Department of Materials
Imperial College

MEng Thesis

Coarse-Graining of Molecular Dynamics Using Neural Ordinary Differential Equations

Jakub Lála

Supervisors: Dr. Stefano Angiolleti-Uberti and Dr. Rafael Gómez-Bombarelli

Date of submission: 6 June, 2022

Abstract

Simulating composite bodies made of many point particles is computationally demanding due to the vast amount of degrees of freedom. By using a data-driven approach of neural ordinary differential equations (neural ODEs), we show an alternative to learning coarse-grained (CG) machine learning (ML) potentials that reduces the number of simulated degrees of freedom. By construction, we reduce the representation of composite bodies only to their orientation and position of their centres of mass. In the thesis, we gradually build on toy problems: first, we employ neural ODEs in Hamiltonian Monte Carlo sampling of thermodynamic averages; secondly, we confirm that neural ODEs can be used to learn pair-wise potentials without losing the ability to be scalable to multiple pair interactions; and lastly, we develop the backbone for an automated coarse-grainer for any composite body, regardless of its shape or surface complexity. Further work aims to develop this method forward, building a complete pipeline that generates training data, learns the CG potential and runs the reduced representation dynamics.

Contents

Collaboration and supervision

This work began at Massachusetts Institute of Technology (MIT) under the supervision of Dr. Rafael Gómez-Bombarelli. With the help of the PhD candidate James Damewood, the first task of the thesis was carried out. Afterwards, the work has been taken over to Imperial College London under the supervision of Dr. Stefano Angiolleti-Uberti, where the application of coarse-graining using neural ODEs was conceived. His PhD candidate Shanil Panara helped with generating molecular dynamics trajectories using `LAMMPS`, while James Damewood from MIT was consulted throughout about deep learning and neural ODEs.

Acknowledgements

I cannot begin to express my thanks to Dr. Stefano Angiolleti-Uberti, who has sparked my interested in computational physics two years ago during a summer internship, helped me secure an unforgettable internship at EPFL the following summer, and then proved to be a caring, thoughtful and friendly supervisor throughout this year's work. I must also thank Dr. Rafael Gómez-Bombarelli and James Damewood from MIT, who showed me the ambition of pushing science forward. Last but not least, I have to thank my family who has supported me throughout my studies and has let me explore the world while doing so, never once doubting that I should return back home.

1 Introduction

1.1 Historical Context

The methods of artificial intelligence (AI) have been in development for almost a century now. Ever since the invention of the perceptron in 1950s [1], neural networks (NNs) have been incrementally applied to more and more tasks. After the breakthrough of AlexNet in 2012 [2], large deep neural networks have surged in popularity in many disciplines, primarily focusing on the advancement of computer vision and natural language processing. Nevertheless, there is an ongoing adoption of these techniques in the scientific domain as well. For instance, the recent success of DeepMind's AlphaFold protein folding solution [3], or their contribution towards approximating functionals in density functional theory (DFT) [4], show that these novel, general approach methods can aid in accelerating scientific discovery (depicted in Figure 1).

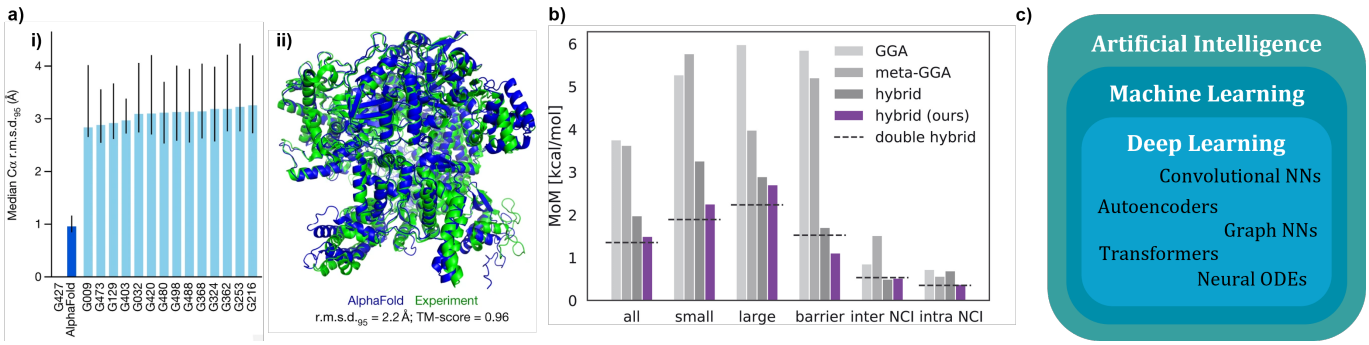


Figure 1: a) i) The performance of DeepMind's AlphaFold on the CASP14 dataset relative to the other top-15 entries (out of 146 entries) ii) Correct domain packing predicted by AlphaFold of the CASP target T1044 (PDB 6VR4) (adapted from [4]). b) Superior performance of DeepMind 21 DFT functional benchmarked on each class of reactions from GMTKN55 database. MoM is the mean of the mean absolute error for each subbenchmark. c) Diagram of the hierarchy of terms relating artificial intelligence, machine learning and deep learning together with some examples of deep learning methods.

Molecular dynamics (MD) and Monte Carlo (MC) simulations attempt to understand material properties by running computer simulations based on fundamental physical principles. By employing the ideas from statistical mechanics, MD simulations aim to hasten material discovery as well as help understand underlying phenomena that might not be observable within an experimental setup. The ongoing improvements in computational power stimulate the capability of simulations, as larger and more complex systems can be simulated faster. These techniques have been successful across a wide range of problems in condensed matter physics or biophysics such as conformational changes, folding, and binding, furthering to bridge the gap between theory and experiment [5]. The main limitation arises from calculating time-consuming force fields from interatomic potentials between interacting particles that are often tabulated and require a fair deal of manual fine-tuning [6]. Moreover, as the scientific community is motivated to investigate more complex systems, traditional computational techniques lag behind the experimental community. For instance, systems such as DNA nanostars [7] have been investigated experimentally, yet their complex shape and many interaction sites make them difficult to be studied *in silico*.

Since 1980s, when the first interatomic potentials were being introduced [8], there have been efforts to improve both the accuracy and the computation speed. Machine learning (ML) potentials have entered the world of computational materials science as a way of constructing data-driven numerical interpolations of the fundamental quantum-mechanical interactions generated from *ab initio* DFT calculations [9]. Their ability to provide potentials of DFT accuracy at a reduced computational cost has proved a step in the right direction towards more accurate MD simulations [10]. Therefore, data-driven discovery definitely plays a crucial role in future scientific endeavours, which has also been referred to as the fourth paradigm of scientific discovery [11]. After the first three paradigms - empirical experimentation, analytical derivation and computational investigation - data science is hence bringing in powerful tools to aid in the scientific effort. Therefore, by providing data generated from fundamental methods, trainable models aim to capture the underlying laws in an efficient solution.

1.2 The Project

The goal of MD and MC simulations is to sample thermodynamic averages of microstates so that one can predict macroscopic observable variables of materials through computation only. To create such MD or MC trajectories, one has to evaluate potential energy functions, or forces, between interacting particles. Systems with more complex rigid bodies can be represented by composite bodies made of many point particles. Due to the high number of pair interactions between these point particles, the computation times increase enormously with increasing number of bodies, or increasing complexity of their shape and surface. Nevertheless, the configurational state of such composite bodies can still be sufficiently described only by the theoretical minimum number of degrees of freedom: the position of the centre of mass and the body's orientation. These composite bodies can then be simulated by a coarse-grained (CG) pair potential as a function of these degrees of freedom only. Such a potential has no obvious analytical form, hence data-driven ML techniques must be employed.

We aim to use neural ordinary differential equations (neural ODEs) to learn these CG potentials from MD trajectories. By generating all-particle trajectories of the composite body, we aim to create an automated pipeline that learns the CG potential. By reducing the required number of degrees of freedom, the ambition is to improve the computational scalibility of rigid body simulations with this reduced representation.

This thesis has an unusual structure, where we divide the project into three parts that incrementally build towards the project's final goal. After reviewing the literature and providing the basics of deep learning and neural ODEs, we present each project's part with their own method and results sections. Firstly, we use neural ODEs within a Hamiltonian Monte Carlo (HMC) simulation to sample toy potentials. Our aim is to show that neural ODEs can be used to learn ML potentials and that they can be used within HMC to sample thermodynamic averages. Secondly, we use neural ODEs to learn a pair-wise ML potential in a diatomic molecule. Afterwards, we employ this potential to simulate a triatomic molecule. Our aim is to show the feasibility of using pair-wise ML potentials learned with neural ODEs in simulations of more than one pair of interacting particles. Thirdly, we use neural ODEs to learn a CG potential on a short trajectory between two simple composite bodies. Our aim is to confirm the feasibility of learning CG potentials with neural ODEs. We also give the background to rigid body kinetics, highlighting our own implementation of integrating such dynamics. We combine the discussion of all parts together, giving the overall contextual significance of all the thesis' results.

1.3 Preliminary Notes

Hardware & Software

The project has been run as a collaboration between Dr. Rafael Góomez-Bombarelli's research group at Massachussets Institute of Technology (MIT) and Dr. Stefano Angioletti-Uberti's research group at Imperial College London. Training and simulations were performed on a variety of hardware, but the final results come from the SoftNanoLab workstation and the High Performance Computing (HPC) facilities at Imperial College London. The first two parts of the thesis have their code on the Github repository `jakublala/md-neural-ode`. The final part is in the repository `jakublala/coarsegrained-md-neural-ode`. Software used primarily includes `Python` with `PyTorch`, and `LAMMPS` with `OVITO`.

Notation and Units

The notation is consistent throughout all parts. Capitalised, bold notation refers to matrices (e.g. $\mathbf{X}$) consisting of vectors (e.g. $\mathbf{x}$). Vectors here refer to a representation of a multi-dimensional data point. The components of these vectors are given as scalars with a subscript (e.g. x_i). Nevertheless, subscripts are also used with the bold vector notation to describe the value for a specific particle in a system or a specific time step in a series. Any exceptions are clearly highlighted. As an example, positions of all particles is given by $\mathbf{X}$, where the position of a single particle i is given by $\mathbf{x}_i$. If we then consider the individual components of $\mathbf{x}_i$, we omit the particle's index and rather give the component's index as x_i, i.e. $\mathbf{x}_i = [x_1, x_2, x_3]$.

The simulations throughout use a system of reduced units. These are denoted by a special variable with a unit subscript (e.g. σ_{unit} and ϵ_{unit} are the units of reduced distance and energy respectively). The definitions of these units are given in figure captions, where the relevant variable appears.

2 Literature Review

2.1 Neural Ordinary Differential Equations

Deep learning is a subfield of machine learning that uses neural networks to create predictive models. After Rosenblatt's invention of the brain-inspired computing unit of NNs called the *perceptron* [1], it took several decades of research, two AI winters and immense improvements in computational capabilities for NNs to become the mainstream way too construct models. The key research developments were, for instance, the extension to multilayer perceptron networks and deeper neural nets, automatic differentiation, efficient GPU acceleration, or the invention of various NN architectures such as convolutional neural nets, transformers, or graph neural networks.

Neural ODEs are a very recent addition to the family of deep neural network models [12] and have since been applied and advanced to many different areas of AI research. By extending the idea of discrete recurrent neural networks (RNNs), they utilize the well-established *adjoint sensitivity method* [13] which has been successful in meteorology for several decades [14]. In terms of modelling time-series phenomena, the parameterized neural ODE gives the change (first derivative) in time rather than outputting a state at a new time. The details of the origin of neural ODEs, the adjoint method as well as the basics of neural networks are explained in Section 3.

Advantages

The advantages of neural ODEs are primarily memory efficiency, adaptive computation and continuous time-series modelling. Deeper NNs posses greater predictive powers as they can model more complex relationships. Nevertheless, during the predictive forward pass through the net, one has to store the intermediate quantities to be able to compute the gradients in the backward pass during training. Hence, a major bottleneck of deeper models is running out of memory [15]. By using the adjoint method, one does not need to store the intermediate quantities and can thus achieve a constant memory cost as a function of depth. However, this is sometimes at the cost of longer computation times. More than 100 years of the development of efficient and accurate ODE solvers also allows to adapt the trade-off between speed and accuracy of the predictive neural ODE. Lastly, compared to the more traditional RNNs, the input data no longer has to consist of discrete observations with specific intervals, but continuously-defined dynamics can naturally be used with arbitrary time steps to train a neural ODE.

Applications

The breakthrough paper by Chen et al. [12] benchmarks performance of neural ODEs on the well-established MNIST dataset for digit recognition [16]. They compare it with a residual neural network (ResNet, which is similar to RNNs) and a neural ODE where the gradients are backpropagated directly through the integrator without using the adjoint method (RK-Net). They show that the memory costs scales as $O(1)$ for neural ODEs compared to the ResNet, which scales with the number of layers, and the RK-Net, which scales with the number of evaluations in the ODE solver. The paper's authors also implemented the adjoint method into the `PyTorch` automatic differentiation module in the `torchdiffeq` [17] package that will form the code basis further in the project.

Further research of neural ODEs has currently branched out into enhancing the method and applying it to real problems. Advancements such as incorporating stochasticity into neural ODEs [18] or the ability to learn on irregular time-series data [19] made neural ODEs useful in economic modelling [20] as well as forecasting disease outbreaks [21] [22]. Success has been also been shown in transient modelling of electronic circuits [23]. As first-order ordinary differential equations are common in physics-based dynamical processes and neural ODEs are proposed to parameterize Hamiltonians [24], it is only a matter of applying this method to appropriate problems to enhance data-driven scientific discovery. For instance, Zhong et al. [25] have used neural ODEs to learn Hamiltonian dynamics with control. They consider physically-consistent models to control the pendulum, the CartPole and the Acrobot from OpenAI Gym [26], an open-source toolkit with toy problems and environments for reinforcement learning.

We can observe a gradual adoption of neural ODEs among the computational sciences. For example, Lee at al. [27] have proposed a parameterized extension to neural ODEs to learn multiple dynamics specified by an input parameter instance. Their work demonstrates the effectiveness in computational fluid dynamic problems, more specifically in one-dimensional inviscid flow modelled by the Burgers' equation,

two-dimensional chemically reacting flow, quasi-one-dimensional inviscid compressible flow expressed by the Euler equation and shallow water equations.

However, the materials science community has yet to fully appreciate this method, as there are only a handful of papers applying neural ODEs to materials phenomena. For instance, Chen et al. [28] predict the outcomes of spintronic experiments, or Zhang et al. [29] demonstrate the potential of learning dynamics in chemical kinetics and combustion modelling.

The most related work comes from Wang et al. [30], where they apply control theory to molecular simulations. They specifically show it is possible to bias molecular dynamics of simulations towards a target state by a set of macroscopic observables on an example of a 3D polymer folding into a helix. Although the control Hamiltonian dynamics they learn are not physical as the target helical fold has no physical origin, they are the first ones to apply neural ODEs to molecular dynamics. Their `PyTorch` implementation `diffMD` has served as the basis for our work. Compared to the original `torchdiffeq` package, `diffMD` uses the symplectic Velocity-Verlet integrator instead of the Runge-Kutta solver [31], which becomes crucial for MD simulations as explained in Section 4.1.

2.2 Coarse-Grained Molecular Simulations

Molecular Dynamics and Hamiltonian Monte Carlo

MD simulations are used to simulate the dynamics of nanoscale systems to predict macroscopic properties. As they obtain system configurations by integrating differential equations of motion, the timescales studied are restricted by the limits of the employed integrator. Therefore, to simulate phenomena at longer timescales often observed in conformational transitions in biophysics, one has to use a longer time step in the integrator that may then deviate the trajectory from the ergodic ensemble that ensures sampling the correct thermodynamic average for the macroscopic observable [32]. *Hamiltonian Monte Carlo* (HMC), or *Hybrid Monte Carlo*, is a combination of MD with MC that takes a correction MC step that keeps the system in the correct statistical ensemble. An alternative view is that the proposal step in the MC Markov chain uses MD for the proposal of the next configuration.

HMC was developed in the 1980s [33] and has recently become popular within the applied statistics community [34]. Betancourt [35] goes into detail about the efficiency of HMC to sample target distributions by exploiting the geometrical information about their surface. MC inefficiency comes from redundantly computing proposal steps for the next system configuration that is then rejected and not used as a sample. By exploiting the geometry of the target distribution, HMC aims to propose states that are more likely to be accepted, whilst still sampling enough of the phase space for a useful macroscopic average. Computational superiority of HMC over both MD and MC has been shown in a large-scale simulation of a $BaTiO_3$ phase transition [36]. A related hybrid MC/MD approach in simulating bond scission of proteins also shows the power of combining the two methods [37].

