} = 2$ and $n_{\mathrm{hidden}} = 100$. This results in the BG with 208 000 trainable parameters. Note that despite the statement by Noé *et al.* that they are using $l_{\mathrm{hidden}} = 3$ for this system [3], they are in fact using 2, just like we do. This is caused by the fact that they are always erroneously referring to the number of the hidden layers together with the output layer.[18] Therefore, one must always subtract 1 from their number of l_{hidden} to get the correct value.

Mark	Iterations	Batch size	w_{ML}	w_{KL}	w_{RC}	ε
ML + KL	100	500	1	0	0	10^{-3}
	300	1000	1	1	0	10^{-4}
ML + KL + RC	100	1000	1	0	0	10^{-3}
	200	4000	1	1	3	10^{-4}
	200	4000	0.1	1	3	10^{-4}

Table 3: Training parameters used for training the BG for the Mueller potential. "ML + KL" stands for training by energy and example while "ML + KL + RC" uses also the RC loss. The last column contains the learning rate used in the corresponding stage. Energy regularization for the KL loss uses $E_{\mathrm{high}} = 10^6$ and $E_{\mathrm{max}} = 10^{10}$ but energy values stay in the linear regime at most times.

At first, we started the training with the schedule which was used in the BG article. Nevertheless, we could not achieve the quality of the results presented in the article. Our main problem was a significant variance of the results for consecutive training runs. Although the training of a deep NN is a stochastic process, we considered the amount of stochasticity we experienced to be very unusual.

[18] See their implementation of functions `invnet` in module `invertible.py` and `nonlinear_transform` in `layers_basic.py`.

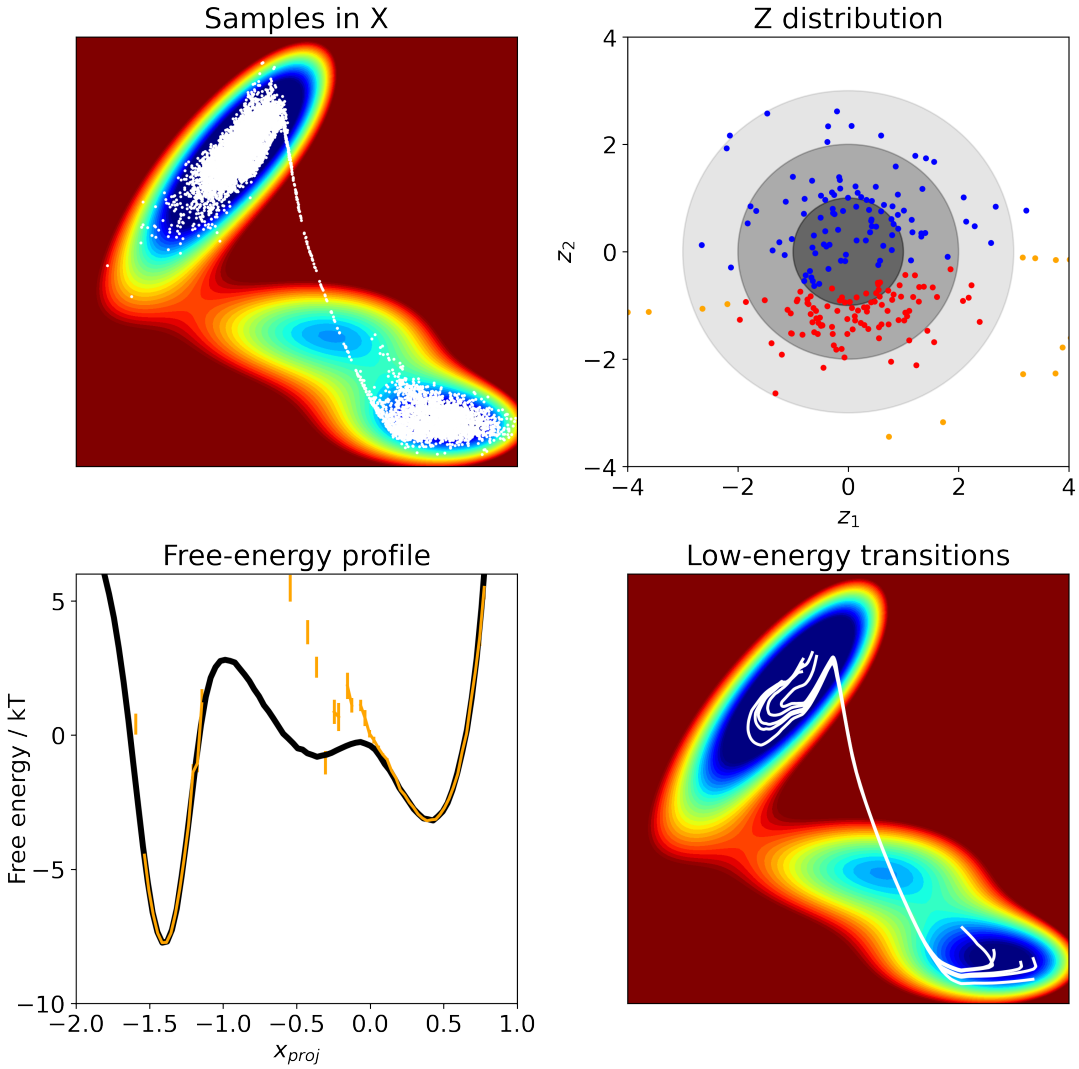

Figure 13: Results for the Mueller potential obtained by training *without* the RC loss. *Top left:* Samples from the BG (white dots). *Top right:* Visualization of the latent space – transformed configurations from the training dataset and the central potential-energy minimum. *Bottom left:* Calculated FEP (orange) along the projection $(1, -1)^\top/\sqrt{2}$. Note that this is a different projection than the one used as the RC. The value of the calculated FEP is not shown for the bins with errors exceeding the preset cutoff. The reference FEP (black) was calculated by a massive uniform sampling of the area after which a normalized histogram of $r(\mathbf{x})$ was computed with samples weighted by the Boltzmann exponential factor. *Bottom right:* 5 best transition paths selected from 100 latent interpolations between random configurations. Note that when no RC is used, the BG cannot calculate the full free-energy profile and linear interpolations lead through the region of very high energy.

Therefore, we decreased the learning rate (i.e. slowed down the training) and increased the batch size to suppress the fluctuations. Furthermore, for the second method which uses the RC loss, we split the training into three stages and adjusted loss weights.

With the improved training schedule, we have been able to replicate the results of the article. However, while results of different runs without the RC loss seem alike, when training using the RC, the results of consecutive runs differ significantly. There is approx-

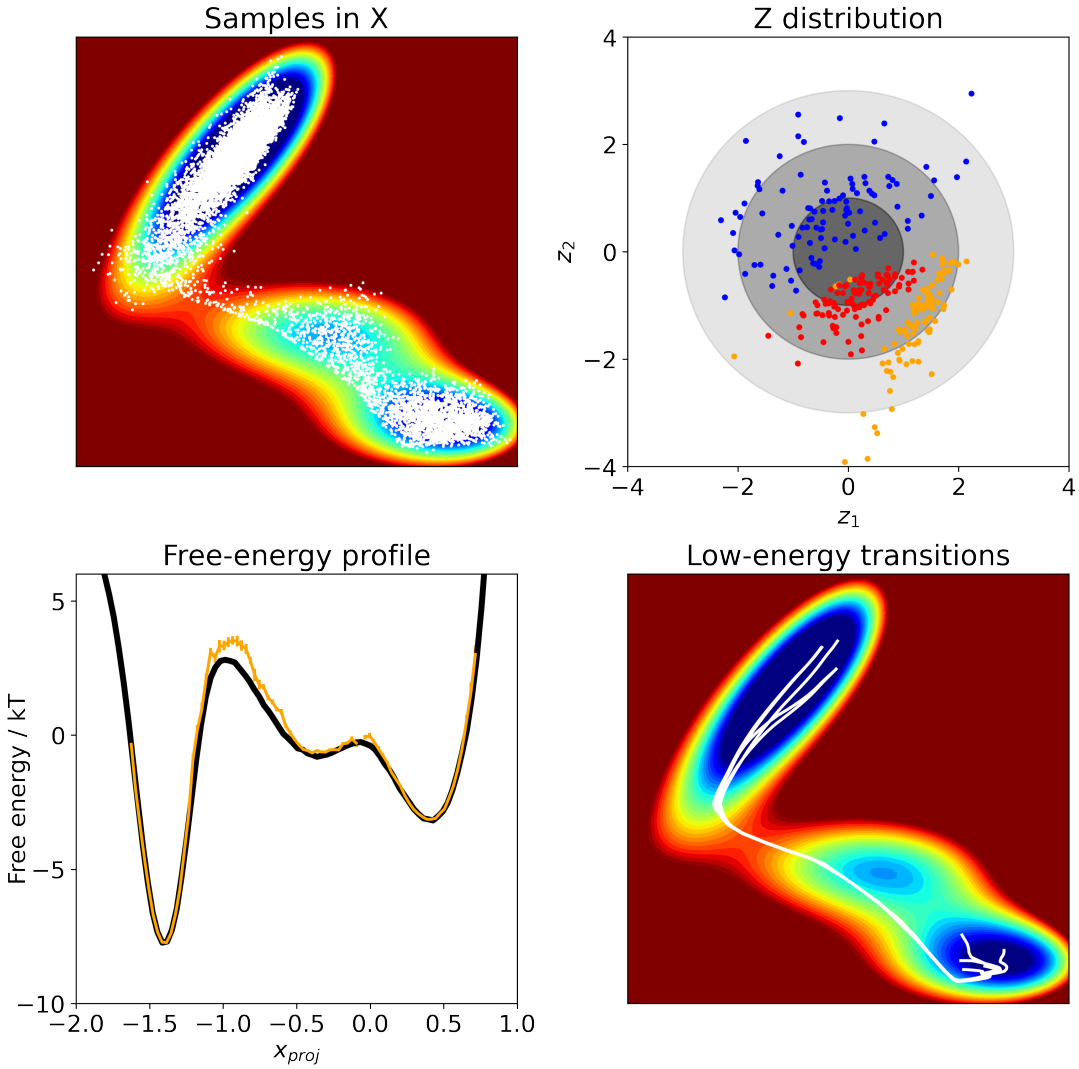

Figure 14: Results for the Mueller potential obtained by training *with* the RC loss. See caption of figure 13 for explanation of the images.

imately a one in five chance to obtain results comparable to the ones shown in fig. 14. Therefore, precise calculation of the full free-energy profile and optimal transition paths between metastable states should not be considered guaranteed to acquire. Nevertheless, we admit that there may be such a training schedule that would enable achieving this consistently.

When trying to optimize the learning process, we observed that there is one key aspect that substantially affects the quality of latent-interpolation paths: whether the BG learns to sample configurations along the optimal path or not. For example, for the Mueller potential, the transition paths lead through the low-energy "valley" between the minima *only* in the case that the BG tends to sample configurations from this area. In other words, the BG must *discover* this region of the configuration space. Otherwise, if the BG does not sample the configurations along the low-energy transition path, the *linear* latent interpolation results in a *linear* transition path which may lead through the low-energy region (double well) or directly through the barrier (Mueller potential) depending on the topology of the free-energy landscape, see figures 11, 13 and 14.