We aim to use HMC to help us alleviate the inaccuracies that might arise when integrating neural ODEs. Although inaccuracy comes with all integrators [38], NN-based solutions are often prone to exploding singularities which would be detrimental in our application. By employing the MC correction step, we can ensure the system's energy does not explode and correct ergodic sampling is achieved. The alternate and equally useful perspective is that we use neural ODEs in HMC for the proposal step. There was no work found on the use of neural ODEs to evolve the dynamics in HMC.

Machine Learning Potentials

The interactions between the particles within the system are governed by parameterized pair-wise interatomic potentials. First derivatives of these energy landscapes define the force fields used to propagate the system forward in time. There are many ways to define such a potential. Classical potentials, such as the Lennard-Jones potential, exhibit cheap $O(n)$ computational complexity and provide long timescales capabilities, but are fairly inaccurate as they are only empirical and their coefficients are weakly fine-tuned. More advanced potentials provide better accuracy at the cost of computation time. Semi-empirical methods based on Hartree-Fock formalism blend empirical and quantum mechanical data, but they scale as $O(n^2)$. DFT-based potentials then scale as $O(n^3)$ and post-Hartree-Fock methods, or self-consistent method, scale as $O(n^7)$, which starts to limit the system sizes that would be feasible to simulate in a reasonable amount of time [39]. To avoid the expensive computation of the Schrödinger's equation in DFT, ML potentials aim to provide a data-driven approximation of these expensive methods at the fraction of the cost. Note that our

work does not utilise DFT calculations as the fundamental ground truth that is being modelled. We rather use MD simulations to be the ground truth, where we aim to find a potential as a function of a coarser representation.

Using NNs as function approximators of interatomic potentials has been used for more than a decade now [40] [41] [42]. Behler-Parinello NNs from 2007 aim to learn free energy landscapes by training on DFT data whilst incorporating all of the relevant physical principles [43]. This is a notable difference from deep learning applications in computer vision or natural language processing tasks, where NNs are free to learn underlying patterns in data without many hard-coded laws or inductive biases. Therefore, in the natural sciences, it is useful to utilize the long history of physics research to restrict the possible predictions to the physically relevant ones [44]. A well established success has been shown with a deep convolutional neural network in the SchNet architecture [45]. Both have reached highly competitive prediction accuracies in terms of chemical compound space as well as configuration space. Their scalibility to large datasets shows a promising future towards extracting novel insights through data-driven discovery. State-of-the-art progress has been made in efforts such as Allegro [46] or NequIP [47].

These NNs are usually trained by either an energy- or a force-matching approach [48]. In energy-matching, the training datasets include atomic configurations as inputs and the associated energies as labels. In force-matching, the labels are then the forces on the atoms. In the latter, an additional gradient layer must thus be incorporated to propagate the training optimization from the matched force into the ML potential [49]. In Figure 2, we show our framework of evolving the system's dynamics to learn the potential energy landscape. The key difference is that we are neither matching the energies, nor the forces. We are rather learning the ML potential that, given the time for the system to evolve, would produce the target trajectory from our training dataset. This is the crucial, unconventional idea that has not yet been investigated in the current literature.

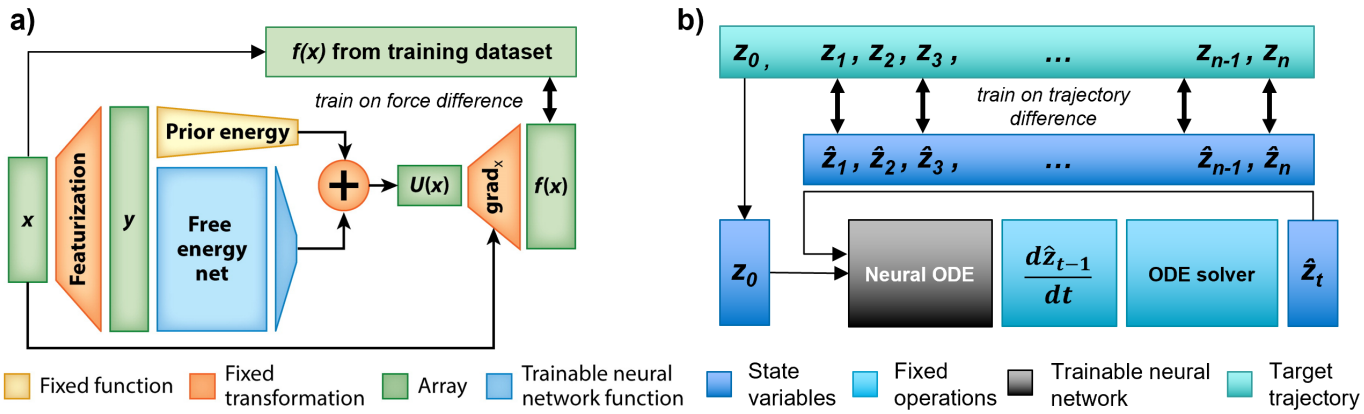


Figure 2: Comparison between the force-matching approach in CGNet by Wang et al. [49] and our trajectory based approach of learning ML potentials. a) Neural network scheme of CGNet, where atomic configurations $\mathbf{x}$ are fed into a neural network, the output gives the free energy and the gradient of the energy gives the force acting on the atoms, which is compared to the labels of the dataset (adapted from [44]) b) Our neural network scheme, where an initial system state is fed into an iterative integrator, where a neural network (black) takes in a state, outputs the change in time and is then integrated using an ODE solver. This produces a trajectory for n time steps, which is then compared to the trajectories from the training dataset. A detailed description of the inner components of the black-box neural ODE is given in Figs. 8 and 15 for translational and rotational dynamics respectively.

In terms of implementation, there are several Python libraries such as `torchMD` [50] or `JAXMD` [51] that provide end-to-end differentiable MD pipelines for learning ML potentials. Wang's `mdgrad` extends `torchMD` with the functionality of `diffMD` but has been not been developed for almost 2 years. Hence, we are going to be using `diffMD` that is effectively equivalent to `mdgrad` and is available only internally on the MIT Enterprise GitHub. Moreover, some packages such as `REANN` [29] can interface with the popular, efficient and easy-to-use open-source MD software `LAMPS` [52]. As we deem `LAMMPS`'s computational efficiency and community adoption [53] as important, the aim is to create an automated coarse-graining pipeline that is compatible with this powerful MD software. Packages in `LAMMPS` such as `ML-IAP` or `ML-SNAP` are essential interfaces to do just that.

Coarse-Graining

Coarse-graining (CG) is the process of representing a thermodynamic system in a coarser set of variables. For instance, a protein backbone or DNA could be modelled as a chain of beads that have lower than atomistic resolution [54] [55] [56]. The goal is thus to define a CG potential as a function of the coarse-grained variables that closely matches the original free energy landscape, hence preserving as much of the physical properties as possible. Traditional approaches involve manual fine-tuning to capture multi-scale interactions [57], which by construction often leads to information loss at the trade-off of improved efficiency [58]. Systems with complicated shapes, surface charges and interaction sites can often be computationally demanding hence coarse-graining the system becomes necessary for any meaningful ensemble sampling. For instance, DNA nanostars [7] are usually simulated with the `oxDNA` package [59], where the DNA backbone is treated as a string of rigid nucleotides that interact only through a potential based on the position and orientation of the nucleotides.

The aforementioned CGNets, introduced by Wang et al. [49], provide a better approximation than functional forms of CG potentials, as CGnets automatically include multibody effects and nonlinearities. They provide evidence of accurate approximations for alanine dipeptide and chignolin folding/unfolding in water. Other attempts at coarse-graining involve transforming high-dimensional configurations into low-dimensional embeddings via autoencoders [60] [61], VAMPnets [62], graph NNs [63] or kernels [64]. All of these show promising future for using ML as automated, data-driven CG models.

Most related work to our project comes from Greener et al. [65], where they use a NN to parameterize a physics-informed three-component CG force field of proteins. Firstly, they constrain the CG potential into a functional form of a pair-wise distance, bond angle and torsion angle components. Although these are well-established types of potentials for protein simulations, the approach underutilises the ability of NNs to capture interactions outside of these three pre-defined contributions. Secondly, their method does not use neural ODEs and hence their forward propagation through the RNN has large memory costs as all intermediate computations have to be stored. This limits the amount of time steps taken forward in time during training. We aim to address both of these limitations.

2.3 Summary

Neural ODEs introduce a fresh perspective of applying NNs to learning dynamical systems. Its use has, however, yet to be fully appreciated by the materials science research community. As natural phenomena are known to obey first-order ODEs, it is only a matter of time once neural ODEs will be become more popular within the community. Compared to conventional ML potentials, the data-driven optimization is performed on dynamical trajectories rather than structure-energy pairs. This makes them the ideal candidates to learn effective potentials as a function of coarse-grained variables from MD trajectories. Figure 3 sums up our intended application. Successfully applying neural ODEs to learn CG potentials of composite rigid bodies might not only deliver a superior method towards defining CG potentials, but it might also help introduce this novel technique to the MD community. Nevertheless, it is clear that neural ODEs trade in low memory cost at the expense of higher computation times, hence it might still require further computational optimization advancements before they will be able to compete with some of the more traditional methods.

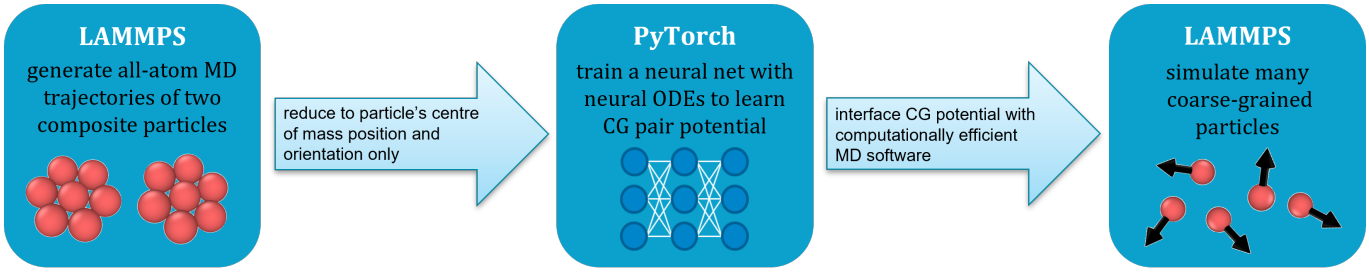


Figure 3: Schematic diagram of the automated coarse-graining pipeline.

3 Theory of Neural ODEs

Neural Networks

A *neural network* is a system of simple computing neurons organised in successive layers that can theoretically approximate any function $f(\mathbf{X})$ [66]. Figure 4a shows a perceptron j at depth d in a neural net. Each neuron sums up the result of multiplying the inputs σ_i^{d-1} with some weights w_{ij}^d, adds in some bias b_j^d (Eq. 1), runs the result through an activation function $\sigma_j^d(a_j^d)$, and then sends the result σ_j^d to the neurons in the next layer. Indices i and j denote the position of the neurons within each layer. Each neuron is thus parametrized by b_j^d and each neuron connection is parametrized by w_{ij}^d. Therefore, the NN can be described as a function approximator $f(\mathbf{X}, \theta)$, where θ encapsulates all the weights and biases.

$$a_j = \sum_i w_{ij}^d \sigma_i^{d-1} + b_j^d \tag{1}$$

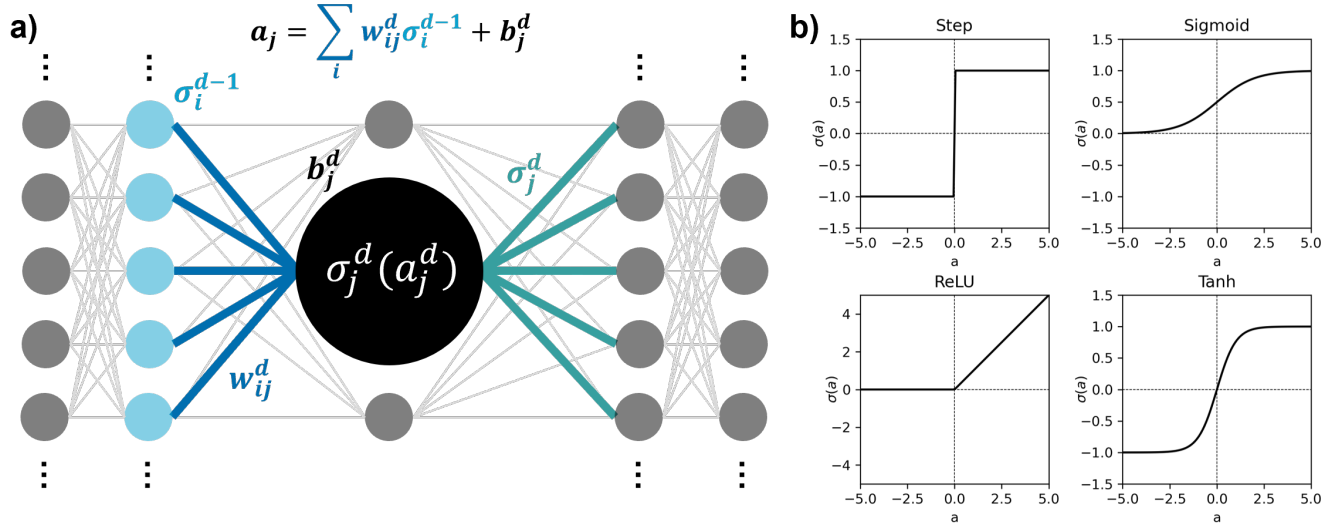


Figure 4: a) Schematic diagram of a single neuron inside a neural network. It takes in values from the previous layer σ_i^{d-1}, multiplying them with their respective weights w_{ij}^d, summing them up, adding a bias b_j^d and then running them through an activation function σ_j^d. The result is then sent to the next layer. b) Examples of four activation functions for $\sigma(a)$.

To optimize θ, the *backpropagation algorithm* is used. To learn our parameters in a supervised training scheme, we need to have a labelled dataset. That is, we need a corresponding set of the input-output pairs $\{\mathbf{X}, \mathbf{Y}\}$, where $\mathbf{X}$ and $\mathbf{Y}$ are matrices build from individual input (feature) and label (target) data points respectively. If our NN has a prediction $f(\mathbf{X}, \theta) = \hat{\mathbf{Y}}$, then we can define a *loss function* $L(\mathbf{Y}, \hat{\mathbf{Y}})$, which captures the difference of our predicted output to the true label. As we want to minimize $L(\mathbf{Y}, \hat{\mathbf{Y}})$, we can update θ by *gradient descent* optimization (Eq. 2). To find the gradients with respect to the parameters, we need to backpropagate the gradients through the network. This is done by essentially applying the chain rule backwards through the computation graph of the NN. An example of this is given in Appendix A.1.

Gradient descent is then performed by a step size η, also known as the *learning rate*, that determines the amount of how much we want to update the parameters based on the computed gradient of the loss. We update θ for a fixed number of epochs, or until the gradients become sufficiently small.

$$w_{ij}^d \to w_{ij}^d - \eta \frac{dL}{dw_{ij}^d} \qquad\qquad b_i^d \to b_i^d - \eta \frac{dL}{db_i^d} \tag{2}$$

To utilize the enhancing power of graphic cards, all of the above math operations may be re-written as matrix operations that can be easily parallelized on GPUs, taking full advantage of their computational capacity. Moreover, instead of optimizing θ by passing in a single data point through the neural network at a time, we might parallelize the evaluation through the network as well by sending in multiple inputs at once. This is called *batching*. Determining the batch size also affects the training performance and the model's accuracy.

As increasing the size of the NNs greatly improves their ability to approximate complicated functions, these models are very prone to overfitting on the *training dataset* $\{\mathbf{X}_{\text{train}}, \mathbf{Y}_{\text{train}}\}$. Therefore, a *test dataset*

$\{\mathbf{X}_{\text{test}}, \mathbf{Y}_{\text{test}}\}$ is used during learning that ensures $f(\mathbf{X}, \theta)$ performs well even on never-seen data. A separate *validation dataset* $\{\mathbf{X}_{\text{validation}}, \mathbf{Y}_{\text{validation}}\}$ is then used for the evaluation metric for *hyperparameter* tuning. For instance, important hyperparameters is the depth and the width of the NN, the learning rate, or the number of training epochs. One would perform at least a grid search to find a reasonable set of hyperparameters, although more robust approaches may be used such Bayesian optimization [67] or genetic algorithms [68]. We use `SigOpt` [69] as a black-box that performs intelligent optimization and finds hyperparameters that perform relatively well. The model's performance is then thus evaluated by $L(\mathbf{Y}_{\text{test}}, \hat{\mathbf{Y}}_{\text{test}})$. Also, one needs to consider weight and bias initialization for the network to achieve stable and efficient training. We use Glorot initilization [70], where initial parameters are sampled from a small Gaussian distribution centered around zero. The reasoning is that such parameters describe a system with no dynamics, which is assumed to be better for the model to start learning from. If we would initialize the model with some unrelated random dynamics, the model might find it harder to re-learn [71].