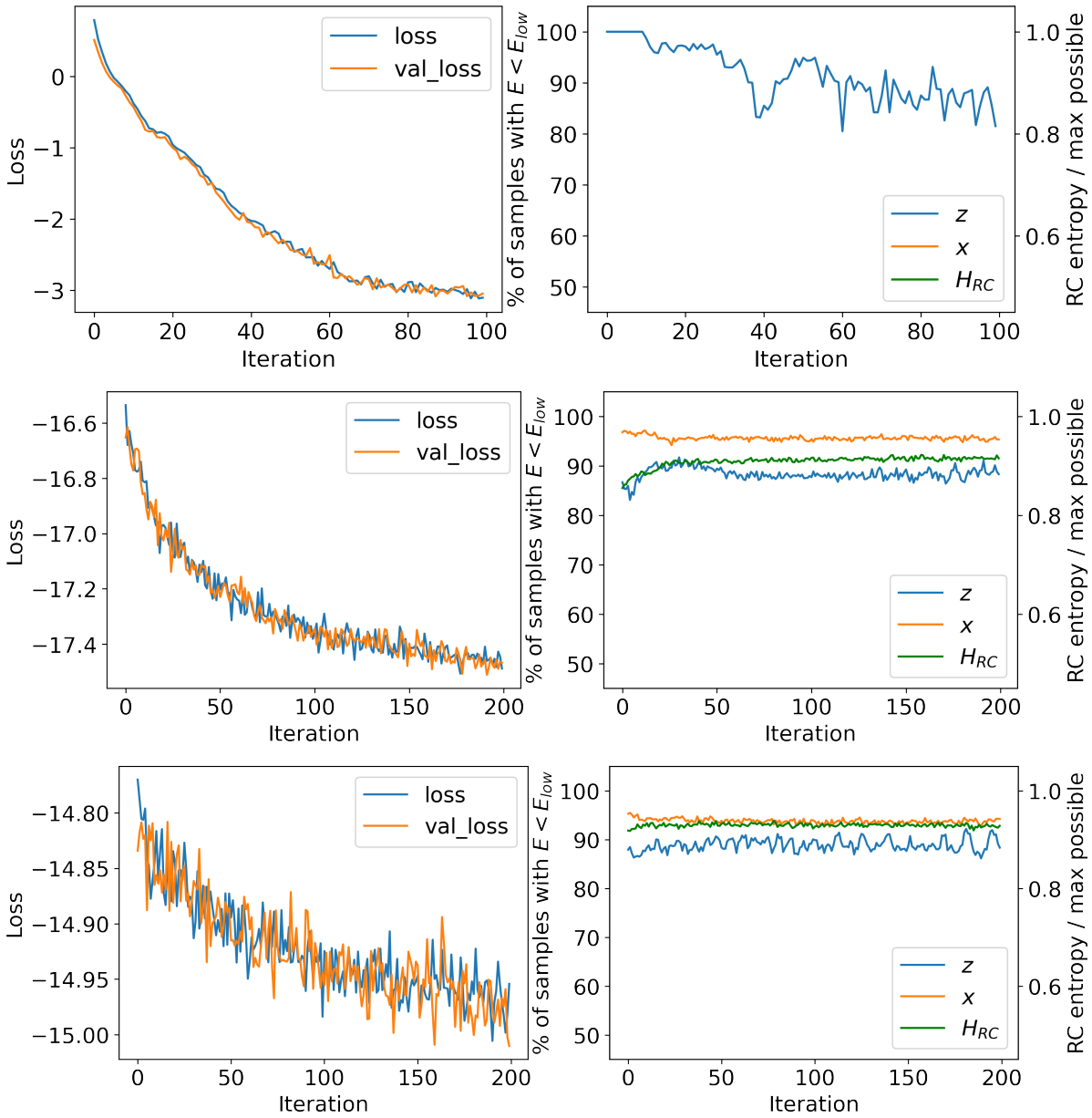

Figure 15: Learning statistics for a BG trained for the Mueller potential using the training schedule marked as "ML + KL + RC" in table 3. Each row corresponds to one training stage. *Left column:* Total training (blue) and validation (orange) loss. The validation dataset was produced running separate MCMC simulations with identical parameters as for the training dataset. *Right column*: Ratio of low-energy configurations in the latent space (blue), real configuration space (orange) together with ratio of the actual and maximal possible RC entropy (green). See text for explanation.

Finally, in figure 15 we present statistics of the training process for illustration. We suggest monitoring the following quantities:

- Evolution of the training and validation losses – the total ones as well as values of the individual (ML, KL, RC) losses.

- Ratio of the configurations from the training dataset that have "low" energy in the

latent space, i.e. $u_Z(F_{xz}(\mathbf{x})) < E_Z$, where E_Z is the cutoff. The same for energy $u(F_{zx}(\mathbf{z}))$ of the samples in the real space produced by the BG.

- Ratio of the *actual* and *maximal possible* RC entropy. The latter corresponds to a uniform distribution. The idea to monitor this ratio comes from us.

Later we will discuss the challenge of setting the right training schedule based on these quantities.

5.3 Solvated bistable dimer

The following model is much more complicated than the previous ones. It consists of a dimer in a solvent enclosed by a solid 2D box. The dimer is comprised of two particles and has two stable positions. Therefore, we use the attribute *bistable* when referring to it. The potential energy of the dimer is a function of the distance of its particles. We further refer to this quantity as the *dimer distance*. The metastable stable state with a lower dimer distance is called the *closed* state and has slightly lower energy than the *open* metastable state, where dimer particles are further apart. The dimer potential energy is shown in fig. 16.

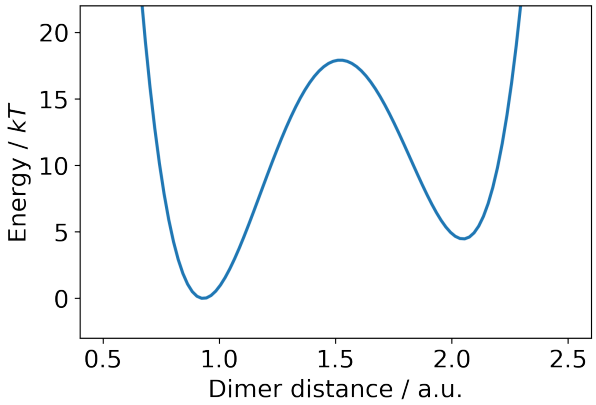

Figure 16: Potential energy of the bistable dimer as a function of the dimer distance.

The dimer can undergo a transition from the closed state to the open state (or backward). However, the dimer is surrounded by particles of the solvent with repulsive interactions and thus the system is *crowded* – a coordinated movement of the particles is needed to allow the dimer to switch its state. This results in a free-energy barrier that is several times higher than the temperature factor k_BT.

The system consists of two dimer particles and $N_S = 36$ solvent particles in two dimensions. Therefore, the dimension of the system is 76. We are going to use the following notation for the configuration vector $\mathbf{x}$:

$$\mathbf{x} = (x_{1x}, x_{1y}, x_{2x}, \ldots, x_{(2+N_S)x}, x_{(2+N_S)y})^\top, \tag{76}$$

i.e. the first index is the number of the particle and the second one denotes the axis. The first four coordinates in $\mathbf{x}$ represent the positions of the dimer particles. The formula for

the potential energy is given by:

$$\begin{aligned} U(\mathbf{x}) = &-4a(d-d_0)^2 + 16b(d-d_0)^4 + 2c(d-d_0) \\ &+ k_d(x_{1x}+x_{2x})^2 + k_d x_{1y}^2 + k_d x_{2y}^2 \\ &+ \sum_{i=1}^{N_S+2} 2k_{\text{box}}\Theta(-x_{ix}-l_{\text{box}}) + \sum_{i=1}^{N_S+2} 2k_{\text{box}}\Theta(x_{ix}-l_{\text{box}}) \\ &+ \sum_{i=1}^{N_S+2} 2k_{\text{box}}\Theta(-x_{iy}-l_{\text{box}}) + \sum_{i=1}^{N_S+2} 2k_{\text{box}}\Theta(x_{iy}-l_{\text{box}}) \\ &+ \varepsilon \sum_{i=1}^{N_S+1} \sum_{j=i+1;j\neq 2}^{N_S+2} \left(\frac{\sigma}{\|\mathbf{x}_i - \mathbf{x}_j\|}\right)^{12}. \end{aligned} \tag{77}$$

The first line of the formula corresponds to the dimer energy with d being the dimer distance. The second line represents the potential that attracts the dimer to the center of the box. The third and the fourth line describe the particle interaction with the box walls and use the Heaviside step function Θ. Here, l_{box} is half of the box side, i.e. its size is $2l_{\text{box}}$ long and its area is $4l_{\text{box}}^2$. Finally, the last line corresponds to repulsive interactions between particles.[19] Parameters of the potential are given in table 4.

As can be seen, the value of the $U(\mathbf{x})$ diverges when the distance of two particles approaches zero. This results in a very rough PES landscape with minima separated by very high barriers.

Note the unusual numeric constants in the first, third and fourth line that are powers of 2. These are used by the authors of the BG article, although they do not mention them in the BG supplementary materials.

d_0	a	b	c	k_d	l_{box}	k_{box}	ε	σ
1.5	25	10	2	20	3	100	1	1.1

Table 4: Parameters of the potential $U(\mathbf{x})$ given by (77) for the solvated bistable dimer model. Temperature is set to $k_B T = 1$.

The training dataset is produced with the MCMC simulations. First, we conducted a search for σ_{metro} with which the fastest sampling of the configuration space is achieved. We found out that $\sigma_{\text{metro}} = 0.2$ is optimal for this purpose. Let us recall that σ_{metro} defines the covariance matrix of the isotropic multivariate normal distribution $\mathcal{N}(\mathbf{0}, \sigma_{\text{metro}}^2 I_n)$ by whose sampling the proposal moves of the MCMC simulation are generated.

After that two MCMC simulations are run, with dimer in the closed and open state, respectively. The initial configuration is created by placing the dimer in a rectangular grid of solvent particles and, therefore, 10 000 MCMC steps are used for equilibration of the system. Subsequently, simulations run for 5×10^6 steps each, using the saving stride of 100, which results in $2 \times 5 \times 10^6/100 = 10^5$ total training configurations comprising both metastable states. No transition states are used in the training dataset as shown in fig. 17.

[19] Note that there is no repulsion between the two dimer particles. Their interaction is described solely by the first line of (77).

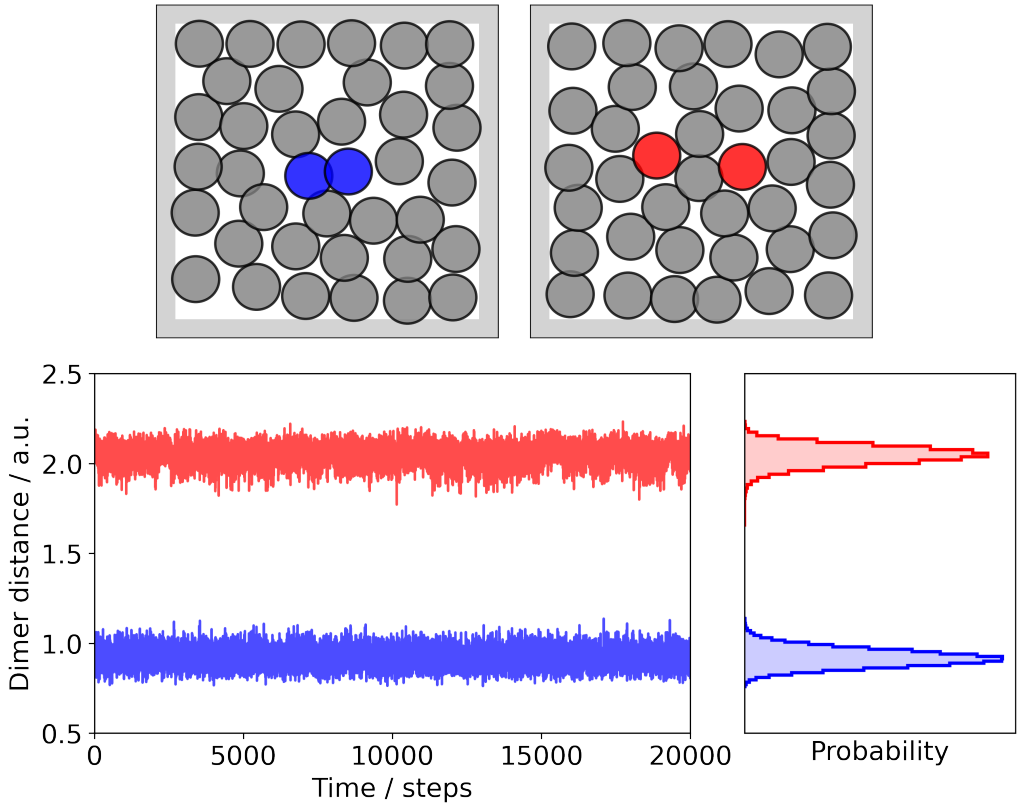

Figure 17: Properties of the training dataset for the solvated bistable dimer. *Top left:* Example of the closed configuration. *Top right:* Example of the open configuration. *Bottom left:* Dimer distance as a function of time (no. of steps) for MCMC simulations of the closed (blue) and open (red) dimer states (only part of the simulations is displayed). *Bottom right:* Histograms of the dimer distance for *whole* MCMC simulations that generated the training dataset.

However, there is one issue regarding the training dataset. Since the solvent is a *fluid*, its particles diffuse over time. This phenomenon is present also in the MCMC simulations. This effect rapidly increases the proportion of the configuration space occupied by the training dataset. Specifically, the expansion factor of the occupied hyperspace is proportional to $N_S!$.

It turns out that learning the BG to sample such a large volume of the configuration space is unfeasible. Therefore, authors of the BGs applied the so-called *Hungarian algorithm* [47] to the training dataset. Simply said, this algorithm removes the diffusion of the solvent particles.[20]

In particular, this algorithm is used in mathematics for solving the *linear sum assignment problem*, which is defined as finding the matrix $\mathbf{X}$ for which:

$$\arg\min_{\mathbf{X}} \sum_i \sum_j C_{ij} X_{ij}. \tag{78}$$

Here, the *cost* matrix $\mathbf{C}$ and $\mathbf{X}$ have the same size and there is a single 1 per each row and column of the $\mathbf{X}$, with other elements being zeros. If we switched min to max, we could think of this problem as maximizing the performance of a group of workers. In this case,

[20]The Hungarian mapper was applied to all particles except the two that constitute the dimer.

Stage	Iterations	Batch size	w_{ML}	w_{KL}	w_{RC}	ε	E_{high}
1	20	256	1	0	0	10^{-3}	10^4
2	200	8000	100	1	1	10^{-4}	10^4
3	300	8000	100	1	5	10^{-4}	10^4
4	300	8000	100	1	10	10^{-4}	10^4
5	1000	8000	20	1	10	10^{-4}	2×10^3
6	2000	8000	0.1	1	10	10^{-4}	10^3

Table 5: Training schedule used for the solvated bistable dimer. ε stands for the learning rate and E_{high} is one of the constants used in the reduced-energy regularization (71). E_{max} was set to 10^{10}. The schedule is almost identical with the one originally used by authors of the BG article. Our only change was replacing the weight of the ML loss from 0.01 to 0.1 in the last stage, as we were able to obtain better results this way.

C_{ij} would describe the performance of the i-th worker in the j-th task and X_{ij} would be equal to one if we assigned the worker i to the task j and zero otherwise. In other words, each worker is assigned to exactly one task and each task has exactly one worker assigned to it.

In the case of the bistable dimer, we minimize the result instead of maximizing it and assign "labels" (i.e. particle ID numbers) instead of tasks. We choose one configuration of the solvent particles as a reference one. Cost-matrix elements C_{ij} can be then defined as the squared distances of the i-th particle from the j-th particle in the reference configuration. By solving (78), we want to minimize the distance of the new configuration from the reference one in the configuration space. This can be imagined in a way that if two particles exchanged their positions, we would map them back by exchanging their labels and thus we are removing the diffusion. The Hungarian algorithm (also referred to as *mapper* in the following text) is implemented in the SciPy library for Python.[21]

We used almost the same training schedule as the original one – the only change we made was adjusting the value of w_{ML} in the last stage. The architecture we used is as follows: R^8 with $l_{\mathrm{hidden}} = 3$ and $n_{\mathrm{hidden}} = 200$.[22] Dimer distance is used as the RC during the training, i.e.:

$$r(\mathbf{x}) = \sqrt{(x_{1x} - x_{2x})^2 + (x_{1y} - x_{2y})^2}. \tag{79}$$

Although the system dimension is much higher than for the previous problems, the BG is able to learn sampling of the Boltzmann distribution. It samples configurations along the whole interval of the dimer distance despite transition states not being part of the training set, see fig. 18.

Note that the BG tends to frequently sample closed configurations with lower energy. Nevertheless, this does not represent a problem since reweighting of the samples is used (see fig. 4 and recall the discussion of the double-well problem). Therefore, what we need before the reweighting is a sufficient overlap of the distributions instead of an exact match.

[21] https://scipy.org

[22] In this case, $l_{\mathrm{hidden}} = 3$ was used in the BG article, even though they state that 3 was number of hidden layers + output layer.

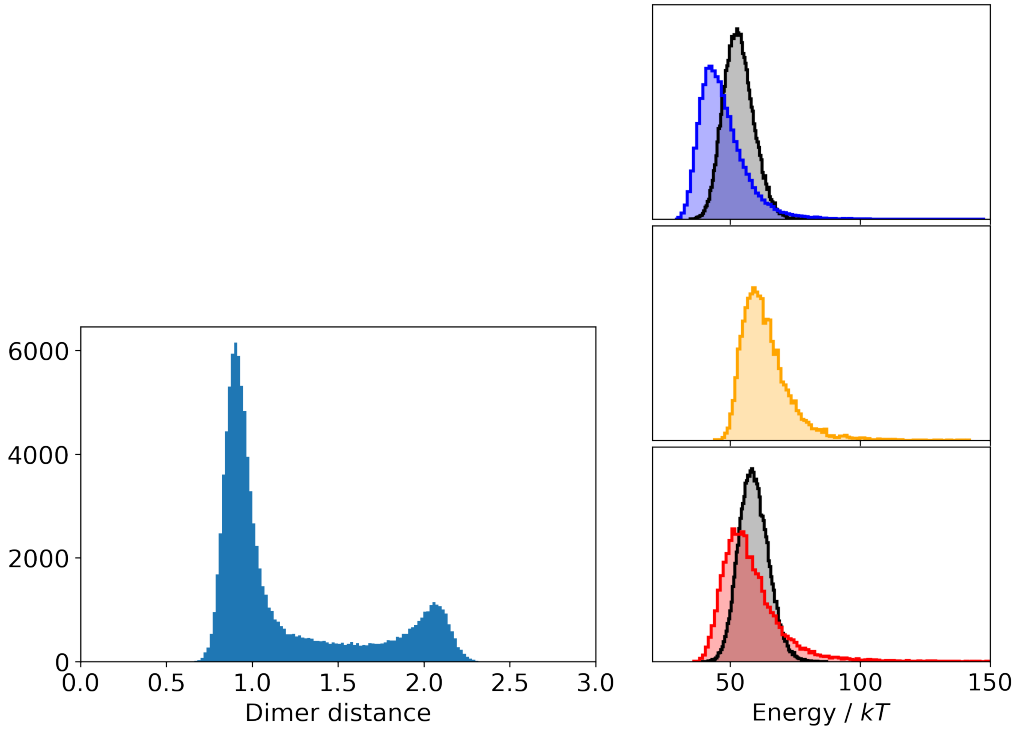

Figure 18: Properties of the configurations sampled by the BG for the bistable-dimer problem. *Left:* histogram of the dimer distance obtained by sampling 100 000 configurations using the BG. *Right:* distribution of the potential energy $U(\mathbf{x})$ for the same configurations with comparison to the training dataset. First row: Closed configurations ($d < 1.3$) from the BG (blue) and the training dataset (gray). Second row: Transition states ($1.3 < d < 1.7$) from the BG (orange). Third row: Open states ($1.7 < d$) from the BG (red) and the training dataset (gray). Each row uses a different y-axis. Note that since reweighting would be done when calculating expectation values, an adequate overlap of the presented histograms is sufficient.

For example, the BG is able to calculate the whole free-energy profile between the closed and open metastable states, see fig. 20.

We also tried the latent interpolation for this system. The best result can be seen in fig. 19. Although the PES is very roughed, the BG is able to find a low-energy transition.

Changed solvent density

After the replication of the results for the BG article,[23] we started producing our own, completely new, results. Firstly, we wanted to examine the effect of the solvent density on the BG performance. With a higher solvent density, there are more high-energy configurations with overlapping solvent particles and thus the PES is even more rough than before.

We started with *decreasing* the solvent density. The easiest way to do this is to keep the number of solvent particles N_S but decrease the box area $4l_{\text{box}}^2$. We set the density to 95% of the original value. As expected, the BG was able to learn faster – the training statistics converged more quickly. However, we obtained no other valuable information.

[23] We were not interested in the protein folding so we have decided to skip this system.

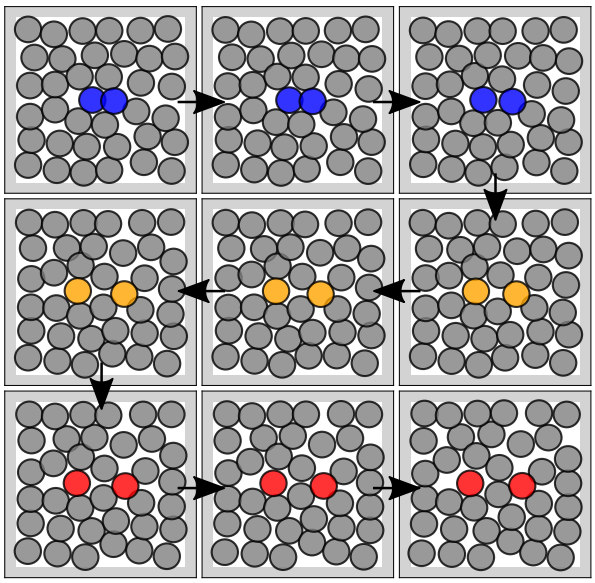

Figure 19: Transition path from the closed to open dimer state obtained by the latent interpolation using the BG. 90 states are selected from MCMC simulations of both the open and closed state and the latent interpolation is performed for each pair, i.e. 8100 in total. The shown result is the one with the lowest maximal potential energy along the transition path.

When the solvent density was *increased* by 5%, we found out that lower σ_{metro} must be used for optimal sampling of the configuration space with the MCMC simulations. Namely, we changed its value from 0.02 to 0.015. Furthermore, we had to prolong the final stage of the training from 2000 to 3000 iterations and change w_{RC} from 10 to 12 for this stage. This way, the length of the training was increased by 15%. After these adjustments, the BG learned sampling of the system's Boltzmann probability distribution – the calculated FEP has approximately the same quality as for the standard density and the BG was able to find low-energy transition paths.