PyTorch Library

Thankfully, all of the theoretical background has already been implemented in various Python deep learning frameworks such as `TensorFlow` [72], `JAX` [73] or `PyTorch` [74]. All of these are effectively equivalent and only vary in the way the programmer interacts with the network and the way the computation graphs are created, stored and evaluated. This project uses `PyTorch` for all its parts.

In terms of the update rule, there are various optimizers implemented in `PyTorch`, where each uses a different approach to estimate the topology of the objective function $L(\mathbf{Y}, f(\mathbf{X}, \theta))$ as a function of θ. We use the common `Adam` optimizer, which is a first-order gradient-based optimization algorithm of stochastic objective functions that adapts its learning rate, penalizing the learning rates for parameters that are updated frequently and increasing the learning rates of those that are not [75]. As these adaptive learning rate algorithms vary in performance based on the specific task, we use `LambdaLR` scheduler (see Appendix A.2) to periodically decrease the learning rate after some number of epochs. Note that the scheduling hyperparameters should not have much effect on training, as `Adam` is already an adaptive learning rate algorithm, hence the scheduler only imposes an upper bound on the learning rate. That helps as the parameters θ approach the minimum of the loss function $L(\mathbf{Y}, f(\mathbf{X}, \theta))$ and the updates should hence be smaller so as to not overshoot the local minimum.

Reccurent Neural Networks

To model time-series data, it is useful to consider that the network provides a discrete update of some hidden state variables $\mathbf{Z}$. Therefore, instead of transforming our inputs $\mathbf{X}$ to outputs $\mathbf{Y}$, we iteratively change $\mathbf{Z}$ by feeding $\mathbf{Z}_t$ into successive computation units, where $\mathbf{Z}_t$ is a sum of the output from the previous computation $f(\mathbf{Z}_{t-1}, \theta_{t-1})$ as well as its input $\mathbf{Z}_{t-1}$. To achieve generalised dynamics, we compute all time steps by the same NN with the same parameters θ, which is then called a *recurrent neural network*.

These discrete, successive operations thus compose a sequence of transformations to a hidden state

$$\mathbf{Z}_{t+1} = \mathbf{Z}_t + f(\mathbf{z}_t, \theta) \tag{3}$$

which is equivalent to the iterative iterations as seen in Euler discretization [76]. One can then imagine the limit of infinitely many discrete computations, or infinitely small time steps (shown in Figure 5), which transforms the above expression into an ODE as follows

$$\frac{d\mathbf{Z}(t)}{dt} = f(\mathbf{Z}(t), t, \theta) \tag{4}$$

Therefore, to evaluate the state at any time, one can employ a wide range of ODE solvers to evolve the hidden state in time. Note that in Eq. 4 the neural net is considered to also be a function of time t, which could also sometimes be referred to as the depth of the network when the ODE solver recurrently performs the forward pass through the net.

Adjoint Method

It is not straightforward to compute the gradients through the ODE solver during the backward pass of the backpropagation algorithm. The adjoint method is a way to compute the gradients by solving a second, augmented ODE backwards in time and is applicable to all ODE solvers [12].

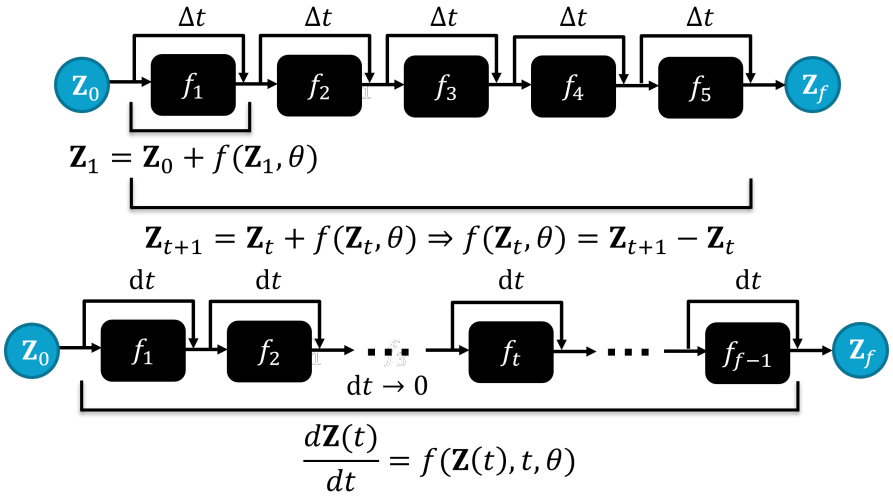


Figure 5: Derivation of neural ODEs from a recurrent computation by the same computational unit f of state $\mathbf{Z}_0$ to state $\mathbf{Z}_f$. Thinking of each computational unit being at some time t and giving us a change in time of Δt, one can realise that at $\lim_{\Delta t \to 0}$ we arrive at the continuous limit, where the continuous change of $\mathbf{Z}(t)$ in time defines an ODE of f parametrised by time t.

As before, our goal is to optimize a loss function $L(\mathbf{Z}(t_f))$, where the input is the result of the ODE solver at final time t_f. Hence we require the gradients of $L(\mathbf{Z}(t_f))$ with respect to parameters θ. This can be summarized by calculating three ODEs as given in the original paper [12] as follows

$$\mathbf{Z}(t_f) = \int_{t_0}^{t_f} \frac{d\mathbf{Z}(t)}{dt} dt = \int_{t_0}^{t_f} f(\mathbf{Z}(t), t, \theta) dt \tag{5}$$

$$\mathbf{A}(t_0) = \int_{t_f}^{t_0} \frac{d\mathbf{A}(t)}{dt} dt = \int_{t_f}^{t_0} -\mathbf{A}(t)^\top \frac{\partial f(\mathbf{Z}(t), t, \theta)}{\partial \mathbf{Z}} dt \tag{6}$$

$$\frac{\partial L}{\partial \theta} = \int_{t_f}^{t_0} \mathbf{A}(t)^\top \frac{\partial f(\mathbf{Z}(t), t, \theta)}{\partial \theta} dt \tag{7}$$

where $\mathbf{A}(t)$ is the adjoint given by

$$\mathbf{A}(t) = \frac{\partial L}{\partial \mathbf{Z}(t)} \tag{8}$$

whose dynamics are governed by an ODE integrated in Eq. 6 and can be thought of as the instantaneous analog of the chain rule. Note that this theoretically restrains to the use of continuously differentiable activation functions [24].

To sum up, Eq. 5 computes the trajectory forward in time, Eq. 6 then computes the adjoint backwards in time and then finally Eq. 7 computes the required gradients for the update rule. Thankfully, these separate integral calls may be simplified in a single call to an ODE solver by concatenating, also known as *augmented state dynamics* (see Appendix A.3).

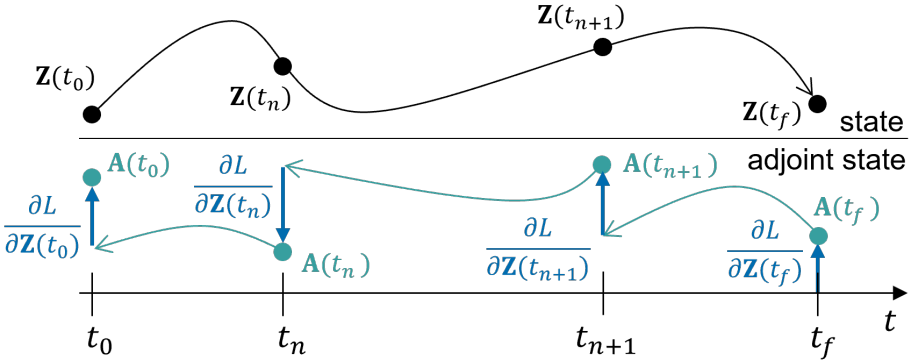


Figure 6: Reverse-mode differentiation of an ODE solution using the adjoint method (re-created from [12])

4 Learning Toy Potentials

4.1 Method

Statistical Mechanics Sampling

In statistical mechanics [77], when one wants to determine the value of a macroscopic variable A, one needs to evaluate the ensemble average over all the microstates of the system as follows

$$\langle A \rangle = \int_Z A(\mathbf{Z}) \frac{e^{-\beta H(\mathbf{Z})}}{Q} d\mathbf{Z} \tag{9}$$

where $\mathbf{Z}$ are the system coordinates, $A(\mathbf{Z})$ is the function of the observable, $\beta = \frac{1}{k_b T}$ (where k_b is the Boltzmann constant and T is the temperature), $H(\mathbf{Z})$ is the Hamiltonian of the system, and Q is the canonical partition function, i.e. $Q = \int_Z e^{-\beta H(\mathbf{Z})} d\mathbf{Z}$.

As the system coordinates $\mathbf{Z}$ consist of the positions $\mathbf{X}$ and the momenta $\mathbf{P}$, where only the integration over momenta $\mathbf{P}$ can be calculated analytically, numerical techniques have to be introduced to integrate over positions $\mathbf{X}$ and thus determine an approximation to the ensemble average $\langle A \rangle$. Sampling techniques are a way of approximating this integral as

$$\hat{A}_N = \frac{1}{N} \sum_{n=0}^{N} A(\mathbf{Z}_n) \tag{10}$$

where N is the number of drawn samples and $\mathbf{Z}_n$ is the system coordinates of sample n. By taking enough samples, $\lim_{N \to \infty} \hat{A}_N = \langle A \rangle$.

As a naive grid search to evaluate this integral would be tedious, clever approaches have been developed to sample only the important regions of the phase space. Here we consider molecular dynamics and Monte Carlo sampling. As we explain MC sampling, we are going to show how it can be utilised with MD to describe the so-called Hamiltonian Monte Carlo sampling scheme.

Molecular Dynamics

The idea behind MD is to evolve many-particle systems by using Newton's equations of motion. We are therefore essentially performing a physical experiment inside a computer, where we observe the evolution of particle trajectories as we iterate through the time. This samples microstates from the desired ensemble.

In Hamiltonian mechanics, the total energy of the system, i.e. the Hamiltonian, is given by

$$H(\mathbf{X}, \mathbf{P}) = K(\mathbf{P}) + V(\mathbf{X}) = \sum_{i=0}^{N} \frac{\mathbf{p}_i^2}{2m_i} + \sum_{i,j} V(\mathbf{x}_i, \mathbf{x}_j) \tag{11}$$

where the kinetic energy K is only dependent on the individual particle's momenta $\mathbf{p}_i$ (where m_i is the particle's mass), and the potential energy V is only dependent on positions $\mathbf{X}$ and is separable into pair-wise contributions between particles i and j. The coupled Hamilton's equations of motion are then

$$\frac{d\mathbf{X}}{dt} = \frac{\partial H(\mathbf{X}, \mathbf{P})}{\partial \mathbf{P}} \tag{12}$$

$$\frac{d\mathbf{P}}{dt} = -\frac{\partial H(\mathbf{X}, \mathbf{P})}{\partial \mathbf{X}} \tag{13}$$

These are the underlying equations we are going to use to evolve our system. If we expand them using Eq. 11 and look at the change of particle i in terms of its position $\mathbf{x}_i$ and $\mathbf{p}_i$ then

$$\frac{d\mathbf{x}_i}{dt} = \frac{\mathbf{p}_i}{m_i} = \mathbf{v}_i \tag{14}$$

$$\frac{d\mathbf{p}_i}{dt} = \sum_{i,j} -\frac{\partial V(\mathbf{x_i}, \mathbf{x_j})}{\partial \mathbf{x}_i} = \mathbf{F}_{ij} \tag{15}$$

Velocity-Verlet Integration

As the equations of motion in Eq. 14 and 15 are differential equations, we need to use an integration algorithm to propagate these equations forward in time. Velocity-Verlet is a well-known numerically stable and simple algorithm used in MD. It is exactly time-reversible and given conservative forces it also conserves linear momentum. More importantly, it has excellent energy-conserving properties even with long time steps, which is essential when simulating MD in the NVE canonical ensemble [38]. Below we give the leapfrog version of the algorithm written in terms of momenta for a single time step as

1. momentum at half-step ('half-kick', $\mathbf{x}_i$ constant)
$$\mathbf{p}_i\left(t+\frac{\Delta t}{2}\right) = \mathbf{p}_i(t) + \frac{\Delta t}{2}\frac{d\mathbf{p}_i}{dt}\Big|_t \tag{16}$$
2. position at full-step ('drift', free flight with $\mathbf{p}_i$ constant)
$$\mathbf{x}_i\left(t+\Delta t\right) = \mathbf{x}_i(t) + \frac{\Delta t}{m_i}\,\mathbf{p}_i\left(t+\frac{\Delta t}{2}\right) \tag{17}$$
3. momentum at full-step ('half-kick', $\mathbf{x}_i$ constant)
$$\mathbf{p}_i\left(t+\Delta t\right) = \mathbf{p}_i\left(t+\frac{\Delta t}{2}\right) + \frac{\Delta t}{2}\frac{d\mathbf{p}_i}{dt}\Big|_{t+\Delta t} \tag{18}$$

Hamiltonian Monte Carlo

In an MC algorithm, the integral from Eq. 9 is computed by sampling from a probability distribution set up as a Markov chain, where each new sample is proposed and accepted as to obey *detailed balance*. The Metropolis-Hastings algorithm can be summarized as follows

1. initial system configuration
$$\mathbf{X}_0$$
2. proposal of a new configuration by random walk
$$\mathbf{X}_n + \delta\mathbf{X} \rightarrow \mathbf{X}_{n+1}$$
3. accept the new configuration according to the acceptance probability
$$a(\mathbf{X}_{n+1}|\mathbf{X}_n) = \min\left(1, \frac{\exp\left(-\beta H(\mathbf{X}_{n+1})\right)}{\exp\left(-\beta H(\mathbf{X}_n)\right)}\right) \tag{19}$$

To improve sampling of important regions in phase space, HMC instead gives the system a random momentum in Step 2. and then evolves it in time using Hamiltonian dynamics to get a new proposal. An example of such HMC trajectories is given in Figure 7.

To obey detailed balance, the acceptance in Eq. 19 has to change, modifying the HMC algorithm as

2. proposal of a new configuration via sampling of a random momentum and then evolve it in time
$$\mathbf{X}_n \rightarrow (\mathbf{X}_n, \mathbf{P}_n)$$
3. accept the new configuration according to the acceptance probability
$$a\big((\mathbf{X}_f, \mathbf{P}_f)|(\mathbf{X}_o, \mathbf{P}_o)\big) = \min\left(1, \frac{\exp\left(-\beta H(\mathbf{X}_f, -\mathbf{P}_f)\right)}{\exp\left(-\beta H(\mathbf{X}_o, \mathbf{P}_o)\right)}\right) \tag{20}$$

The choice of momenta sampling is crucial here as it determines the target sets one is going to explore within the energy landscape. We choose the Euclidean-Gaussian kinetic energies defined by

$$K(\mathbf{X}, \mathbf{P}) = \frac{1}{2}\mathbf{P}^T\mathbf{M}^{-1}\mathbf{P} + \log|\mathbf{M}| + \text{constant} \tag{21}$$

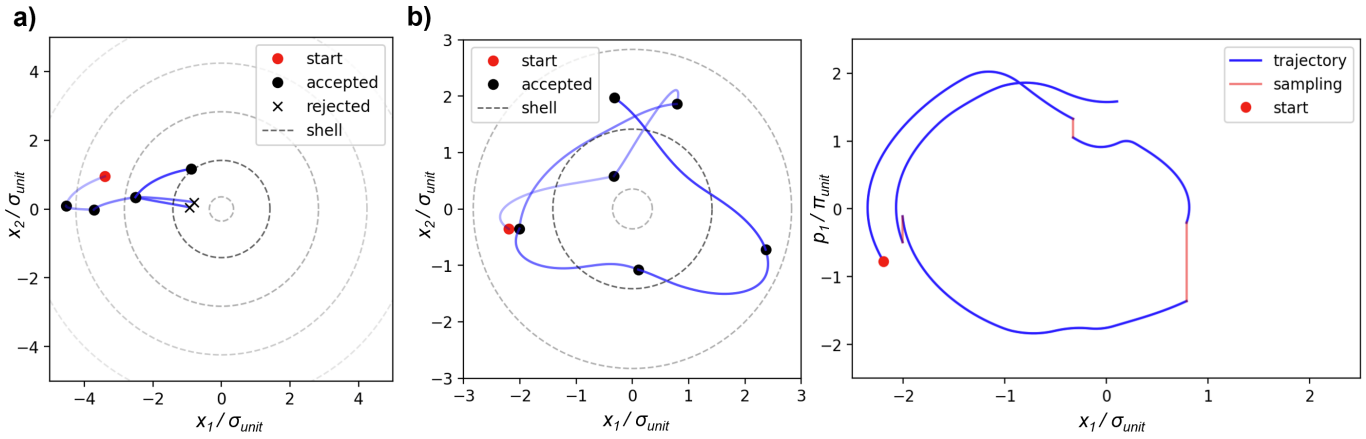


Figure 7: Example HMC trajectories inside a 2D Shell potential. a) Three accepted successive steps are followed by two rejections. Dashed lines show the contours of the potential, where darker shade refers to lower energy. b) Left: All steps are accepted. Right: Phase space in the first dimension showing the discontinuity when sampling of new momentum occurs (red line).

where $\mathbf{M}$ is in the physical perspective also called the *mass matrix* and it defines the sampled Gaussian distribution of momenta for particle i as

$$\mathbf{p}_i \sim \mathcal{N}(0, \mathbf{M})\,. \tag{22}$$

where the variance is approximated by the covariance of the most recent sampled positions as

$$\mathbf{M}^{-1} = \mathbb{E}[(\mathbf{X} - \mu)(\mathbf{X} - \mu)^T] \tag{23}$$

where μ is the mean of the recent $\mathbf{X}$ used for sampling. This dynamics proposal step is where we aim to use neural ODEs to evolve our trajectories. We are going to abbreviate this implementation of neural ODEs within the HMC sampling scheme as *neural HMC*.