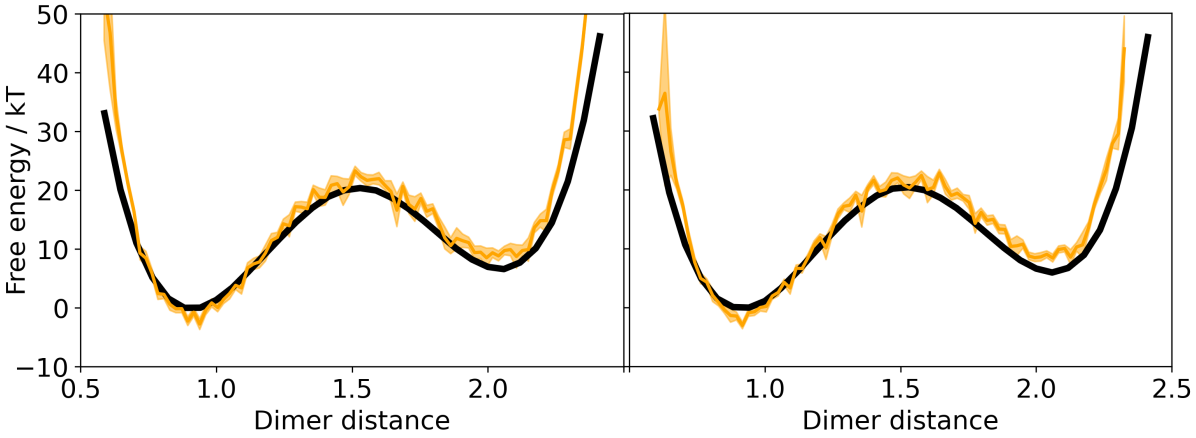

Figure 20: Free-energy profiles obtained with the BG for solvated bistable dimer (orange) compared to the reference umbrella-sampling calculations (black). *Left:* standard solvent density. *Right:* 5% higher solvent density. The error given by bootstrapping (recall algorithm 3) is marked by the semi-transparent area around the orange line.

Nevertheless, when the solvent density was increased by another 5% to a total of 10% growth, the BG was not able to learn the sampling even with slight changes in the training schedule. Therefore, it seems that a higher capacity of the BG would be needed and one would have to fully remake the training schedule.

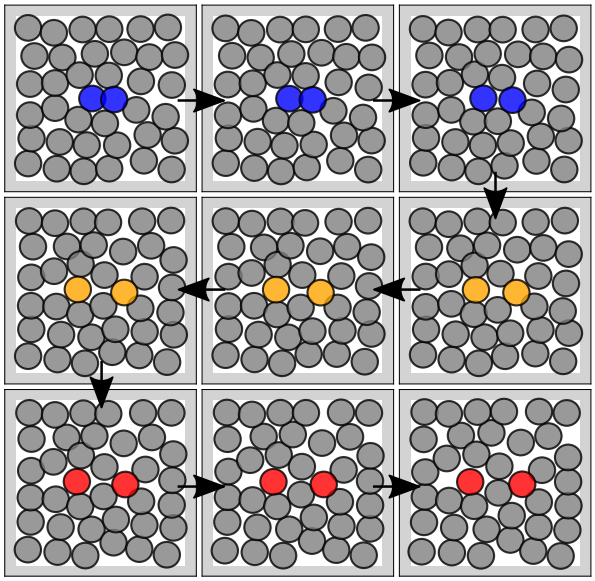

Figure 21: Transition path from closed to open dimer state obtained by the latent interpolation for the system with 5% higher solvent density. This time 100 configurations are selected from both MCMC simulations resulting in a 10^4 trial interpolations. The best result is shown in this figure.

6 Application of the BG to a small LJ cluster

6.1 The system

The next system was not explored in the original article and, therefore, fully represents our own work. It consists of $N = 7$ particles with Lennard-Jones interactions. Due to the attractive part of the LJ potential, the particles naturally form a cluster so, unlike for the solvated dimer, there is no need for a box.

We will use the following notation for the configuration vector $\mathbf{x}$, which is similar to the one used for the bistable dimer:

$$\mathbf{x} = (x_{1x}, x_{1y}, x_{2x}, \ldots, x_{Ny})^\top. \tag{80}$$

The potential energy is defined as follows:

$$U(\mathbf{x}) = 4\varepsilon \sum_i \sum_{j>i} \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] + \frac{1}{2} k_{\text{origin}} \left(x_{\text{CM}}^2 + y_{\text{CM}}^2 \right), \tag{81}$$

where r_{ij} is the distance between particles i and j and

$$x_{\text{CM}} = \frac{1}{N} \sum_i x_{ix} \quad y_{\text{CM}} = \frac{1}{N} \sum_i x_{iy},$$

i.e. $(x_{\text{CM}},\ y_{\text{CM}})^\top$ is the position of the center of mass. The first term in eq. (81) corresponds to the LJ interactions and the second one represents the potential which attracts the cluster's center of mass to the origin of the coordinate system. Parameters of the used potential are given in table 6.

N	ε	σ	k_{origin}	$k_B T$
7	10	1	15	1

Table 6: Parameters of the reduced potential used for the LJ cluster.

The cluster has a few metastable states, determined by its potential, and thus can undergo various transitions between them. We will be interested in such transitions where the initial and final configurations look alike but the positions of the particles are permuted. Therefore, it might be more natural to use the term *rearrangement* instead of *transition* when referring to this process. Nevertheless, we are going to use both of them in the following text. The rearrangements are only possible if we can distinguish between the particles – if numbers or other IDs were not assigned to the particles, the final configuration would be considered to be the same as the initial one, see fig. 22.

In particular, we are going to select one specific particle and define the state of the system based on its position. There are two options:

- either the particle is in the center of the cluster or
- it is on its surface.

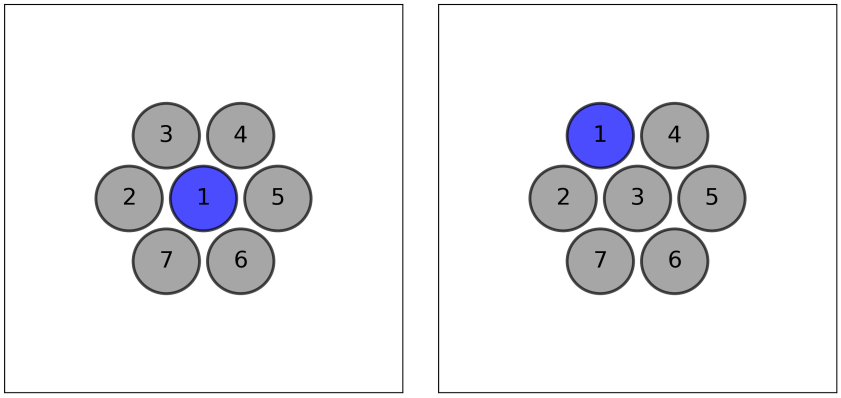

Figure 22: Example configurations of the LJ cluster in its base state. *Left:* center state. *Right:* surface state. Note that any permutation of the particles that keeps the marked particle in the center/on the surface of the cluster is considered to be center/surface configuration, respectively.

We will call these configurations as *center* and *surface* states, respectively. The selected (*marked*) particle will be assigned the number 1 and it will be also distinguished by using a different color.

The transitions we will be interested in will be defined as *displacement of the marked particle from the center to the surface of the cluster while the initial and final configurations represent the base state of the cluster*. Naturally, there are several ways of rearranging the particles which lead to this transition. The system and its transitions has been already discussed in the literature [48, 49] and it is used as a benchmark for transition-path finding algorithms.

We developed a new energy model `LennardJonesCluster`, as was already explained in section 4. The calculation of the potential energy $U(\mathbf{x})$ uses only functions from the NumPy and TensorFlow libraries; so no other software package was used for this purpose. We have implemented also other auxiliary methods for this class in order to facilitate working with the system.

After that, we investigated how the system reacts to the bias umbrella potential using different reaction coordinates. Our first proposal was to use the distance between two particles, the marked particle and a second one that is initially on the surface of the cluster, as the RC. However, it turned out that this RC does not conveniently characterize the transition, which was apparent mainly from the inaccurate FEP calculation obtained from the umbrella sampling.

The second proposal was to use the *coordination number* (CN) of the marked particle as the RC and this choice was much more suitable. The CN of the particle A is defined as, roughly speaking, the number of other particles *around* A. Naturally the CN cannot be defined as the number of particle centers in a circle with the center in A. This would result in the function $\mathrm{CN}(r)$ having derivative equal to zero everywhere but a single point with derivative equal to minus infinity. Therefore, various smooth *switch functions* are used in the definition of the CN. We used the following implementation for the CN of the

particle i:[24]

$$\mathrm{CN}(i) = \sum_{j \neq i} s_{ij} \qquad \text{where} \qquad s_{ij}(r_{ij}) = \frac{1 - \left(\frac{r_{ij} - d_0}{r_0}\right)^n}{1 - \left(\frac{r_{ij} - d_0}{r_0}\right)^m}. \tag{82}$$

The parameters of the switching function s are shown in table 7 and its graph in figure 23. It turns out that when this setup is used, the center state corresponds to $r(\mathbf{x}) \equiv \mathrm{CN}(1) = 5.6$ and the surface state to $\mathrm{CN}(1) = 3.3$. Note that these numbers would be 6 and 3, respectively, if we used the *step* switch function with an discontinuous derivative.

d_0	r_0	n	m
0.3	1.3	6	12

Table 7: Parameters of the switching function s used in the calculation of the coordination number which was used as the RC for this system.

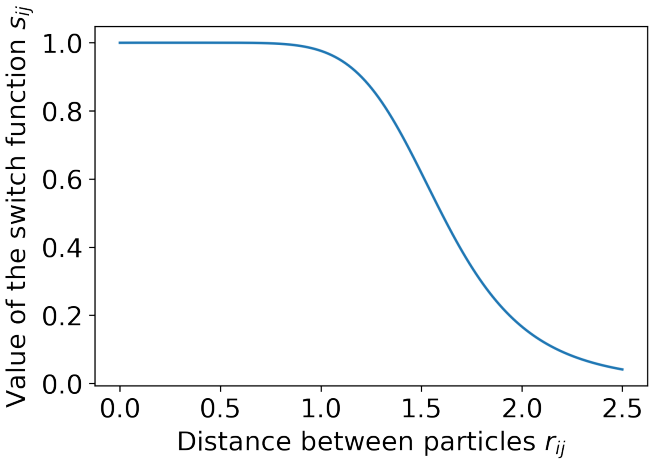

Figure 23: Graph of the switch function s_{ij} used in the computation of the coordination number (82).

We have found out that $\sigma_{\text{metro}} = 0.025$ is the value with the largest mean squared displacement of the configuration $\mathbf{x}$ between the consecutive MCMC steps and, therefore, this value was chosen for all the following MCMC simulations, see fig. 24.

We have conducted an Umbrella sampling MCMC simulation with the following parameters: $r_{\min} = 2.7$, $r_{\max} = 6$ with the interval being divided to 60 windows. The force constant of the bias potential (20) was set to 400. Each window used 10^5 MCMC steps, of which 20 000 were burn-in and thus not saved. The simulation ran both forwards and backward the RC interval resulting in a total of 120 simulation windows. The free-energy profile was calculated using the MBAR [22] and can be seen in fig. 30. Note that the FEP has two minima corresponding to CN ≈ 4 and CN ≈ 4.8. Example cluster configurations corresponding to these minima are shown in fig. 25.