Figure 8 shows how we set up the ODEs in a physics-informed way, where the change in position is defined explicitly from the momentum and hence we only need to parametrize the potential. Here note that if one would employ neural ODEs to learn the coupled differential equations of motion together without any physics-informed structure (i.e. a single net outputs two ODE approximations), then the acceptance rule (Eq. 20) would have to be adjusted as neural ODEs would no longer be phase space preserving [78]. A brief outline of this adjustment is given in Appendix A.4.

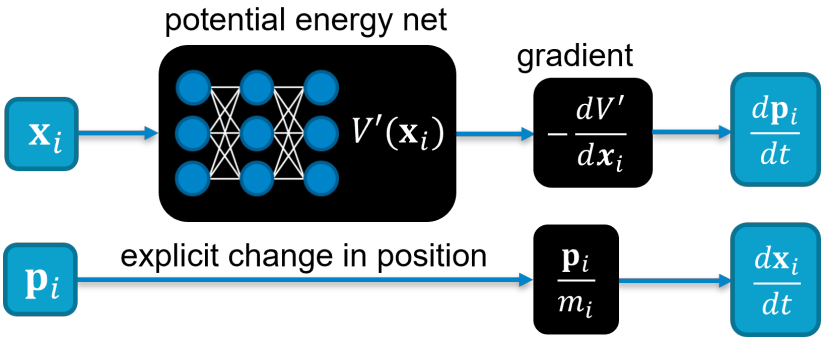


Figure 8: Schematic diagram of state variable evolution of the potential $V'(\mathbf{x}_i)$ approximating $V(\mathbf{x}_i)$.

Experiment

Hamiltonian dynamics trajectories were generated for three different potentials - 2D Shell, 10D Gaussian and 2D Wolfe-Quapp. We are effectively simulating a single particle in a potential well, hence we omit particle index i in the notation and the positions and momenta are only $\mathbf{x}$ and $\mathbf{p}$. These are defined respectively as

$$V_{\text{Shell}}(\mathbf{x}) = \frac{|r - r_0|}{\alpha^{\text{Shell}}} \quad r = ||\mathbf{x}|| \tag{24}$$

where $||\mathbf{x}||$ is the norm of position $\mathbf{x}$, r_0 is the radius of the shell (set to $\sqrt{2}$) and α^{Shell} is the slope of the shell (set to 0.5). The shell potential was chosen to investigate the properties of neural ODEs in an unusual energy landscape with a discontinuity in the force, as the first derivative is discontinuous at the *canyon* of the shell at $r = r_0$ as well as the origin $r = 0$.

$$V_{\text{Gaussian}}(\mathbf{x}) = \frac{1}{2} \sum_i \frac{x_i}{\alpha_i^{\text{Gauss}}} \tag{25}$$

where α_i^{Gauss} is a constant (set to 1 for all i) and i denotes the dimension (where we simulate in 10 dimensions). A Gaussian potential is a common energy landscape to be tested. Here, we purposely tackle more dimensions to investigate neural ODE behaviour when scaling the number of degrees of freedom.

$$V_{\text{Wolfe-Quapp}}(\mathbf{x}) = x_1^4 + x_2^4 - 2x_1^2 - 4x_2^2 + x_1 x_2 + 0.3x_1 + 0.1x_2 \tag{26}$$

where x_1 and x_2 are the positions in the two dimensions. This toy potential is interesting to investigate as the dominant fluctuations that lead from one minima to the other are in an oblique direction with respect to the two dimensions [79]. Figure 9 shows the three investigated potentials.

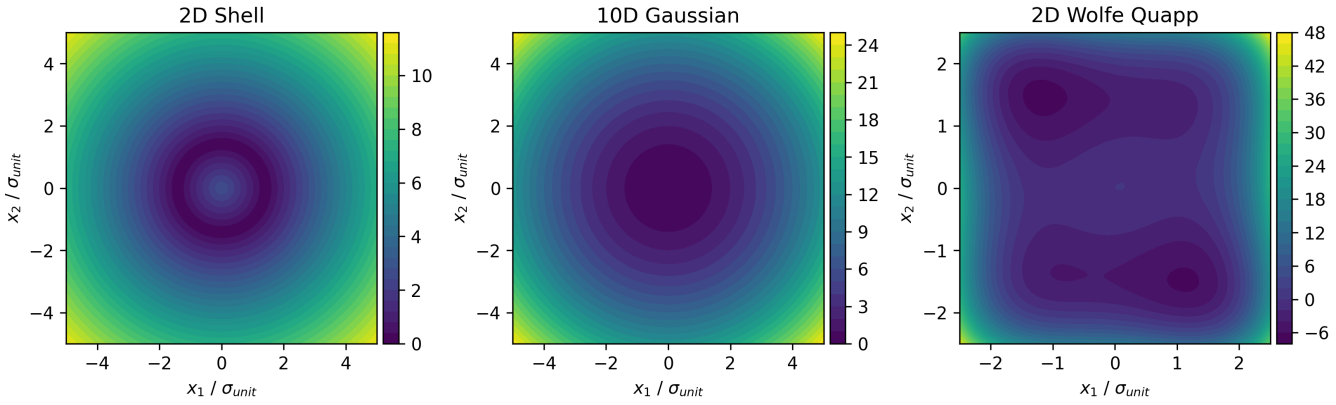


Figure 9: Potential energy contour plots of the three toy potentials - 2D Shell, 10D Gaussian (first two dimensions) and 2D Wolfe Quapp.

The training dataset of positions and momenta $\{\mathbf{X}, \mathbf{P}\}$ was generated from HMC simulations for 20 000 samples, where the momentum was sampled based on the covariances of the 500 most recent positions. The trajectories come from the dynamics evolved during the proposal step. These dynamics were run for 100 time steps, where for the 2D Shell the step size was 0.01 and for the others it was 0.1. The choice of the time step was chosen so that the total energy has relatively low fluctuations. A test dataset is then also generated with the same parameters but only 5 000 trajectories.

All NNs used are 2-layers deep and 50-neurons wide. Apart from the final layer neuron with a linear activation function, all other neurons used Sigmoid as the activation function. The `Adam` optimizer was used for 20 000 epochs with the `LambdaLR`, where the scheduling frequency was every 2 000 epochs and the scaling factor was 0.6. The initial learning rates used were 0.1 for the 2D Shell and 10D Gaussian, while 0.025 was used for the 2D Wolfe-Quapp potential. The batch size was 800 trajectories of 10 time steps each.

This part of the investigation has three subtasks done for each potential. Firstly, we investigate **the effect of loss function choice**. For each potential, we run 5 independent training procedures with the same hyperparameters as above (but only 10 000 epochs) for three different scenarios. We either define the loss based on the mean absolute difference in position (Eq. 27), momentum (Eq. 28), or both (Eq. 29).

Here we use the entire training dataset and hence evaluate the performance using the test dataset. For completeness, the loss functions go as follows

$$L(\mathbf{X}, \hat{\mathbf{X}}) = \frac{1}{NTd} \sum_{n=1}^{N} \sum_{t=t_0}^{t_f} \frac{|\mathbf{X}_n(t) - \hat{\mathbf{X}}_n(t)|}{\sigma_{\text{unit}}} \tag{27}$$

$$L(\mathbf{P}, \hat{\mathbf{P}}) = \frac{1}{NTd} \sum_{n=1}^{N} \sum_{t=t_0}^{t_f} \frac{|\mathbf{P}_n(t) - \hat{\mathbf{P}}_n(t)|}{\pi_{\text{unit}}} \tag{28}$$

$$L(\mathbf{X}, \hat{\mathbf{X}}, \mathbf{P}, \hat{\mathbf{P}}) = \frac{1}{2NTd} \sum_{n=1}^{N} \sum_{t=t_0}^{t_f} \left(\frac{|\mathbf{X}_n(t) - \hat{\mathbf{X}}_n(t)|}{\sigma_{\text{unit}}} + \frac{|\mathbf{P}_n(t) - \hat{\mathbf{P}}_n(t)|}{\pi_{\text{unit}}} \right) \tag{29}$$

where N denotes the number of trajectories in a batch, T is the number of time steps between times t_0 and t_f and d is the dimensionality. Note that we formalize the loss by normalizing the difference with the position and momenta units σ_{unit} and π_{unit} respectively, but this should not theoretically affect the results as $\pi_{\text{unit}} = \sigma_{\text{unit}}/\tau_{\text{unit}}$ and so the two are inherently related and scale appropriately already. The loss is then considered to be unitless.

Secondly, we obtain the **learning curves of the test loss** with respect to the number of training trajectories used. We use the best performing loss function from the first subtask above. We run 5 independent training procedures at training sizes of 100, 200, 500, 1 000, 2 000, 5 000 and 10 000 trajectories.

Thirdly, we learn 5 independent potentials whose performance we evaluate by **HMC sampling**. We therefore run 25 regular HMC runs using the usual Verlet dynamics proposal mechanism, obtaining 100, 200, 500, 1 000, 2 000, 5 000 and 10 000 MC samples as a benchmark reference. Then for each of the independent NN models, we run 5 independent runs of neural HMC for the same number of MC samples, thus giving us 25 neural HMC trajectories. For these neural HMCs, we also separate between two scenarios, where the Hamiltonian in the acceptance probability (Eq. 20) is calculated by either the ML potential or the exact expression of the potential (Eqs. 24, 25 and 26).

4.2 Results

Loss Function Choice

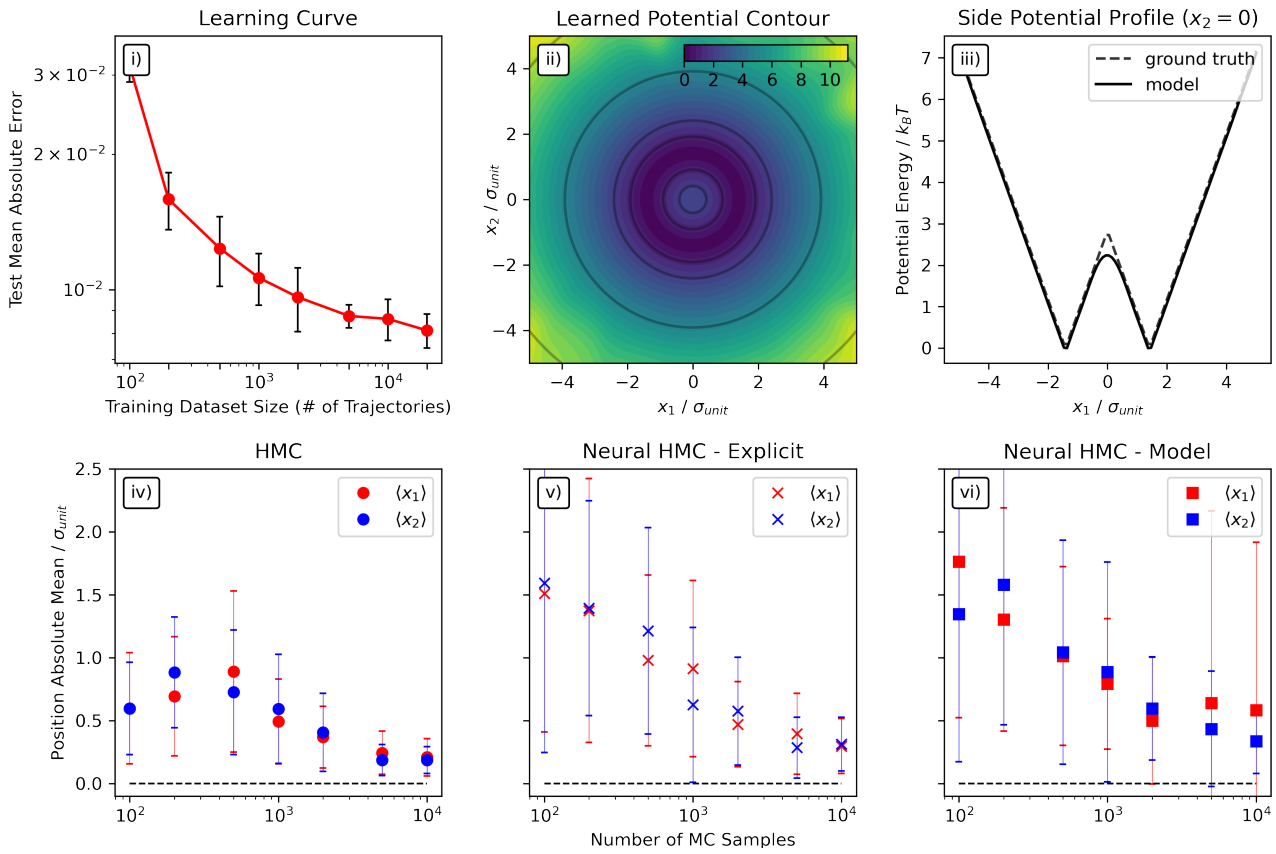
Table 1 shows the loss function considering both positions and momenta performs relatively well across all the tested potentials. The uncertainties are relatively high for the 10D Gaussian and the 2D Wolfe-Quapp, which is to be expected due to their more complex nature than the 2D Shell. These results suggest that it is reasonable to use $L(\mathbf{X}, \hat{\mathbf{X}}, \mathbf{P}, \hat{\mathbf{P}})$ as the loss function captures the maximum possible discrepancy between the true and the predicted trajectories. There are no significant gains to only learning on the positions or the momenta.

Table 1: Results of learning various ML potentials with neural ODEs using different definitions of the loss function. The hat symbol refers to the predicted variable by the model. The test loss is defined by the mean absolute difference of the trajectories including both the positions and momenta. The uncertainty comes from the standard deviation of 5 independent trials.

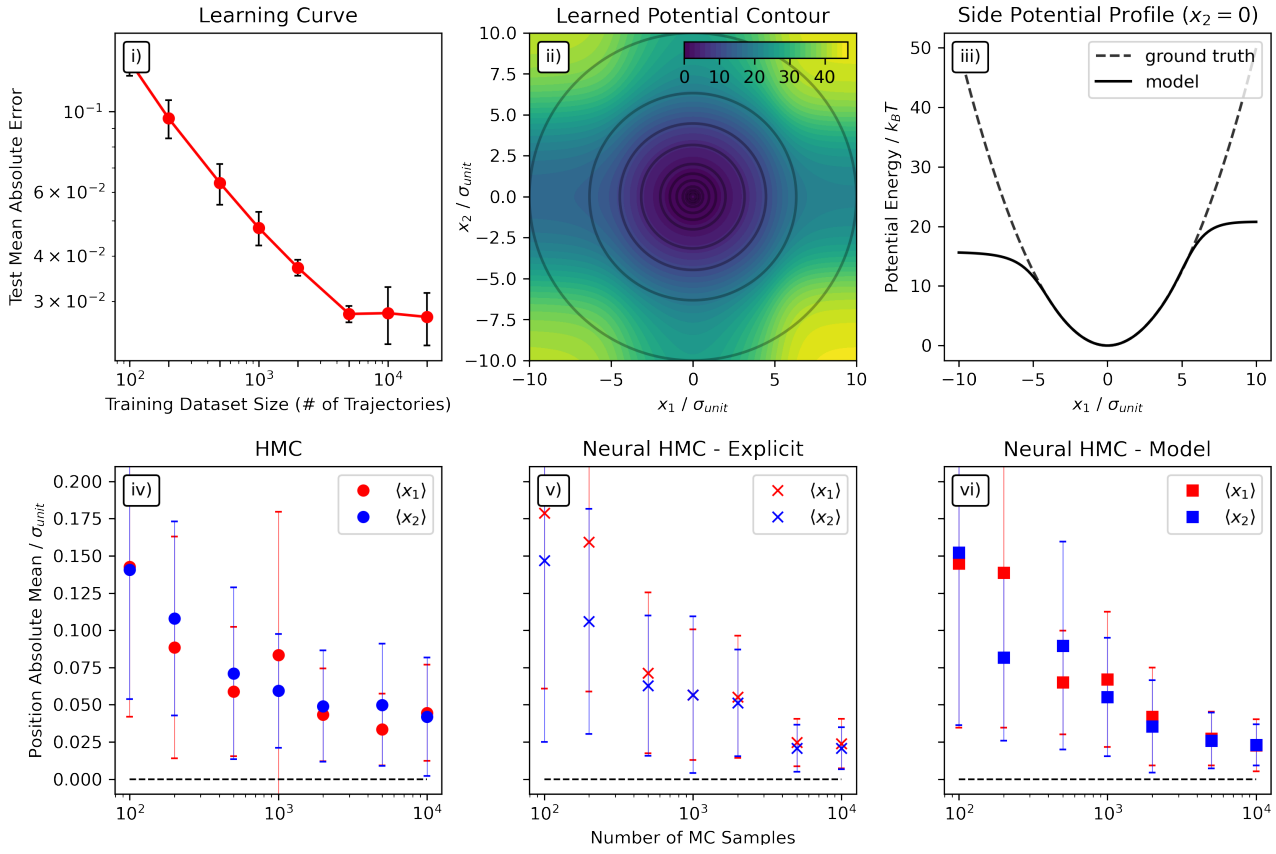
	Test Loss		
Loss Function	2D Shell	10D Gaussian	2D Wolfe-Quapp
$L(\mathbf{X}, \hat{\mathbf{X}})$	0.31058 ± 0.00665	0.02739 ± 0.00365	0.12252 ± 0.02130
$L(\mathbf{P}, \hat{\mathbf{P}})$	0.28479 ± 0.00033	0.02217 ± 0.00295	0.18427 ± 0.06049
$L(\mathbf{X}, \hat{\mathbf{X}}, \mathbf{P}, \hat{\mathbf{P}})$	0.28198 ± 0.00009	0.02323 ± 0.00218	0.12123 ± 0.01693

HMC and Neural HMC

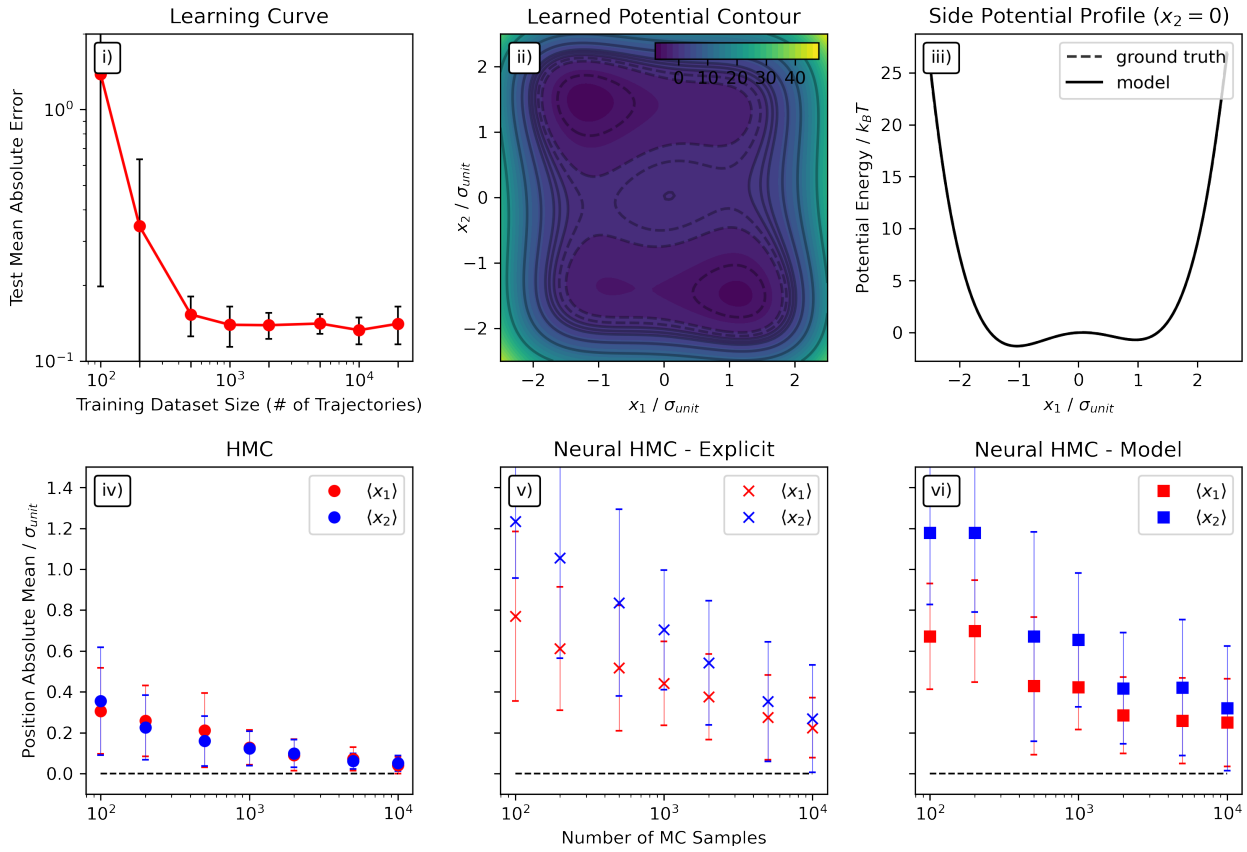
Figure 10 summarizes the results for all three potentials. In general, we can see increasing the number of training trajectories decreases the test error, but at some point there is diminishing returns and a larger dataset brings only marginal improvements. Different potentials achieve such saturation at different dataset sizes, depending on the potential's complexity and trajectory distribution in the training dataset. Note the high uncertainty when learning the 2D Wolfe-Quapp for small training datasets (Fig. 10c-i). This



(a) 2D Shell. Unit distance $\sigma_{\text{unit}} = 0.5r_0^2$, where r_0 is the radius of the shell.



(b) 10D Gaussian. Unit distance $\sigma_{\text{unit}} = \alpha_i^{\text{Gaussian}}$, where $\alpha_i^{\text{Gaussian}}$ is the coefficient in the Gaussian (Eq. 25).



(c) 2D Wolfe-Quapp. Unit distance $\sigma_{\text{unit}} = \alpha_0$, where α_0 is the unit factor for the quartic part of x_1 and x_2 (Eq. 26).

Figure 10: Results of learning three toy ML potentials using neural ODEs. All uncertainties are given by the standard deviation. All simulations use reduced units of energy, mass and time respectively as follows: $\epsilon_{\text{unit}} = k_B T$, $m_{\text{unit}} = m_0$, where m_0 is the particle's mass, and $\tau^2_{\text{unit}} = (\sigma^2_{\text{unit}} \cdot m_{\text{unit}})/\epsilon_{\text{unit}}$. For each potential, we give: i) Relationship between increasing the number of training trajectories and the decreasing test mean absolute error (defined by Eq. 29). ii) Heat map of learned ML potential, where the black lines show the contours of the true at logarithmic intervals (dashed represents negative energies). iii) Side profile of the ML potential where we fix $x_2 = 0$ from ii) compared to the true potential. Both ii) and iii) plots have the ML potential shifted so they align with the analytical potential, which does not affect gradients derived. iv) Sampled means from HMC simulations with regular Verlet evolution of the proposal step. v) Sampled means from neural HMC, where the proposal step is integrated through neural ODEs, but the MC step uses the exact definition of the potential. vi) Same as v) but the MC step uses the ML potential to determine the acceptance probability. The dashed black line shows the target mean at the origin for all potentials.

is presumably due to the complex topology, as well as training instability observed for the potential and was more common with lower dataset sizes. The loss would fail to decrease monotonically; we give some examples of the loss evolution in Appendix A.6.

The contour plots for the 2D Shell (Fig. 10a-ii) and 10D Gaussian (Fig. 10b-ii) show the inability for NNs to extrapolate outside of the position distributions from the training datasets. The ML potentials fail to capture the rounded nature of these for $|x_1| > 4$ and $|x_1| > 5$ respectively. In the side profile for the 2D Wolfe-Quapp (Fig. 10a-iii) the interpolation of the sharp turn at the origin also suffers.

As expected, regular HMC outperforms both versions of neural HMC in the 2D Shell (Figs. 10a-iv to 10a-vi) and 2D Wolfe-Quapp (Figs. 10c-iv to 10c-vi). Suprisingly though, neural HMC performs reasonably well for the 10D Gaussian and outperforms regular HMC in terms of a lower mean with a lower uncertainty (Figs. 10b-iv to 10b-vi). The regular HMC for 10D Gaussian hence performs relatively poorly with a high uncertainty, suggesting a high-dimensional potential might be harder to sample with regular HMC than a low-dimensional.

The two neural HMC approaches then perform similarly in the 10D Gaussian (Figs. 10b-v and 10b-vi) and the 2D Wolfe-Quapp (Figs. 10c-v and 10c-vi). For the 2D Shell (Figs. 10a-v and 10a-vi), the neural HMC that uses the ML potential for the MC step has a very high uncertainty and a higher mean, suggesting that the inaccurate prediction at the origin is corrected for by using the explicit potential.

5 Learning Diatomic Molecule

5.1 Method

Pair-Wise Interactions

To simplify computations, many MD simulations assume no multi-body interactions and only pair-wise interactions are considered. Here we simulate a triatomic molecule connected by harmonic springs by learning the pair-wise potential from a trajectory of a diatomic molecule simulation (Figure 11). In other words, we learn the quadratic potential between two atoms and then use that learned model to evolve three atoms, where we assume that only the neighbouring atoms interact.

As before, the position of each atom i given by x_i evolves by its momentum p_i as in Figure 8. Here we omit the bold notation as we only consider a one-dimensional molecule. The harmonic potential is

$$V(x_i, x_j) = \frac{1}{2}k(|x_i - x_j| - r_0)^2 \tag{30}$$

where k is the harmonic bond constant (set to 1), r_0 is the equilibrium distance (set to 1). In the diatomic molecule setup shown in Figure 11a, one can reduce the representation into a single reduced mass oscillating around the origin, i.e. centre of the spring. The potential and force then become

$$V(x^*) = -\frac{1}{2}kx^{*2} \tag{31}$$

$$F(x^*) = -\frac{\partial V(x^*)}{\partial x^*} = -k(x^* - r_0) \tag{32}$$

where x^* is the displacement from the origin. This reduced particle has a reduced mass $m^* = \frac{m^2}{2m}$ when the two atoms have identical mass m. Our NN will thus learn this $V(x^*)$ parametrized by the position of the reduced particle x^*. When we employ $V(x^*)$ in the triatomic molecule, we have to make sure that the reduced coordinates of the system are reflected in the direction of the restoring force as shown in Figure 11b. Therefore, combining that knowledge with Eq. 15 and the neighbour interaction assumption we evolve p_i as

$$\frac{dp_1}{dt} = -\frac{\partial V(x_1, x_2)}{\partial x_1} - \frac{\partial V(x_1, x_3)}{\partial x_1} = \frac{\partial V(x_A^*)}{\partial x_A^*} \tag{33}$$

$$\frac{dp_2}{dt} = -\frac{\partial V(x_2, x_1)}{\partial x_2} - \frac{\partial V(x_2, x_3)}{\partial x_2} = -\frac{\partial V(x_A^*)}{\partial x_A^*} + \frac{\partial V(x_B^*)}{\partial x_B^*} \tag{34}$$

$$\frac{dp_3}{dt} = -\frac{\partial V(x_3, x_1)}{\partial x_3} - \frac{\partial V(x_3, x_2)}{\partial x_3} = -\frac{\partial V(x_B^*)}{\partial x_B^*} \tag{35}$$

where x_A^* and x_B^* are the reduced particle's displacements from equilibrium for the two springs respectively.

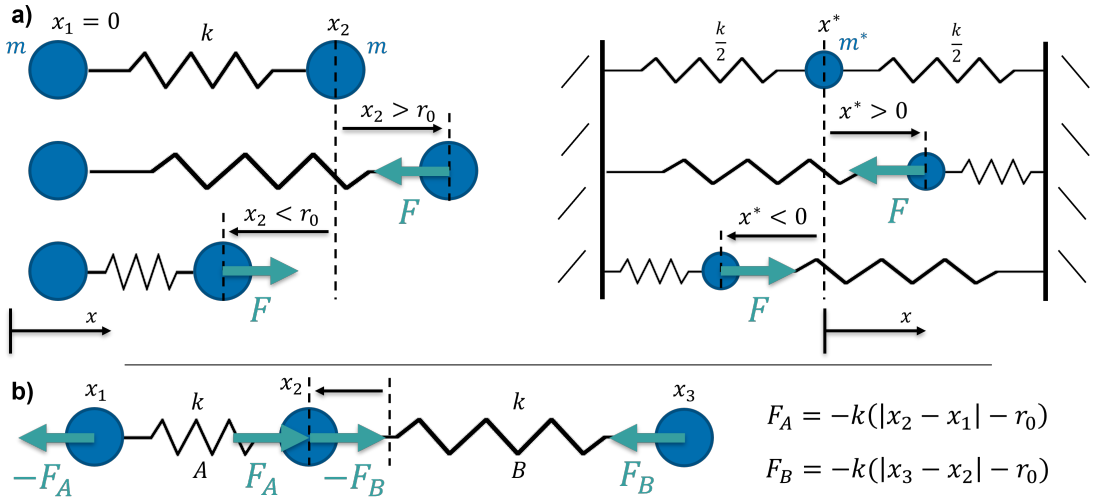


Figure 11: Schematic of the simple harmonic oscillator experiment. a) Equivalent representation of a diatomic molecule with a harmonic spring between two atoms and a reduced particle between two springs. Left representation has atom x_1 fixed at the origin, while x_2 is free to oscillate. Right representation has two springs of half the spring constant connected to a fixed wall with a reduced mass at displacement x^* from the origin. b) Extension of the diatomic spring to a triatomic molecule, where one needs to be careful about the directionality of the acting forces. All atoms are free to move and neighbouring atoms exert forces of same magnitude but opposite sign. For instance, the compressed spring A will exert a positive force F_A as its x_A^* will be smaller than zero, hence the right atom experiences force F_A, but the left atom has to experience $-F_A$ to move leftward.

Experiment

Similarly to the toy potentials, the dataset of positions and momenta $\{\mathbf{X}, \mathbf{P}\}$ of a reduced particle describing a diatomic molecule with a spring was generated by performing HMC simulations for 20 000 samples. The proposal step dynamics ran for 100 time steps of step size 0.1. Afterwards, 5 independent ML potentials were trained for 5 000 epochs using the `Adam` optimizer. The initial learning rate was 0.02 and was scheduled with `LambdaLR` by 0.9 every 1 000 epochs. The nets have a depth of 1 and a width of 50 neurons. The batch size was 800 trajectories of 40 time steps each. In the triatomic molecule, the positions of neither atom is fixed and are free to move in respect to the frame of reference of the system.

5.2 Results

Figure 12a to 12c show the gradual improvement in modelling the harmonic potential and its force. As with the toy potentials, we can see how well the NN can interpolate but fails to extrapolate beyond the trajectories of the training dataset. For reference, see Figures 12e and 12f that give the positions and velocities distribution in the dataset. As the largest bond extension is at $|x^*| = 2.5$, we see the NN failing to approximate for $|x^*| > 2.5$. For completeness, Figure 12d shows comparison between ReLU and Sigmoid activation functions, where ReLU's discontinuity fails to approximate the gradient for larger extensions x^*.