The training dataset was produced, as always in the thesis, with two disconnected MCMC simulations – one for the center and one for the surface state. Each simulation

[24] See e.g. http://www.plumed.org/doc-v2.5/user-doc/html/_c_o_o_r_d_i_n_a_t_i_o_n.html and http://www.plumed.org/doc-v2.5/user-doc/html/switchingfunction.html

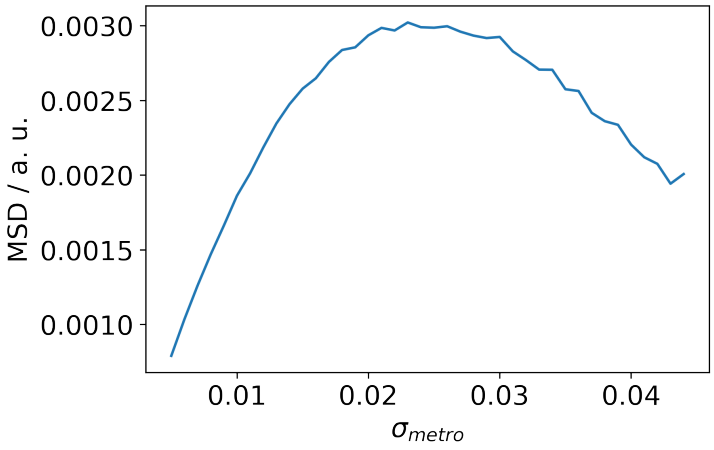

Figure 24: Mean square displacement (MSD) between consecutive MCMC steps for various settings of σ_{metro}. Based on this, we have decided to use $\sigma_{\text{metro}} = 0.025$ for our MCMC simulations.

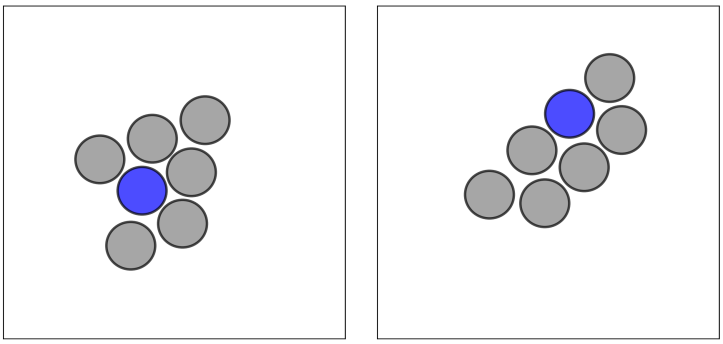

Figure 25: Example configurations with values of the coordination number corresponding to the two free energy local minima. *Left:* configuration with CN(1) = 4.8. *Right:* configuration with CN(1) = 4.

ran for 5×10^5 steps with a saving stride of 20. The resulting dataset consists of 50 000 configurations and is composed of an equal number of the center and surface states.

Similarly to the bistable dimer, the Hungarian mapper was applied to the training dataset. The way of its application plays the *crucial* role in the training. Keep in mind that for the given configuration vector $\mathbf{x}$ there are 6! particle permutations corresponding to the center configurations and $6 \times 6!$ permutations corresponding to the surface ones. As can be easily imagined and also seen in [48, 49], different transition mechanisms result in different final configurations, even if the initial one was the same.

The Hungarian algorithm maps these permutations to the single one which is used as a reference. However, the BG can produce low-energy transition paths *only* between the configurations (particle permutations) *used in the training dataset*. Therefore, if we applied the algorithm to both MCMC simulations and used one reference configuration for each of them, the BG would be able to produce only *single* transition mechanism defined by chosen particle permutations for the center and surface states.

To sum up, there are two opposing effects here:

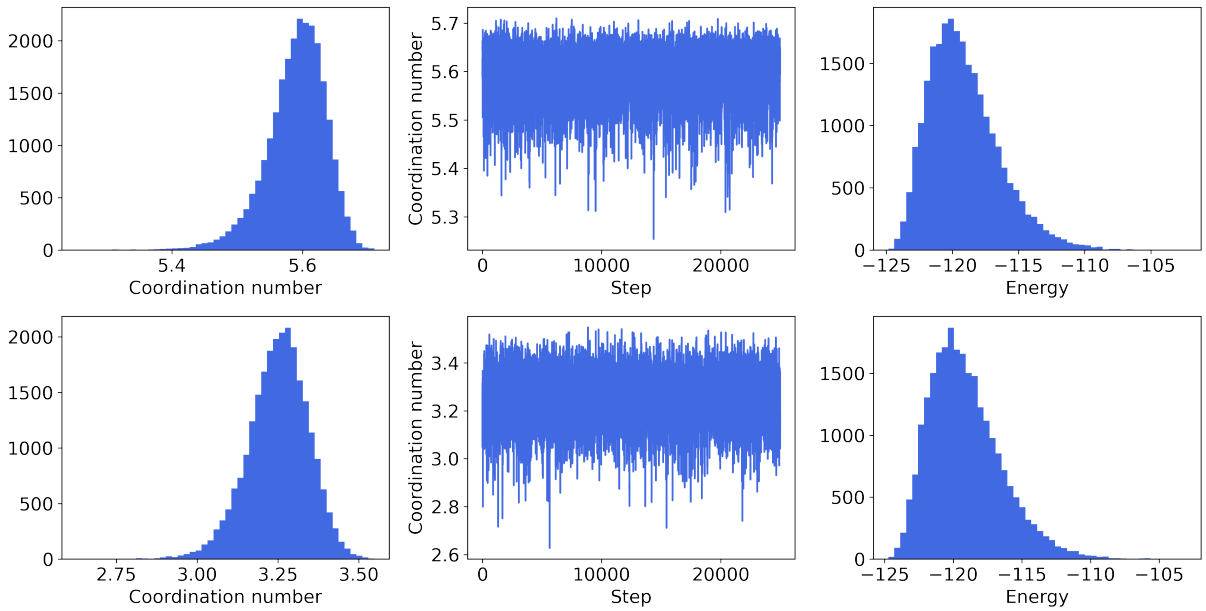

Figure 26: Statistics of the MCMC simulations that produced the training dataset. From left to right: histogram of the coordination number $r(\mathbf{x})$, coordination number as a function of the no. of step during the simulation and, finally, histogram of the potential energy $U(\mathbf{x})$ for the simulation. *Top:* simulation running in the center state. *Bottom:* Simulation running in the surface state.

- The more permutations we accept, the more transition mechanisms will be obtained from the BG latent interpolation, but
- the more permutations we accept, the higher volume of the configuration space will be occupied in the training dataset. This results in a slower BG training and the need for a more complex BG to be used – the training of the BG may become even unfeasible.

Let us illustrate these ideas on the bistable dimer. Here, the part of the system switching between the two states (biparticle dimer) was strictly separated from the rest of the system (solvent). Therefore, one could apply the Hungarian mapper with the same reference configuration to MCMC simulations of both dimer states. This led to the BG learning only those transitions for which the initial and final "permutations" of the solvent particles are the same.

Our aim for the LJ cluster was to reproduce all four transition mechanisms described in [48, 49].[25] We found out that when the same initial configuration is used, these mechanisms have only 3 different final permutations of the particles.[26] Therefore, we chose one reference configuration for the center states and three of them for the surface ones. These can be seen in fig. 27. The whole center MCMC simulation was mapped using the same reference configuration, whereas the surface simulation was split into three parts of approximately the same size and a different reference configuration was used for each of

[25]The mechanisms shown in fig. 3(a) of [49] and last row of fig. 4 in[48] are in fact identical.

[26]Namely, the first two mechanisms described in [48, fig. 4] have the same final state (after ignoring the rotation of the whole cluster).

these parts. In other words, the center-phase half of the training dataset was comprised of only one particle permutation while there were three particle permutations present in the surface-phase half of the training dataset.

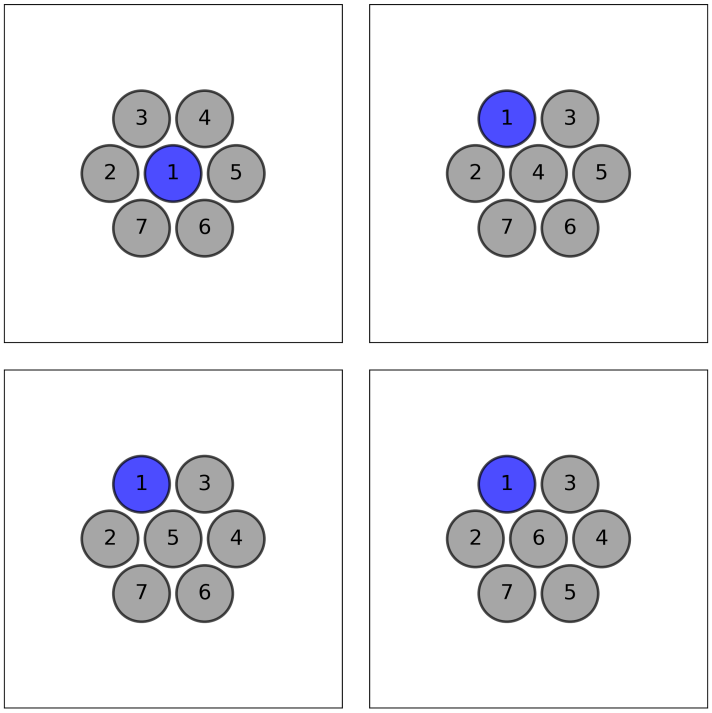

Figure 27: Used reference configurations for the Hungarian mapper. *Top left:* Ref. configuration for the center states. *Others:* Ref. configurations for the surface states, which are the final states of transition mechanisms described in the literature.

6.2 Challenge of setting up the training schedule

The small Lennard-Jones cluster was the first system for which we had to form the training schedule from the ground up. In this section, we would like to share our experience and give some ideas which might help the reader to train the BG for another new system.

Using the current level of understanding the BGs, its training process can be divided into three phases:[27]

1. Getting the latent image of the training dataset close to the origin of the latent space, which is sampled frequently due to the normal distribution in $\mathbf{z}$.

2. Decreasing potential energy of the produced samples $\mathbf{x} = F_{zx}(\mathbf{z})$.

3. Enhancing the sampling along the reaction coordinate $r(\mathbf{x})$.

It can be seen that all the training schedules presented by the original article follow these individual phases and our experience confirms the rightness of this approach.

The first phase is usually very short and can be executed in merely one training stage that uses only the ML loss and has $10^1 - 10^2$ iterations. Training by example

[27] Note the difference from training *stages* which have been introduced sooner.

generally helps the BG to focus on the important parts of the configuration space, which are represented by the training dataset. We suggest using just a small number of iterations in the first stage and stopping it at the point where the loss is still quickly (linearly) decreasing.

At the beginning of the second phase, the training by energy (KL loss) is introduced. Note that the three used losses have very different intervals of their possible (absolute) values. The main contribution to the KL loss comes from the reduced potential $u(\mathbf{x})$. Therefore, typical values of the J_{KL} are determined by the typical energies of the configurations. For systems with divergent repulsive interactions (like solvated dimer or LJ cluster), the KL loss can have immense values and thus the gradient as well. Values of the ML loss depend on the dimensionality of the problem but for a non-trivial system are by order(s) of magnitude smaller than the KL loss since $|u_Z| \ll |u_X|$. For the LJ cluster we had $J_{\mathrm{ML}} \sim \mathcal{O}(10)$. The RC loss is the smallest one with the upper bound given by the entropy of the uniform distribution. In the current implementation this means that $J_{\mathrm{RC}} \in [-\ln(11); 0]$.

Therefore, contributions of the individual losses to the total gradient differ significantly. One should keep this in mind when making a training schedule. Nevertheless, this effect can be controlled by the proper setting of the loss weights.[28]

Because of this, there is a danger when the training by energy starts – the large gradient of the KL loss resulting from the BG sampling mainly random configurations with numerous particle overlaps. When the BG follows this gradient and learns too fast (this can be identified e.g. by an intense drop in the total loss), it results in the BG starting to sample only a close vicinity of the lowest energy configuration. Usually, there is no feasible way of correction when such collapse happens; the schedule should be adjusted and the training restarted. For completeness, this danger is present also later in the training but the risk is much lower than in the moment when J_{KL} is introduced.

There are several ways of preventing this from coming. If the BG learns too fast, one can:

- Lower the $w_{\mathrm{KL}}/w_{\mathrm{ML}}$ ratio. This puts more emphasis on sampling configurations from the training dataset and thus forestalls the BG from focusing too much on a small volume of the configuration space.

- Decrease the learning rate – the training is slowed down and the BG learns more slowly but more wisely.

- Decrease the parameter E_{high} of the energy regularization. This way the higher energies will end up in the area of the logarithmic cut and corresponding gradients will be reduced.

On the other hand, in some cases, the BG may learn too slowly in the sense that it fails to increase the ratio R^x_{low} of low-energy configurations among the samples. This is an indication that the sampling problem is too complex for the BG. In this case, one should either increase the complexity of the BG (i.e. increase number of its layers, l_{hidden}

[28] Also keep in mind that by scaling values of the loss its gradient is being scaled as well.

or n_{hidden}) or use the Hungarian mapper more strictly to decrease the volume of the configuration space occupied by the training dataset.

In the second training phase, we aim to gradually increase the ratio of produced low-energy samples and the phase usually comprises multiple stages. One should also include the RC loss soon, otherwise, the BG may ignore other regions of the configuration space than the ones occupied by the training dataset. In some cases, if the RC loss is not applied very soon, this phenomenon cannot be averted later. This also holds for the system of the LJ cluster.

In the last phase, we aim for an enhancement of the sampling along the RC. We suggest to slightly increase the weight of the RC loss and use a high number of the training iterations so that the BG can take small steps. Rising w_{RC} too much usually results in a drop in R^x_{low}. In other words, the KL and RC loss often oppose each other: the former tries to decrease samples' energies while the latter tries to strengthen the sampling along the RC (and thus usually raises the average energy of the samples). Going too far with any of these losses will result either in sampling only the metastable states or sampling configurations with unnaturally high energies along the RC. Therefore, the ratio of these two has to be carefully adjusted so that the BG can sample configurations with as low energy as possible along the whole RC interval.

Last but not least, let us give also some general advice:

- In case that high oscillations of the loss are arising, one should increase the batch size or decrease the learning rate.
- Note that the relationship between a decrease in the loss and improvement of the BG statistics is not straightforward – sometimes small changes in the loss improve the statistics significantly and, on the other hand, a substantial drop in the loss may have a merely slight impact on the BG performance.
- We found out that training the BG for a long time with just a slight decrease in the loss deteriorates the training performance in the following stages. Therefore, we suggest using this technique only in the last stage and make sure that the next stage is started if the loss decreases too slowly for the current stage.
- Due to the very different sizes of the used losses we strongly recommend tracking all losses individually, not merely the total loss.

Using these points, we developed a training schedule for the LJ cluster. We tried to correctly set one training stage at the time and we continued to the next one only if we were satisfied with the previous one. After setting the parameters for the last stage, we tried to understand the training as a whole and adjusted the previous stages further. The resulting schedule can be seen in table 8 and the example training statistics in figure 28.

We used the BG with R^8 architecture using $l_{\text{hidden}} = 5$ and $n_{\text{hidden}} = 200$. As a result, this is the most complex BG we used – it has 5×10^6 trainable parameters. Note that although the *physical* dimension of the solvated dimer was higher, it seems that the LJ cluster represents a more difficult challenge for the BG due to multiple permitted particle permutations – the BG has to learn sampling of multiple *separated* regions of the configuration space.

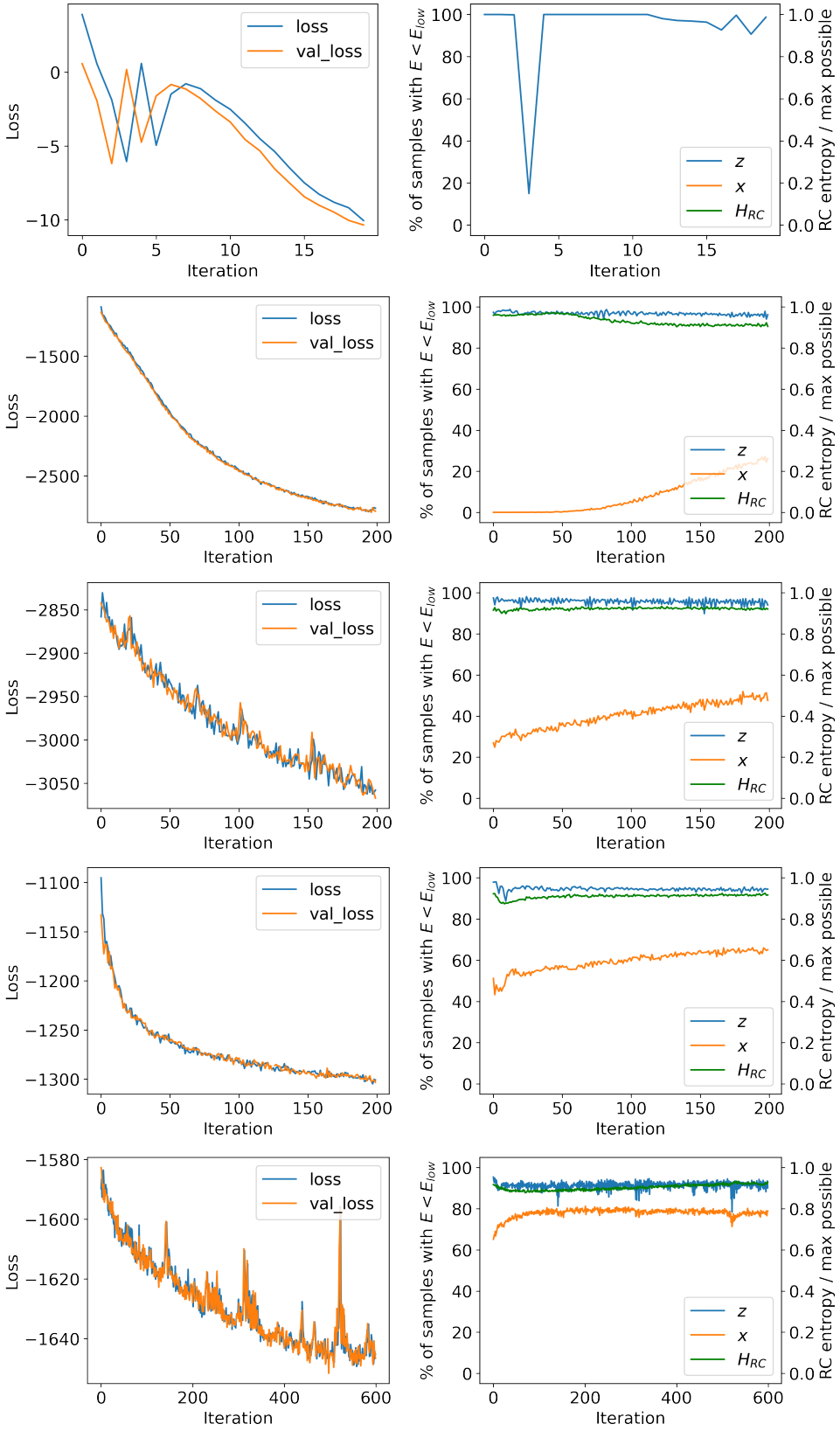

Figure 28: Training statistics for the LJ-cluster system. Five rows correspond to five training stages, see table 8. *Left column:* value of the total training (blue) and validation (orange) loss. *Right column:* blue – ratio of mapped samples $\mathbf{z} = F_{xz}(\mathbf{x})$ whose energy in the latent space is lower than $\langle u_Z \rangle + 2\sigma[u_Z]$; orange – ratio of the generated samples $\mathbf{x} = F_{zx}(\mathbf{z})$ that have energy $u(\mathbf{x}) < -110$; green – entropy H_{RC} of the RC probability distribution divided by the maximal possible value (corresponding to the uniform distribution).

i	Iterations	Batch size	w_{ML}	w_{KL}	w_{RC}	ε	E_{high}
1	20	2000	1	0	0	10^{-3}	10^4
2	200	4000	100	0.01	50	10^{-4}	10^4
3	200	4000	100	0.1	100	10^{-4}	5×10^3
4	200	4000	30	3	100	10^{-4}	10^3
5	600	4000	20	10	130	10^{-4}	10^3

Table 8: Training schedule used for the LJ cluster. ε stands for the learning rate and E_{high} is one of the parameters of the energy regularization (71), E_{max} was set to 10^{10}. Although we used a very complex BG for this system, we were able to train it much faster than the BG for the solvated bistable dimer.

6.3 Results

After the training, we evaluated the qualities of the BG. Properties of the samples can be seen in fig. 29. The BG could sample configurations from the whole interval of the RC. Let us remind that after the BG learns the probability distribution, it can generate independent samples without any major computational cost.

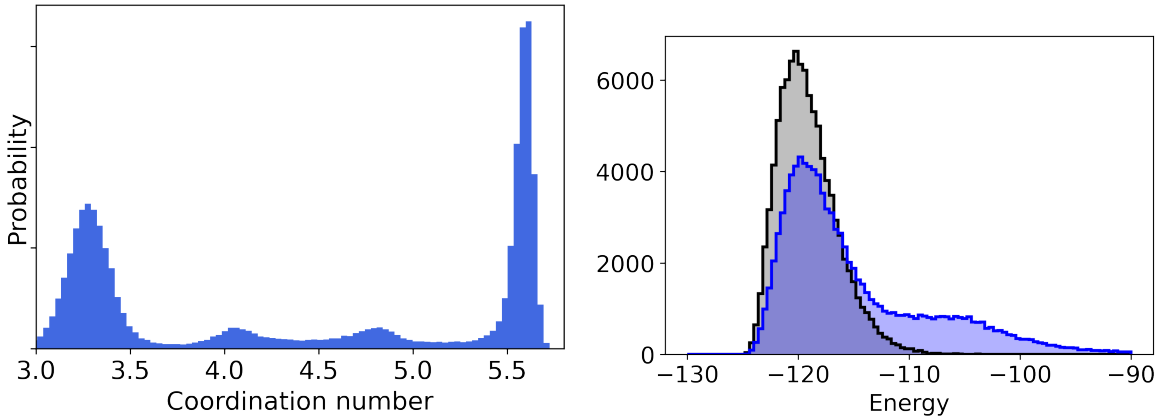

Figure 29: Properties of 100 000 samples produced with the BG. *Left:* Distribution of the coordination number calculated for the marked particle. Center states have CN $\approx$ 5.6 and surface ones $\approx$ 3.3. Note the two local maxima corresponding to two local minima of the free energy. *Right:* Histogram of the samples' potential energy (blue) compared with the training dataset (gray; with weight 2 since it contained 50 000 samples).

Nevertheless, we did not succeed in the accurate calculation of the free-energy profile, see fig. 30. Although the FEP is qualitatively correct, it is shifted upwards in the central part and predicts a lower value in the rightmost minimum. This defect turned out to be systematic and has been recurring for multiple training schedules. In our opinion, this is related to the application of the Hungarian mapper – allowed particle permutations define transition paths and the BG tends to learn a sampling of the configurations mainly along these paths. The most probable reason why this did not have an impact on the solvated dimer is that the dimer undergoing transition was more strictly separated from the solvent and the Hungarian algorithm was not used for the dimer.