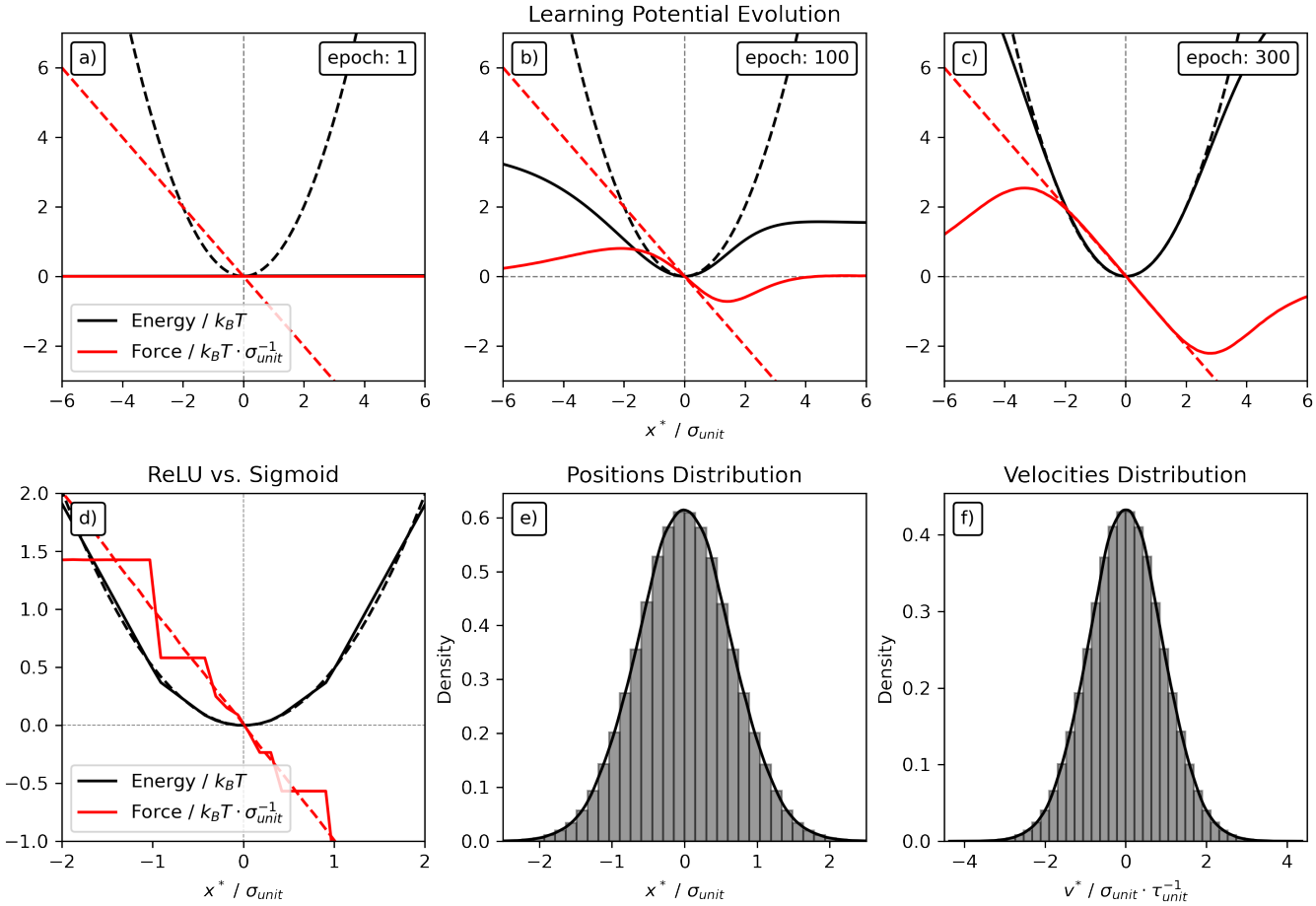


Figure 12: Unit distance $\sigma_{\text{unit}} = r_0$, where r_0 is the equilibrium distance between the atoms. a), b), c) Evolution of the learned potential energy and its first derivative at various points throughout the training. The dashed lines show the analytical solution given in Eqs. 31 and 32. d) Comparison of ML potentials with ReLU (solid) and Sigmoid (dashed) activation functions. e), f) Distributions of positions and velocities respectively in the training dataset for the diatomic molecule's reduced particle.

Figure 13 shows the divergence in position is similar for the diatomic molecule as for the triatomic molecule. Nevertheless, the velocity of the middle particle v_2 is more divergent than the velocity of the reduced particle v^* in the diatomic molecule. This is because of the error accumulation as the middle atom experiences two forces, one from each side atom, whilst the side atoms always experience only one force.

Figure 14 shows the ML potential can be extended to the triatomic molecule when assuming pair-wise interactions only. Special solutions in Figures 14a and 14b are captured, as well as a random initial condition in Figures 14c to 14f. The system's energy oscillates in Figure 14e but does not explode.

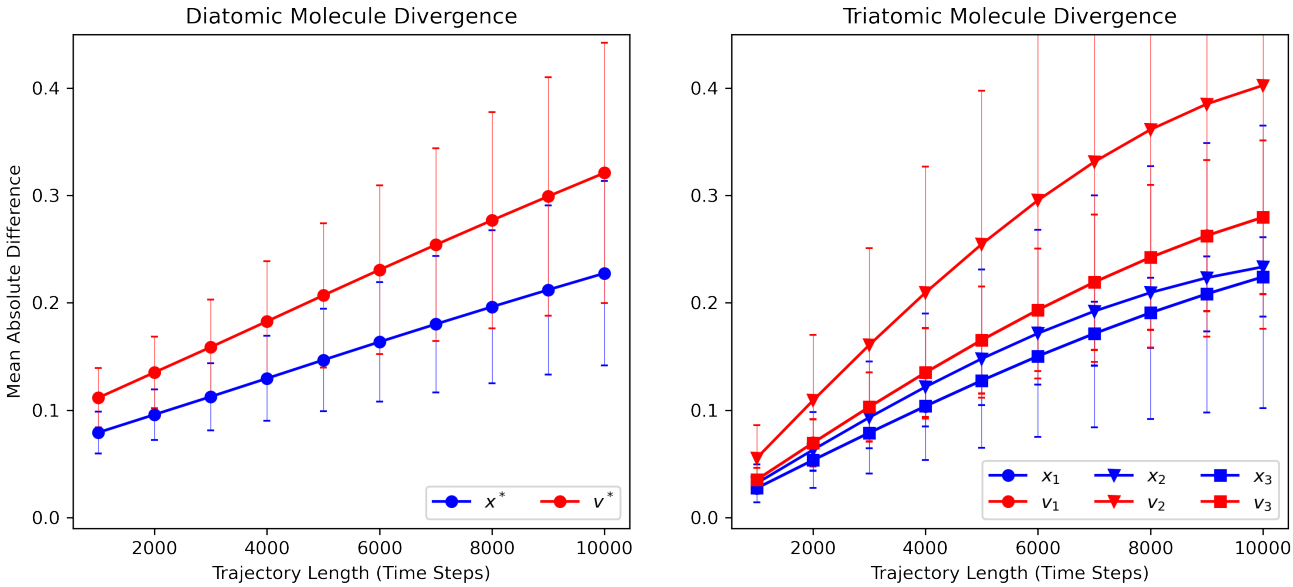


Figure 13: All uncertainties are given by the standard deviation of 5 independent ML potenatials. Mean absolute difference is equivalent to the loss given in Eq. 29. Left: Divergence of the position and momenta of the reduced particle within the diatomic molecule between the integrated ML potential and analytical solution given by $x(t) = \frac{v_0}{\omega_0}\sin(\omega_0 t) + x_0\cos(\omega_0 t)$ and $v(t) = v_0\cos(\omega_0 t) - \omega_0 x_0 \sin(\omega_0 t)$, where $\omega_0^2 = k/m$. Right: Divergence of the position and momenta for the atoms inside the triatomic molecule between the integrated ML potential and a numerical solution (given by integration with the exact potential). Data points for x_1 and v_1 follow that of x_3 and v_3, hence are hidden behind.

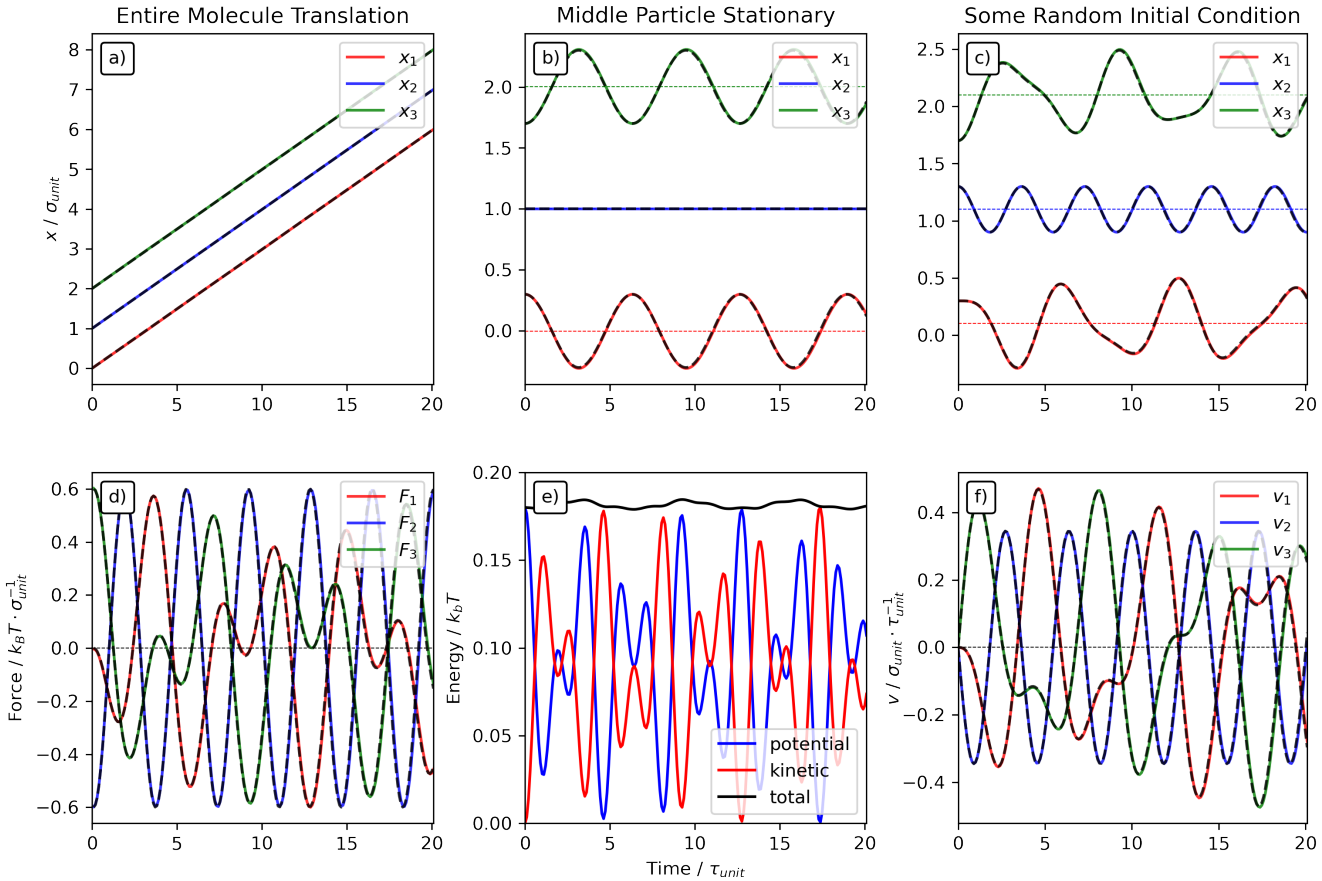


Figure 14: Unit distance $\sigma_{\text{unit}} = r_0$, where r_0 is the equilibrium distance between the atoms. Unit time $\tau_{\text{unit}} = \omega_0^{-1}$, where ω_0 is the natural oscillation frequency from Fig. 13. Triatomic molecule trajectories integrated using the ML potential. Black dashed lines show the numerical solution integrated using the exact potential. a) No oscillation solution (initial conditions: $x_1 = 0.0$, $x_2 = 1.0$, $x_3 = 2.0$, all $v_i = 0.3$). b) Oscillation solution with middle particle stationary (initial conditions: $x_1 = 0.3$, $x_2 = 1.0$, $x_3 = 1.7$, all $v_i = 0$). c), d), e), f) Example of a random solution (initial conditions: $x_1 = 0.3$, $x_2 = 1.3$, $x_3 = 1.7$, all $v_i = 0$) showing c) positions of atoms, d) forces on atoms, e) system's energy and f) velocities of atoms.

6 Learning Coarse-Grained Composite Body Potentials

6.1 Method

Rigid Body Kinetics

For a rigid body i, the minimum number of degrees of freedom to fully describe its configuration is six: $\mathbf{x}_i^{\text{com}} = (x_1^{\text{com}}, x_2^{\text{com}}, x_3^{\text{com}})$ for the position of the centre of mass (COM) and $\Phi_i = (\phi, \psi, \xi)$ for the Euler angles of the body's orientation. A system of two bodies can hence be characterised by 9 degrees of freedoms when one sets one of the body's centre of mass as the origin. Such bodies hence interact by some effective potential $V_{CG}(\mathbf{X}^{\text{com}}, \mathbf{Q})$, where $\mathbf{X}^{\text{com}}$ and $\mathbf{Q}$ are the COM positions and quaternion body rotations respectively. Note that instead of using Euler angles Φ we are introducing quaternions $\mathbf{Q}$ as they are not prone to geometrical singularities and are computationally more efficient (see Appendix A.5 for more).

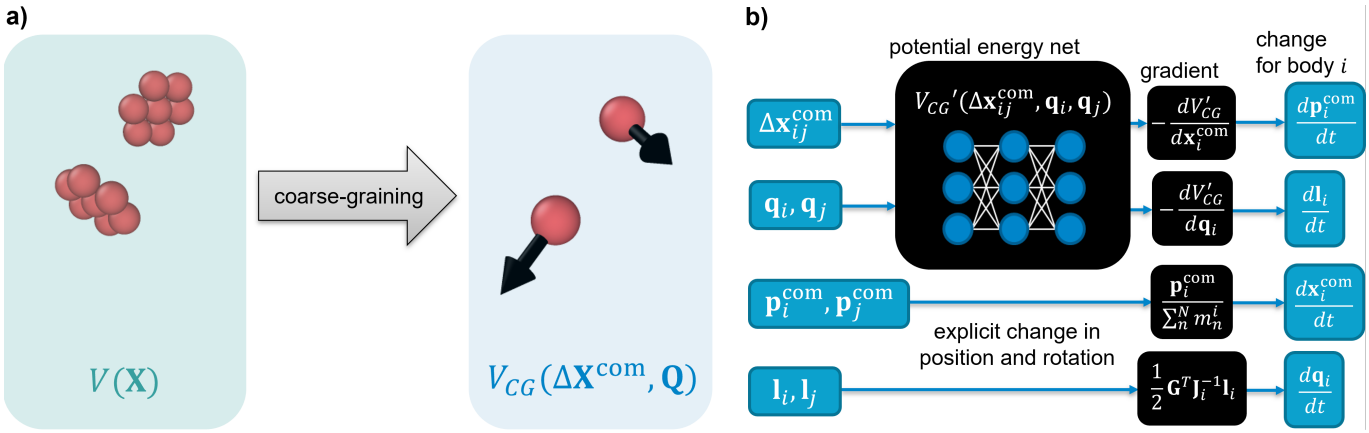


Figure 15: a) Composite bodies made of point particles that interact with pair potential $V(\mathbf{X})$ can be represented as centre of masses with an orientation interacting with a coarse-grained potential $V_{CG}(\Delta\mathbf{X}^{com}, \mathbf{Q})$. b) Schematic diagram of state variable evolution of the coarse-grained pair potential model $V'_{CG}(\Delta\mathbf{X}^{\text{com}}, \mathbf{Q})$ approximating $V_{CG}(\Delta\mathbf{X}^{\text{com}}, \mathbf{Q})$. This is essentially an extension of Fig. 8 with rotational dynamics now included as well.

Re-parameterizing the all-particle system Hamiltonian $H(\mathbf{X}, \mathbf{P})$ (Eq. 11), where $\{\mathbf{X}, \mathbf{P}\}$ are the positions and momenta of all point particles making up the rigid body, gives the new Hamiltonian

$$H(\mathbf{X}^{\text{com}}, \mathbf{P}^{\text{com}}, \mathbf{Q}, \mathbf{L}) = K_{\text{trans}}(\mathbf{P}^{\text{com}}) + K_{\text{rot}}(\mathbf{L}) + V_{CG}(\mathbf{X}^{\text{com}}, \mathbf{Q}) \tag{36}$$

where $\mathbf{P}^{\text{com}}$ describe the COM momentum, $\mathbf{L}$ are the 3-component angular momenta in body-fixed coordinates, and K_{trans} and K_{rot} are the translational and kinetic energies respectively. Hence, using the derivation from [80] and Eqs. 11 to 15, we show the differential equations for moving and rotating a N-particle composite body i when interacting with another body j only:

$$\frac{d\mathbf{x}_i^{\text{com}}}{dt} = \frac{\mathbf{p}_i^{\text{com}}}{\sum_n^N m_n^i} = \mathbf{v}_i^{\text{com}} \tag{37}$$

$$\frac{d\mathbf{p}_i^{\text{com}}}{dt} = -\frac{\partial V_{\text{CG}}(\mathbf{x}_i^{\text{com}}, \mathbf{x}_j^{\text{com}}, \mathbf{q}_i, \mathbf{q}_j)}{\partial \mathbf{x}_i^{\text{com}}} = \mathbf{F}_{ij} \tag{39}$$

$$\frac{d\mathbf{q}_i}{dt} = \frac{1}{2}\mathbf{G}^\top \mathbf{J}_i^{-1}\mathbf{l}_i \tag{38}$$

$$\frac{d\mathbf{l}_i}{dt} = -\boldsymbol{\Omega}\mathbf{l}_i - \frac{1}{2}\mathbf{G}\frac{\partial V_{CG}(\mathbf{x}_i^{\text{com}}, \mathbf{x}_j^{\text{com}}, \mathbf{q}_i, \mathbf{q}_j)}{\partial \mathbf{q}_i} \tag{40}$$

where $\mathbf{l}_i$ is the angular momenta in body-fixed coordinates and $\mathbf{J}_i$ is the diagonalized matrix of moments of inertia. The matrix $\mathbf{G}$ effectively performs quaternion multiplication and is defined by the 4 components of the quaternion $\mathbf{q} = [q_0, q_1, q_2, q_3]$ as

$$\mathbf{G} = \begin{bmatrix} -q_1 & q_0 & q_3 & -q_2 \\ -q_2 & -q_3 & q_0 & q_1 \\ -q_3 & q_2 & -q_1 & q_0 \end{bmatrix} \tag{41}$$

and $\boldsymbol{\Omega}$ is defined by its time derivative as

$$\boldsymbol{\Omega} = 2\mathbf{G}\dot{\mathbf{G}}^\top \tag{42}$$

Lastly, we can use the fact that the pair-wise interaction is rotationally invariant in terms of the relative COM positions, hence all we need to consider is the separation distance of the COMs giving us the modelled pair potential as $V'_{CG}(\Delta\mathbf{x}_{ij}^{\text{com}}, \mathbf{q_i}, \mathbf{q_j})$, where $\Delta\mathbf{x}_{ij}^{\text{com}}$ is the relative distance between the COMs.