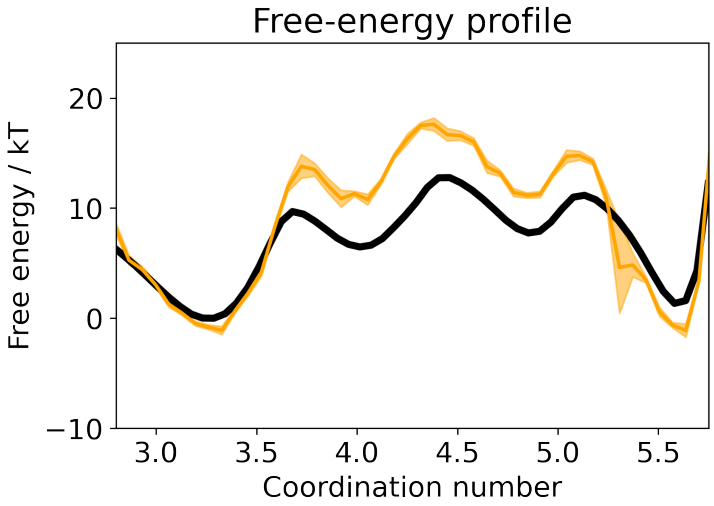

Figure 30: Free-energy profile for the LJ cluster calculated with bootstrapped calculation using the samples from the BG (orange) compared with the reference Umbrella sampling calculation (black). The predicted error is marked by the semi-transparent orange area.

The most interesting results are, however, the latent interpolations using which we obtained three of the four transition mechanisms described in the literature. The individual observed transition mechanisms are shown in figures 32 – 34. We were not able to reproduce one of the mechanisms which shared the initial and final permutation of the particles with the transition path shown in fig. 34 – the latent interpolation for these boundary conditions resulted always in the same transition mechanism. All of these paths have maximum energy lower than −96 and thus with three mechanisms we exceeded the results presented in [49]. Energy profiles of the transition paths shown in [48] are merely schematic and, therefore, not suitable for precise comparison.

We were also interested in the quality of *all* latent interpolations instead of just the best ones. As a result, in fig. 31 we present a histogram of the maximal values of potential energy $u(\mathbf{x})$ along the paths produced with 25 000 interpolations.

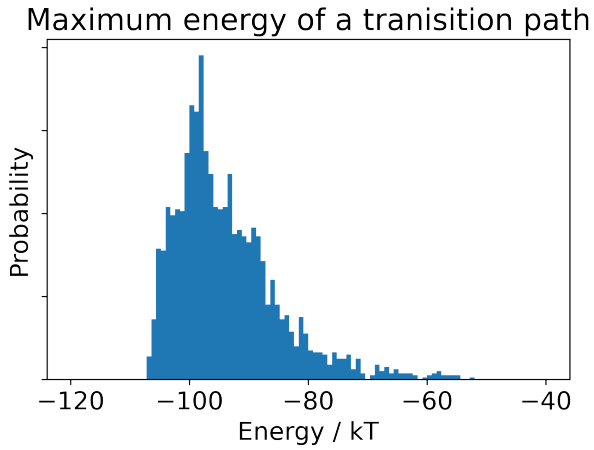

Figure 31: Distribution of maximal potential energy along the transition path for 25 000 trial latent interpolations.

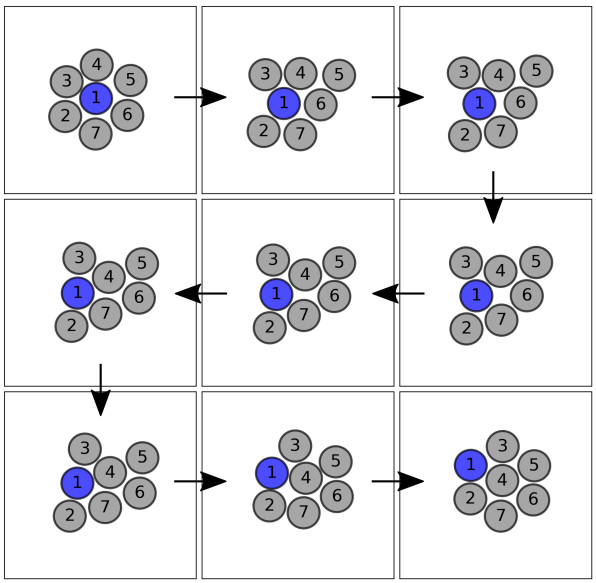

Figure 32: The first transition mechanism of the LJ cluster, already presented in [49, fig. 3(a)]. Particles 1, 3 and 4 rotate around their common center which causes particle 1 to get on the surface of the cluster. Maximal energy along the shown path is −107.

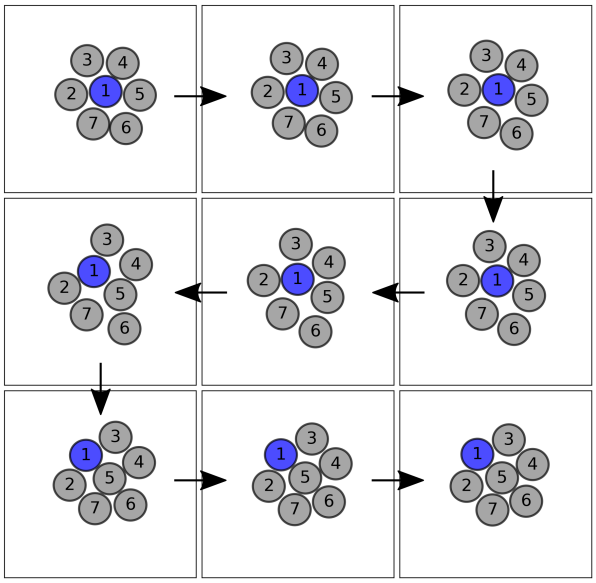

Figure 33: The second transition mechanism of the LJ cluster, already presented in [49, fig. 3(b)]. Particles 3 and 4 slide down on top of particles 1 and 5. The rest of the cluster simply reacts to this motion. This transition is the least optimal and has maximal energy of −96.5

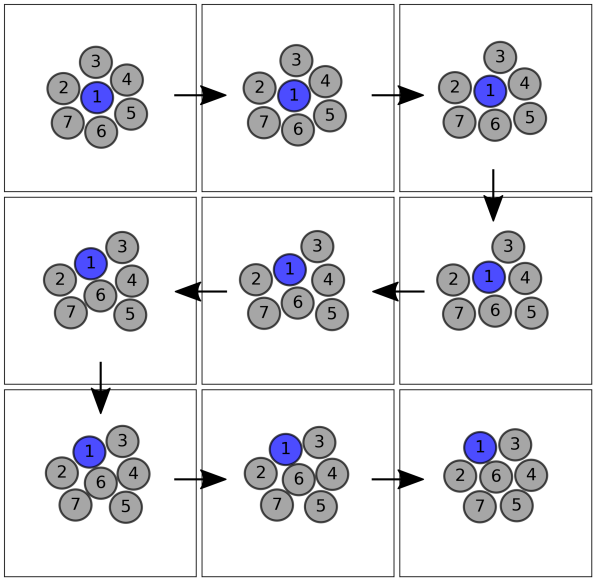

Figure 34: The third transition mechanism of the LJ cluster, already presented in [48, first row of fig. 4]. First, particles 3, 4 and 5 slide down and are followed by particles 2 and 7 on the other side of the cluster. Maximal energy along the path is −102.4.

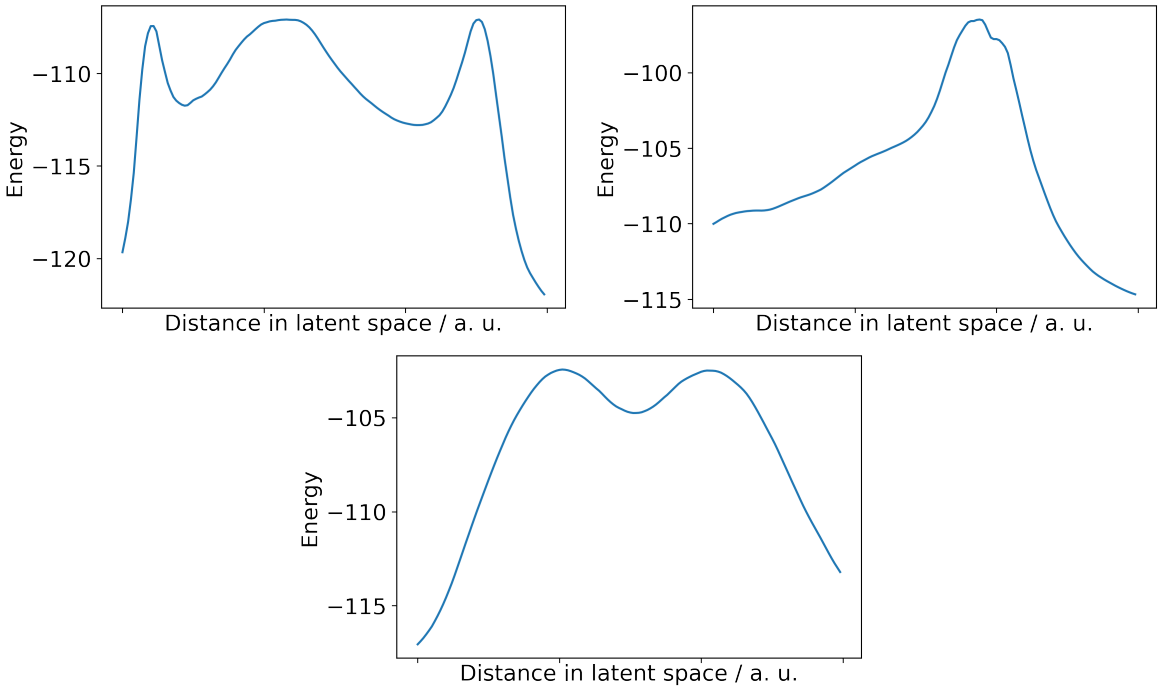

Figure 35: Potential-energy profiles along the shown transition paths. Top left, top right and bottom graph corresponds to transition 1, 2 and 3, respectively.

Conclusions

In the present thesis, we explored the possibilities of the novel enhanced sampling method called the Boltzmann generator. After discussing the theoretical base of the technique, we replicated the results of the original article [2] for the 2D toy models and the system of the solvated bistable dimer. We continued in our work beyond the scope of the original BG article and investigated the effect of the solvent density on the last-mentioned model. After that, we applied the BG to the new system – a small Lennard-Jones cluster in two dimensions. We studied mainly the rearrangements of the cluster and their various mechanisms.

The BG uses a totally different approach than other contemporary methods and is based on deep generative neural-network models. We have refactored the originally used code and adapted it to TensorFlow 2.0. Furthermore, we have been able to acquire intuition for the training of the BG which resulted in successfully using it for the new physical system.

Nevertheless, we have not discussed some topics in more detail. Namely, we have not tried to calculate the free-energy difference between two states using two separate BGs [2, fig. 6]. Moreover, we have not used the BG to gradually explore the configuration space [2, fig. 4].

We also skipped the discussion of BG efficiency when compared to the other methods. Our main objection regarding this topic is that in the original article, the BG is usually compared to the RC-free approaches like plain MCMC or MD. However, in order to get optimal results, the RC was used for all the systems presented in the article. When no RC is used, the BG can

- compute the free-energy difference between the metastable states of which the training dataset is comprised and
- sample independent configurations from all metastable states without any significant computational cost.

However, in order to obtain the full free-energy profile or low-energy transition paths between the states, one has to include the RC loss in the training.