Rotational Velocity-Verlet Algorithm

There are many variations of the Velocity-Verlet (leapfrog) algorithm to integrate rotational equations of motions [81] [82] [83]. Here, we employ the integrator from `LAMMPS` which uses Richardson iteration [84] as a correction step for the quaternion evolution. That should help in minimising deviations of our neural ODE solution from the training trajectories generated by `LAMMPS`. Note that the algorithm is given in body-fixed angular momenta $\mathbf{l}_i$, but one still has to be computing system-fixed $\tilde{\mathbf{l}}_\mathbf{i}$ for the Richardson correction. Also, each update to a quaternion is followed by quaternion normalization (meaning each Eq. 46 to 49 is followed by $\mathbf{q}_i \rightarrow \frac{\mathbf{q}_i}{||\mathbf{q}_i||}$).

1. translational and angular momentum at half-step ('half-kick', $\mathbf{x}_i$ and $\mathbf{q}_i$ constant)

$$\mathbf{p}_i^{\text{com}}\left(t+\frac{\Delta t}{2}\right) = \mathbf{p}_i^{\text{com}}(t) + \frac{\Delta t}{2}\frac{d\mathbf{p}_i^{\text{com}}}{dt}\Big|_t \quad (43) \qquad \mathbf{l}_i\left(t+\frac{\Delta t}{2}\right) = \mathbf{l}_i(t) + \frac{\Delta t}{2}\frac{d\mathbf{l}_i}{dt}\Big|_t \quad (44)$$

2. centre of mass position at full-step ('drift', free flight with $\mathbf{p}_i$ constant)

$$\mathbf{x}_i^{\text{com}}(t+\Delta t) = \mathbf{x}_i^{\text{com}}(t) + \frac{\Delta t}{\sum_n^N m_n^i}\mathbf{p}_i^{\text{com}}\left(t+\frac{\Delta t}{2}\right) \quad (45)$$

3. Richardson iteration for quaternion leapfrog ('drift', free rotation with $\mathbf{l}_i$ constant)
   (a) full-step Richardson update

$$\mathbf{q}_i^{\text{full}}(t+\Delta t) = \mathbf{q}_i(t) + \frac{\Delta t}{2}\,\mathbf{G}^\top\mathbf{J}^{-1}\,\mathbf{l}_i\left(t+\frac{\Delta t}{2}\right) \quad (46)$$

   (b) half-step Richardson update

$$\mathbf{q}_i^{\text{half}}\left(t+\frac{\Delta t}{2}\right) = \mathbf{q}_i(t) + \frac{1}{2}\frac{\Delta t}{2}\,\mathbf{G}^\top\mathbf{J}^{-1}\mathbf{l}_i\left(t+\frac{\Delta t}{2}\right) \quad (47)$$

   (c) re-compute $\mathbf{l}_i(t+\frac{\Delta t}{2})$ at $\mathbf{q}_i^{\text{half}}(t+\frac{\Delta t}{2})$ from $\tilde{\mathbf{l}}_i(t+\frac{\Delta t}{2})$ at $\mathbf{q}_i(t)$
   (d) second half-step Richardson update

$$\mathbf{q}_i^{\text{half}}(t+\Delta t) = \mathbf{q}_i^{\text{half}}\left(t+\frac{\Delta t}{2}\right) + \frac{1}{2}\frac{\Delta t}{2}\,\mathbf{G}^\top\mathbf{J}^{-1}\mathbf{l}_i\left(t+\frac{\Delta t}{2}\right) \quad (48)$$

   (e) corrected Richardson update

$$\mathbf{q}_i(t+\Delta t) = 2\mathbf{q}_i^{\text{half}}(t+\Delta t) - \mathbf{q}_i^{\text{full}}(t+\Delta t) \quad (49)$$

4. translational and angular momentum at full-step ('half-kick', $\mathbf{x}_i$ and $\mathbf{q}_i$ constant)

$$\mathbf{p}_i^{\text{com}}(t+\Delta t) = \mathbf{p}_i^{\text{com}}\left(t+\frac{\Delta t}{2}\right) + \frac{\Delta t}{2}\frac{d\mathbf{p}_i^{\text{com}}}{dt}\Big|_{t+\frac{\Delta t}{2}} \quad (50) \quad \mathbf{l}_i(t+\Delta t) = \mathbf{l}_i\left(t+\frac{\Delta t}{2}\right) + \frac{\Delta t}{2}\frac{d\mathbf{l}_i}{dt}\Big|_{t+\frac{\Delta t}{2}} \quad (51)$$

Experiment

For our composite body, we choose a simple 7-particle hexagon. All-atom trajectories were generated using `LAMMPS` and its `RIGID` package that fixes the relative position of point particles into a desired shape. All forces are transferred among the point particles whose positions remain constant relative to one another. The package also conveniently computes $\mathbf{L}$ and $\mathbf{Q}$. The bodies then interact through the pair-wise interactions between the point particles by the Lennard-Jones (LJ) potential defined as

$$V_{LJ}(\Delta x) = 4\epsilon\left[\left(\frac{\sigma^{LJ}}{\Delta x}\right)^{12} - \left(\frac{\sigma^{LJ}}{\Delta x}\right)^{6}\right] \qquad \Delta x < r_c \quad (52)$$

where Δx is the particle separation, ϵ^{LJ} is the depth of the potential well, σ^{LJ} is the distance at which the potential energy is zero (or equivalently the size of the particle) and r_c is the cutoff radius of the potential. To simplify the problem, we define a purely repulsive LJ potential shown in Figure 16a. ϵ^{LJ} and σ^{LJ} are both set to 1. To achieve random initial configuration, the system is initialized in an NVT canonical ensemble for 10^5 time steps, where the time step is $10^{-5}\,\tau_{\text{unit}}$. Then a short trajectory is generated in an NVE microcanonical ensemble for 10^7 time steps with a logging frequency of every 10^2 steps. `LAMMPS`

outputs COM positions, COM velocities as well as quaternion rotations and angular momenta. Our training dataset hence contains a single trajectory of 10^5 time steps, where the effective time step is $10^{-3}\,\tau_{\text{unit}}$.

As we want to primarily consider the phase space where the two bodies interact, we introduce a harmonic restraint keeping the two bodies within an interaction distance. This external potential is given by

$$V_{\text{ext}}(\Delta x^{\text{com}}) = \frac{1}{2} k_{\text{ext}} (\Delta x^{\text{com}} - \Delta x_0^{\text{com}})^2 \tag{53}$$

where the equilibrium distance Δx_0^{com} is set to $2l$ (where l is the hexagon's dimension shown in Figure 16b). k_{ext} is then defined so that there is $V_{\text{ext}} = 5k_BT$ when the bodies stop interacting and is given by

$$k_{\text{ext}} = \frac{10T^*}{r_c^2} \qquad\qquad T^* = \frac{k_BT}{\epsilon_{\text{unit}}} = \frac{T}{\epsilon_{\text{unit}}} \tag{54}$$

where T^* is the reduced temperature and k_B is set to 1 in `LAMMPS` by default. The NVE simulation is done at $T^* = 0.5$, hence $k \approx 3.96852\,\epsilon_{\text{unit}}/\sigma_{unit}^2$. `LAMMPS` was set to the `lj` reduced units system and the input file is given in Appendix A.7. During training we add analytically calculated V_{ext} to the effective potential of the system $V_{eff} = V_{ext} + V_{CG}$, hence `PyTorch` automatically learns the difference V_{CG} between V_{eff} and V_{ext}.

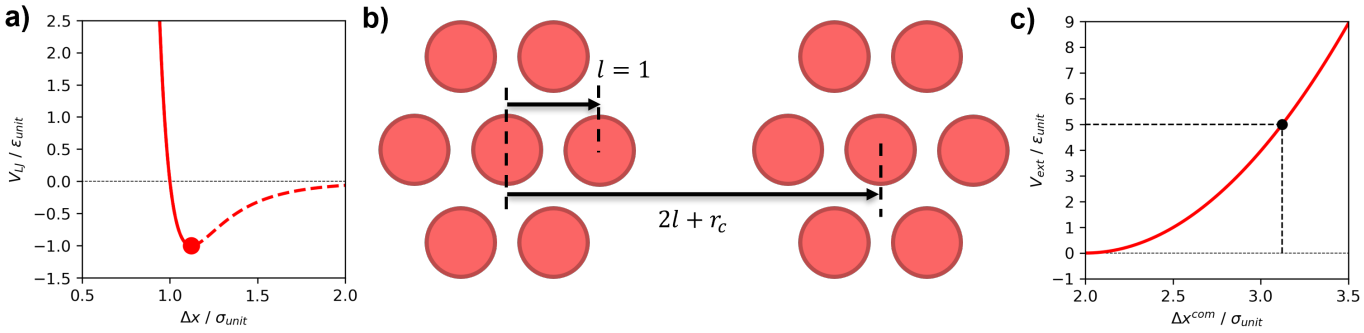


Figure 16: a) Lennard-Jones potential with cut-off at the minimum energy given by $r_c = \sqrt[6]{2}\sigma^{LJ} \approx 1.2246\,\sigma_{\text{unit}}$. b) Description of the regular hexagons and the distance where $V_{\text{ext}}(2l + r_c) = 5k_BT$. c) Harmonic restraint potential with the highlighted separation distance and energy from b).

We purposely choose a short trajectory to overfit our CG model so that we test out our implementation of neural ODEs with rotational dynamics. Therefore, we use a small 2-layer NN with 1000 neurons in both layers. To be more specific, we learn the potential $V'_{CG}(\Delta\mathbf{x}^{\text{com}}, \mathbf{q}_i, \mathbf{q}_j)$ as a function of 11 component variables. The input layer of 11 neurons receives the 3-component separation vector between COMs, and the 4-component quaternions for bodies i and j. As before, the output layer has a single linear neuron that gives us the potential energy, of which we take the gradients in respect to the inputs. The model was pushed to the limits to overfit, hence there were several training stages as shown in Table 2. The training and test loss are then the mean absolute difference of all state variables as given in Eq. 55.

Table 2: Training procedure parameters for overfitting a pair-wise CG potential on the 7-particle hexagon system using the `Adam` optimizer and `LambdaLR` scheduler in `PyTorch`, where the test loss is given by Eq. 55. For all iterations, each batch had 600 trajectories of 20 time steps each.

Iteration	Epochs	Initial Learning Rate	Scheduling Freq.	Scheduling Factor	Final Test Loss
1	30 000	0.035	1 000	0.85	0.0013018
2	50 000	0.020	5 000	0.85	0.0003558
3	50 000	0.010	10 000	0.85	0.0002310

$$L(\mathbf{X}, \hat{\mathbf{X}}, \mathbf{P}, \hat{\mathbf{P}}, \mathbf{Q}, \hat{\mathbf{Q}}, \mathbf{L}, \hat{\mathbf{L}}) = \frac{1}{13NT} \sum_{n=1}^{N} \sum_{t_0}^{t_f} \left(\frac{|\mathbf{X}_n^{\text{com}} - \hat{\mathbf{X}}_n^{\text{com}}|}{\sigma_{\text{unit}}} + \frac{|\mathbf{P}_n^{\text{com}} - \hat{\mathbf{P}}_n^{\text{com}}|}{\pi_{\text{unit}}} + |\mathbf{Q}_n - \hat{\mathbf{Q}}_n| + \frac{|\mathbf{L}_n - \hat{\mathbf{L}}_n|}{\lambda_{\text{unit}}} \right) \tag{55}$$

where we extend Eq. 29 with the rotational parts; the hat symbol refers to the predicted variable by integrating the model, λ_{unit} is the unit of angular momenta, quaternions are unitless and a factor of 13 comes from adding together the dimensions of the four variables, i.e. $3 + 3 + 4 + 3 = 13$.

6.2 Results

Figure 17 shows the overfitted trajectories and their divergences with longer integration times. The model primarily captures the interactions at shorter timescales while struggling with longer ones. It was also observed that the translational integration diverges more and oftentimes explodes, which is starting to happen to the velocities in Figure 17b. This is then supported by the divergence at the bottom in Figures 17g to 17i, where the translational component becomes dominant at longer trajectories. Nevertheless, this is to be expected as the contribution from quaternions is always going to be lower due to their normalization.

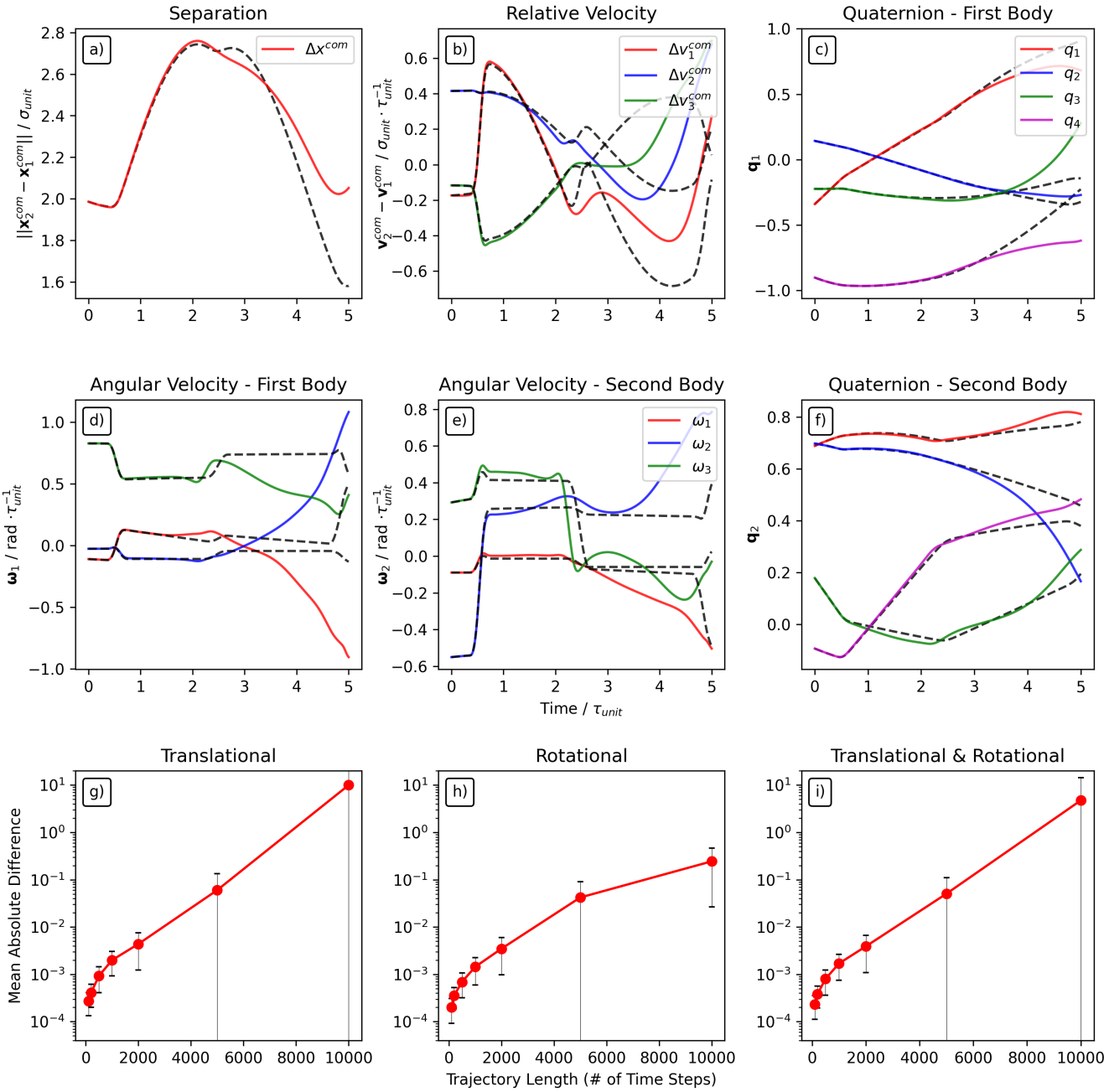


Figure 17: Results of overfitting $V'_{CG}(\Delta\mathbf{x}^{\text{com}}, \mathbf{q}_i, \mathbf{q}_j)$ on a short 10^5 time step trajectory of two 7-particle hexagons. The black dashed lines shows the training trajectory components from LAMMPS. a) Separation distance between centre of masses. b) Components of the relative velocity of the centre of masses. c), f) Components of the quaternions describing the bodies. d), e) Components of the angular velocities describing the rotational motion. g), h), i) Divergence of the translational, rotational and all state variables with increasing number of time steps (1 time step = $10^{-3}\,\tau_{\text{unit}}$). This is equivalent to Eq. 55, where for g) we only consider positions and momenta of the centres of mass; and for h) we only consider the angular momenta and quaternions. Uncertainties are given by the standard deviation of 100 independent trajectories.