It seems that the BGs are not the most optimal choice for individual kinds of tasks such as the calculation of FEP or the construction of transition paths but are very unique and useful due to their complexity since they can solve all these tasks at once. This way, their application to the system can give an overall insight into the problem. Nevertheless, the BGs are just in their beginnings so we suggest not to rely on precise comparisons with other methods until the BGs further evolve.

Due to the method being at its very outset, there are numerous areas where progress can be made. We would like to emphasize the following ones:

- Using more advanced layers than the Real NVP: the research progresses in the area of the normalizing flows so one could try to use the newer designs in the BGs. However, the ability to produce transition paths strongly depends on the architecture of the used layers. We would particularly like to point to the GLOW layers [44], which have been proven to have this feature.

- Linking up the BGs with the software specialized for the calculation of the potential energy $U(\mathbf{x})$: we have used only the Python code and NumPy/TensorFlow libraries for calculation of the energy. However, this is not feasible for larger systems. The authors of the BGs have used OpenMM[29] package for the protein problem but this choice does not seem to be convenient for physical systems since OpenMM is focused on computational biology. We suggest linking the BGs with LAMMPS [43], particularly with its Python interface called PyLammps.[30]

- Finding the way of incorporating symmetries of the system into the architecture of BG's transformation F: this is of particular importance for the scaling ability of the method. Currently, it is also the most important research area of the team that invented the BGs [50, 51]. One should also find a way to reduce the negative impact the Hungarian mapper may have (as, in our opinion, it had for the LJ cluster).

- Using the BG to sample the isothermal-isobaric (NPT) ensemble: this would mean to use the reduced potential $u_{\mathrm{NPT}}(\mathbf{x}) = u(\mathbf{x}) + PV(\mathbf{x})/k_BT$ and include information about the volume in the configuration vector $\mathbf{x}$. For example, for the triclinic structures, the configuration vector $\mathbf{x}$ would contain six independent components of vectors $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ defining the unit cell.[31]

We believe that when these challenges are solved, the BGs will become one of the core methods of enhanced sampling.

[29] http://openmm.org
[30] https://lammps.sandia.gov/doc/Howto_pylammps.html
[31] https://lammps.sandia.gov/doc/Howto_triclinic.html

References

[1] Paul A. M. Dirac. “Quantum mechanics of many-electron systems”. In: *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character* 123.792 (1929), pp. 714–733.

[2] F. Noé et al. “Boltzmann generators: Sampling equilibrium states of many-body systems with deep learning”. In: *Science* 365.6457 (2019). ISSN: 0036-8075. DOI: 10.1126/science.aaw1147.

[3] F. Noé et al. *Supplementary Materials for Boltzmann generators: Sampling equilibrium states of many-body systems with deep learning*. 2019. URL: http://science.sciencemag.org/content/suppl/2019/09/04/365.6457.eaaw1147.DC1.

[4] F. Noé et al. *Boltzmann generators: Sampling equilibrium states of many-body systems with deep learning* (Version 1.0). 2019. DOI: 10.5281/zenodo.3242635.

[5] R. G. Parr and W. Yang. *Density-functional theory of atoms and molecules*. New York, Oxford: Oxford University Press, 1989. ISBN: 0195042794.

[6] Mark E. Tuckerman. *Statistical Mechanics: Theory and Molecular Simulation*. Oxford university press, 2010.

[7] D. Frenkel and B. Smit. *Understanding Molecular Simulation*. 2nd. USA: Academic Press, Inc., 2001. ISBN: 0122673514.

[8] N. Metropolis et al. “Equation of State Calculations by Fast Computing Machines”. In: *The Journal of Chemical Physics* 21.6 (1953), pp. 1087–1092. DOI: 10.1063/1.1699114.

[9] W. K. Hastings. “Monte Carlo Sampling Methods Using Markov Chains and Their Applications”. In: *Biometrika* 57.1 (1970), pp. 97–109. ISSN: 00063444. URL: http://www.jstor.org/stable/2334940.

[10] Loup Verlet. “Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules”. In: *Phys. Rev.* 159 (1 July 1967), pp. 98–103. DOI: 10.1103/PhysRev.159.98. URL: https://link.aps.org/doi/10.1103/PhysRev.159.98.

[11] W. C. Swope et al. “A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters”. In: *The Journal of chemical physics* 76.1 (1982), pp. 637–649.

[12] P. H. Hünenberger. “Thermostat algorithms for molecular dynamics simulations”. In: *Advanced computer simulation* (2005), pp. 105–149.

[13] R. Swendsen and J. S. Wang. “Replica Monte Carlo Simulation of Spin-Glasses”. In: *Physical review letters* 57 (Dec. 1986), pp. 2607–2609. DOI: 10.1103/PhysRevLett.57.2607.

[14] E. Marinari and G. Parisi. “Simulated Tempering: A New Monte Carlo Scheme”. In: *Europhysics Letters (EPL)* 19.6 (July 1992), pp. 451–458. DOI: 10.1209/0295-5075/19/6/002. URL: https://doi.org/10.1209%2F0295-5075%2F19%2F6%2F002.

[15] U. H. E. Hansmann. “Parallel tempering algorithm for conformational studies of biological molecules”. In: *Chemical Physics Letters* 281.1 (1997), pp. 140–150. ISSN: 0009-2614. DOI: https://doi.org/10.1016/S0009-2614(97)01198-6. URL: http://www.sciencedirect.com/science/article/pii/S0009261497011986.

[16] Y. Sugita and Y. Okamoto. “Replica-exchange molecular dynamics method for protein folding”. In: *Chemical Physics Letters* 314.1 (1999), pp. 141–151. ISSN: 0009-2614. DOI: https://doi.org/10.1016/S0009-2614(99)01123-9. URL: http://www.sciencedirect.com/science/article/pii/S0009261499011239.

[17] G. M. Torrie and J. P. Valleau. “Nonphysical sampling distributions in Monte Carlo free-energy estimation: Umbrella sampling”. In: *Journal of Computational Physics* 23.2 (1977), pp. 187–199. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(77)90121-8. URL: http://www.sciencedirect.com/science/article/pii/0021999177901218.

[18] J. Kästner. “Umbrella sampling”. In: *WIREs Computational Molecular Science* 1.6 (2011), pp. 932–942. DOI: 10.1002/wcms.66. eprint: https://onlinelibrary.wiley.com/doi/abs/10.1002/wcms.66.

[19] S. Kumar et al. “The weighted histogram analysis method for free-energy calculations on biomolecules. I. The method”. In: *Journal of computational chemistry* 13.8 (1992), pp. 1011–1021.

[20] M. Souaille and B. Roux. “Extension to the weighted histogram analysis method: combining umbrella sampling with free energy calculations”. In: *Computer physics communications* 135.1 (2001), pp. 40–57.

[21] C. H. Bennett. “Efficient estimation of free energy differences from Monte Carlo data”. In: *Journal of Computational Physics* 22.2 (1976), pp. 245–268.

[22] M. R. Shirts and J. D. Chodera. “Statistically optimal analysis of samples from multiple equilibrium states”. In: *The Journal of chemical physics* 129.12 (2008), p. 124105.

[23] A. Laio and M. Parrinello. “Escaping free-energy minima”. In: *Proceedings of the National Academy of Sciences* 99.20 (2002), pp. 12562–12566. ISSN: 0027-8424. DOI: 10.1073/pnas.202427399. eprint: https://www.pnas.org/content/99/20/12562.full.pdf.

[24] A. Barducci, M. Bonomi, and M. Parrinello. “Metadynamics”. In: *WIREs Computational Molecular Science* 1.5 (2011), pp. 826–843. DOI: 10.1002/wcms.31. eprint: https://onlinelibrary.wiley.com/doi/abs/10.1002/wcms.31.

[25] A. Laio and F. L. Gervasio. “Metadynamics: a method to simulate rare events and reconstruct the free energy in biophysics, chemistry and material science”. In: *Reports on Progress in Physics* 71.12 (Nov. 2008). DOI: 10.1088/0034-4885/71/12/126601. URL: https://doi.org/10.1088%2F0034-4885%2F71%2F12%2F126601.

[26] G. Bussi and A. Laio. “Using metadynamics to explore complex free-energy landscapes”. In: *Nature Reviews Physics* 2.4 (2020), pp. 200–212. DOI: https://doi.org/10.1038/s42254-020-0153-0.

[27] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[28] Tom M. Mitchell. *Machine Learning*. New York: McGraw-Hill, 1997. ISBN: 978-0-07-042807-2.

[29] Ian J. Goodfellow et al. *Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks*. 2014. arXiv: 1312.6082 [cs.CV].

[30] Y. LeCun and C. Cortes. *MNIST handwritten digit database*. 2010. URL: http://yann.lecun.com/exdb/mnist/.

[31] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444.

[32] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning Representations by Back-propagating Errors". In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0. URL: http://www.nature.com/articles/323533a0.

[33] J. Duchi, E. Hazan, and Y. Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *J. Mach. Learn. Res.* 12.null (July 2011), pp. 2121–2159. ISSN: 1532-4435.

[34] D. P. Kingma and J. L. Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[35] A. Oussidi and A. Elhassouny. *Deep generative models: Survey*. Apr. 2018. DOI: 10.1109/ISACV.2018.8354080.

[36] S. Bond-Taylor et al. *Deep Generative Modelling: A Comparative Review of VAEs, GANs, Normalizing Flows, Energy-Based and Autoregressive Models*. 2021. arXiv: 2103.04922 [cs.LG].

[37] Z. Zhu et al. "Using novel variable transformations to enhance conformational sampling in molecular dynamics". In: *Physical review letters* 88.10 (2002), p. 100201.

[38] P. Minary, M. E. Tuckerman, and G. J. Martyna. "Dynamical spatial warping: A novel method for the conformational sampling of biophysical structure". In: *SIAM Journal on Scientific Computing* 30.4 (2008), pp. 2055–2083.

[39] G. Papamakarios et al. "Normalizing flows for probabilistic modeling and inference". In: *arXiv preprint arXiv:1912.02762* (2019).

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

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

[42] M. Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[43] S. Plimpton. "Fast parallel algorithms for short-range molecular dynamics". In: *Journal of computational physics* 117.1 (1995), pp. 1–19. URL: http://lammps.sandia.gov.

[44] D. P. Kingma and P. Dhariwal. *Glow: Generative flow with invertible 1x1 convolutions*. 2018. arXiv: 1807.03039.

[45] K. Müller and L. D. Brown. "Location of saddle points and minimum energy paths by a constrained simplex optimization procedure". In: *Theoretica chimica acta* 53.1 (1979), pp. 75–93.

[46] K. Müller. "Reaction paths on multidimensional energy hypersurfaces". In: *Angewandte Chemie International Edition in English* 19.1 (1980), pp. 1–13.

[47] H. W. Kuhn. "The Hungarian method for the assignment problem". In: *Naval research logistics quarterly* 2.1-2 (1955), pp. 83–97.

[48] C. Dellago, P. G. Bolhuis, and D. Chandler. "Efficient transition path sampling: Application to Lennard-Jones cluster rearrangements". In: *The Journal of chemical physics* 108.22 (1998), pp. 9236–9245. DOI: 10.1063/1.476378.

[49] D. Passerone and M. Parrinello. "Action-derived molecular dynamics in the study of rare events". In: *Physical Review Letters* 87.10 (2001), p. 108302. DOI: 10.1103/PhysRevLett.87.108302.

[50] J. Köhler, L. Klein, and F. Noé. "Equivariant Flows: Exact Likelihood Generative Learning for Symmetric Densities". In: 2020. arXiv: 2006.02425 [stat.ML].

[51] W. Hao, K. Jonas, and F. Noé. *Stochastic Normalizing Flows*. 2020. arXiv: 2002.06707 [stat.ML].