7 Discussion

All of the results confirm the usability of neural ODEs as a training procedure towards learning ML potentials. The first task has shown that neural ODEs might be employed in the HMC sampling scheme. The second task then confirmed the feasibility of learning pair-wise ML potentials. Lastly, the final task has shown the ability to learn pair-wise CG potentials for rigid bodies. Below we discuss the overall importance of the results and suggest further exploration paths towards building knowledge of using neural ODEs to create automated coarse-graining pipelines for neural HMC simulations.

Training Datasets

The importance of generating relevant training trajectories is highlighted by the weak extrapolation abilities of NNs in the toy potentials and the diatomic potential. The dataset generation should thus be carefully studied and monitored to ensure training trajectories explore enough of the relevant phase space. For the toy potentials, our aim was to sample the observable mean, hence using HMC as the algorithm that produces important regions of the phase space was ideal. For the harmonic potential, we limited the maximum spring extension possible, hence we could have defined the relevant regions of the phase space by exploiting the physical limits of the system. By assuming the atoms cannot collide, we limited the spring extensions to a subset of the training position trajectories, meaning our ML potentials never had to extrapolate. This should be done with care as it might introduce a bias towards consequent microstate sampling, deviating the MD trajectory away from the ergodic ensemble.

For the CG potential, it is difficult to evaluate the effect of the harmonic restraint as we only modelled a very short trajectory, hence we learned on only a part of the phase space. Once we learn the phase space using trajectories at different temperatures, with a distribution of translational and angular momenta, we realise that the harmonic restraint might become insufficient and we might need to utilise a more robust method of exploring configurations. Moreover, composite bodies with more complex shapes will also require more powerful sampling methods such as HMC [35] or metadynamics [85]. Also, as NNs might extrapolate incorrectly, we stress the importance to sample enough of the phase space where the bodies do not interact at all, hence ensuring that our model learns those regions that have flat topology, i.e. no force.

Neural Net Architecture

In the first two tasks, the 2-layer net with 50 neurons in each layer exhibited the best performance learning the 10D Gaussian and the diatomic potential, demonstrating how NNs with the same size perform better at approximating simpler potentials compared to complicated ones. For the 2D Shell, the NN failed to interpolate the sharp turn with a discontinuous first derivative at the origin. We suggest to compare ReLU and Sigmoid activation functions to learn the 2D Shell and thus aid the net in capturing the region near the origin. We do this despite echoing the claims of [24] that non-differentiable activation functions, such as ReLU, should not be used for neural ODEs. Moreover, we suggest this despite ReLUs failing to model the force of the diatomic potential, because the modelled potential itself still resembled the harmonic potential reasonably well. Furthermore, for the 2D Wolfe-Quapp we believe that a deeper net might yield significantly better results in terms of the sampled mean in neural HMC compared to regular HMC. For reference, Bonati et al. [86] have used a 3-layer NN with 48, 24 and 12 neurons in successive layers respectively to learn the same Wolfe-Quapp potential.

In the last task, we have not explored enough of the phase space to evaluate the effect of the NN size on the results. More importantly, we have purposely chosen a small NN to ensure our implementation works correctly. As the net is able to approximate any function, if there was a mistake in any of the differential equations or data manipulations defined, a sufficiently large net might learn the required correction transformation, thus deceiving us into thinking our implementation is accurate. We have also not tried any other activation functions, although as we are approximating a differentiable potential and it is assumed that Tanh and Sigmoid activations perform similarly, naively using Sigmoid should be sufficient. Nevertheless, as the CG potential combines and mixes the interactions from translational and rotational contributions, it might be worthwhile investigating differently sized layers with various extents of connectivity. An example of such procedure might be *pruning* of neuron connections once a weight falls below a certain threshold during gradient descent. This provides memory and computational compression, as well as it can help the NN to emphasize certain pathways through the network over others.

Neural HMC

We also find that employing neural ODEs within HMC does not need to dramatically hinder the performance of sampling thermodynamic averages, and in the case of the 10D Gaussian it even outperforms the regular HMC. Although the difference is not statistically significant, we propose to reproduce this surprising result by sampling more regular HMC and neural HMC trajectories. One could also investigate the effect of increasing dimensionality of the Gaussian. That might uncover whether the higher dimensionality of the potential plays a role in making neural HMC competitive on performance.

Comparing neural HMC with the explicit and the model potentials shows that using the NN for the MC step does not need to significantly increase the error in the sampled average. If the exact expression of the potential is thus expensive to compute compared to the forward pass through the net, one might consider the trade-off between accuracy and speed. Referring to the CG potential, as the complexity of the rigid body increases, the modelled potential might become easier to compute than all of the individual point-particle pair interactions. Therefore, one could evaluate the CG potential rather than the original potential, possibly gaining computational speed at the cost of accuracy. A statistical algorithm could also be established that would alter between MC steps with the exact point-particle potential and the modelled NN potential, ensuring we stay in the correct ergodic sampling ensemble.

We acknowledge the limitations of benchmarking the neural ODE performance by comparing neural and regular HMC sampled means. It might be worth considering what is the effect of having the average at the origin of the energy landscape. Shifting the average away from the origin might show that the integrated trajectories are somehow exploiting translation around the origin. Such exploitation may then vary between the regular Velocity-Verlet and the neural ODE integrations, hence giving us a biased comparison.

Pair-Wise Potentials

The feasibility of extending pair-wise diatomic ML potentials learned with neural ODEs to triatomic molecules shows promising application towards scalibility. These results were to be expected as the NN is only involved during the potential approximation, hence if the function approximation is reasonably accurate, then all the established practices from MD should hold. Nevertheless, we still show it as a proof-of-concept that neural ODEs can be used within such context. We also note the accumulation of error that occurs for particles with the most pair interactions. Note that neural ODEs can also be easily generalised to learning multi-body potentials by increasing the number of neurons in the input layer.

Our current CG potential implementation could be further improved by considering just the separation distance between the centres of mass, rather than using the relative separation vector. Although the net should be able to learn the simple transformation from the vector to its norm, it might still provide marginal improvement.

Loss Functions

We do not learn much insight from testing different loss functions for the toy potentials. Even without further investigation, we however believe that using both positions and momenta seems like a reasonable and working approach. On the other hand, a more suitable loss function may exist for the CG potential as it also includes the more advanced ideas of rotational dynamics. That is because a loss defined as the mean absolute difference will always be smaller for quaternion and angular velocity evolution compared to translational separation and momenta. Although translational position and momenta of bodies might oscillate as we impose the harmonic restraint, the rotational variables are by definition going to oscillate, especially the quaternions. Therefore, one might consider at least using the mean squared error to penalise the translational variables from exploding. Another improvement might be to somehow scale or compute the loss of the translational and rotational parts separately.

The choice of the loss function is quite arbitrary in the end. As Greener et al. [65] discusses, any property that can be computed from the system with a meaningful gradient can be used as a guide to learn the relevant potential and hence its force field. Choosing the state variables of the spatial and rotational configuration is primarily a matter of convenience, as one has to integrate those variables either way to propagate the system forward in time. For specific applications, however, one might consider calculating statistical or thermodynamic properties at each time step such as the radial distribution function, the correlation of different particle velocities, the energy of a system, a measure of phase change, or a combination of the

above. Note, however, that this introduces an additional calculation to extract such a property from the training dataset and the predicted trajectories.

Speed and Optimization

We were not able to benchmark the memory cost advantages of neural ODEs over RNNs from [65]. As the intermediary values do not have to be computed, neural ODEs favour using large NNs integrated over many time steps. Nevertheless, as we have not optimized our implementation of neural ODEs, integrating for more time steps has dramatically increased training times. We suggest that the performance of longer integrations is compared to shorter integrations, seeing whether less long trajectories can optimize the model faster than many short trajectories.

In addition, our code has not been properly optimized. Cleaning up data manipulation, minimising variable assignment, or employing faster libraries for quaternion transforms (such as `pytorch3d` [87]) should hopefully shorten training times. For instance, our current implementation for the rotational Velocity-Verlet has to convert between system- and body-fixed coordinates, which should be addressed to minimise cost of converting the frame of reference within each integration step. Also, due to convenience and practicality of `Python` and `PyTorch`, we also lose much computational performance only due to the inefficiencies of `Python` compared to other programming languages. For production code, `Julia`'s `DifferentialEquations.jl` [88] is probably the best non-`Python` option. One could then also consider the various ways to automate the pipeline of generating MD trajectories from which to learn the CG potential. In an ideal scenario, one could design software where the user only provides the description of the simulated bodies. As a black-box, the program could then run MD simulations for long enough based on some heuristic so to approximate the CG potential sufficiently. The faster CG simulations would then take over and produce the final results to the user. One could also provide pre-trained models on common composite body types, or rigid body shapes.

For the toy and harmonic potentials, we have run for far more iterations than was required to fully train the NN. Hence only for the more complicated CG potential did the training times become the major bottleneck. We suggest utilising more GPUs in parallel using `PyTorch`'s `DataParallel` module which splits the batched inputs across multiple training replicas on multiple GPU devices during the forward pass, summing the gradients together into the original replica during the backward pass. This would allow for faster iterations of hyperparameter tuning as well as faster revisions of training procedures when we have to monitor the extent of phase space exploration in our trajectories.

Training a CG Potential

As shown in the method details in Section 6.1, training the CG potential was quite tedious as no clear set of hyperparameters was found to perform well over the entire training procedure. We hope to develop suitable good practices for training CG potentials using neural ODEs as there is currently no available literature on that topic.

Although neural ODEs should be able to find the dynamics that fit even sparse trajectories, we found it difficult to find hyperparameters that would learn a CG potential for trajectories with longer time steps. We also struggled to learn on trajectories with many frequent collisions between bodies. This suggests the choice of the time step as well as the temperature should be tuned carefully as to simplify the training procedure for the adjoint method as much as possible. For instance, one might start training on trajectories from low temperature simulations with slow dynamics, where collisions happen infrequently, learning the essentials of the ML potential first; and then gradually adding trajectories from higher temperatures which might be harder to train on from scratch otherwise.

As we now have some parameters of a CG model, we will attempt to further improve accuracy by feeding in new trajectories from different regions of the phase space. This procedure is referred to as *fine-tuning*, where a pre-trained model is utilised as the initial set of NN parameters to be further improved by learning on new data for a specific task of choice.

Advanced Implementations

Regardless of the shape of the composite body, the NN describing the CG potential might have learned some essential, baseline properties of potentials that mix the energy contributions from translational and rotational configurations. Therefore, one may employ trained CG models for other types of composite bodies than the ones trained on, hopefully shortening the amount of training epochs required as one is only fine-tuning the model, transfering some inherent knowledge about rigid body based CG potentials along the way.

Moreover, neural ODEs and the adjoint method only define the training procedure rather than the NN architecture. Therefore, it might be exciting to attempt using different architectures such as the recently popular graph NNs, which have shown success in defining interatomic potentials for MD simulations [47] [89]. One could hence try to apply graph neural ODEs [90] by combining the trajectory training principles of learning potentials as we have done here with the graph structure.

8 Conclusion

This thesis investigated the use of neural ODEs as a procedure to learn ML potentials from MD trajectories within the context of CG simulations. After introducing the theory of neural ODEs and HMC sampling, we performed two experiments where we learned valuable lessons for the final aim of developing an automated coarse-graining pipeline for composite body simulations. Due to time constraints of the thesis, the final task was cut short and only worked as a proof-of-concept. Still, the thesis included the first uses of neural ODEs in HMC sampling and coarse-graining of rigid bodies found in literature.

In the first experiment we learned ML potentials with neural ODEs on three examples: 2D Shell, 10D Gaussian and 2D Wolfe-Quapp. The accuracy varied for each, but all models performed competitively well when employed in proposing the next sample in an HMC simulation. We also showed one can use the learned potentials to perform the acceptance probability calculation in the MC step, potentially increasing simulation speeds.

The second experiment showed the feasibility of extending pair-wise potentials to multi-body interactions when learned through neural ODEs. The example used was a diatomic molecule connected by a harmonic spring, where we employed the learned potential to simulate a triatomic molecule. We are thus confident in the scalibility of the neural ODE approach for simulations of large systems with many particles.

The third experiment demonstrated the proof-of-concept application of learning a CG potential of composite bodies solely based on the position of their centre of mass position and their orientation. Our toy problem was made of two 7-particle hexagons interacting with the LJ potential. We successfully overfitted on a short trajectory, validating our neural ODE implementation that learns based on rotational kinetics. The generality of NNs hence proved useful when parametrizing such a complicated, high-dimensional potential with no clear analytical form.

Further work will consist of enhancing the CG model to learn the entire phase space of the two hexagon system. After optimizing the computational cost, the CG model will be benchmarked to an all-particle simulation. Afterwards, it will be useful to see the demands on the NN size and architecture when attempting to learn on bodies with complex surfaces and various interaction sites.

Neural ODEs seem like one of the plausible solutions to scaling computer simulations and should be explored further. They provide an alternative to the common approach of learning potentials on structure-energy or structure-force snapshots. Nevertheless, neural ODEs are still computationally demanding and thus may prove useful only in specific MD simulations. It will therefore be necessary to follow the advancements of both the AI research related to neural ODEs optimization, and the field of computational physics related to CG simulations, to fully determine their utility in the future.

9 Bibliography

[1] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65 6 (1958), pp. 386–408. DOI: 10.1037/h0042519.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386. URL: https://doi.org/10.1145/3065386.

[3] John Jumper et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596.7873 (July 2021), pp. 583–589. DOI: 10.1038/s41586-021-03819-2. URL: https://doi.org/10.1038/s41586-021-03819-2.

[4] James Kirkpatrick et al. "Pushing the frontiers of density functionals by solving the fractional electron problem". In: *Science* 374.6573 (Dec. 2021), pp. 1385–1389. DOI: 10.1126/science.abj6511. URL: https://doi.org/10.1126/science.abj6511.

[5] Eric H. Lee et al. "Discovery Through the Computational Microscope". In: *Structure* 17.10 (Oct. 2009), pp. 1295–1306. DOI: 10.1016/j.str.2009.09.001. URL: https://doi.org/10.1016/j.str.2009.09.001.

[6] Jay W. Ponder and David A. Case. "Force Fields for Protein Simulations". In: *Protein Simulations*. Vol. 66. Advances in Protein Chemistry. Academic Press, 2003, pp. 27–85. DOI: https://doi.org/10.1016/S0065-3233(03)66002-X. URL: https://www.sciencedirect.com/science/article/pii/S006532330366002X.

[7] Supriyo Naskar et al. *A minimal coarse-grained model to study the gelation of multi-armed DNA nanostars*. 2021. DOI: 10.48550/ARXIV.2110.11251. URL: https://arxiv.org/abs/2110.11251.

[8] J. Tersoff. "New empirical approach for the structure and energy of covalent systems". In: *Phys. Rev. B* 37 (12 Apr. 1988), pp. 6991–7000. DOI: 10.1103/PhysRevB.37.6991. URL: https://link.aps.org/doi/10.1103/PhysRevB.37.6991.

[9] Ryo Nagai, Ryosuke Akashi, and Osamu Sugino. "Completing density functional theory by machine learning hidden messages from molecules". In: *npj Computational Materials* 6.1 (May 2020). DOI: 10.1038/s41524-020-0310-0. URL: https://doi.org/10.1038/s41524-020-0310-0.

[10] Y. Mishin. "Machine-learning interatomic potentials for materials science". In: *Acta Materialia* 214 (Aug. 2021), p. 116980. DOI: 10.1016/j.actamat.2021.116980. URL: https://doi.org/10.1016/j.actamat.2021.116980.

[11] Charles Hansen et al. "The Fourth Paradigm: Data-Intensive Scientific Discovery". In: Jan. 2009, pp. 153–163.

[12] Ricky T. Q. Chen et al. *Neural Ordinary Differential Equations*. 2018. DOI: 10.48550/ARXIV.1806.07366. URL: https://arxiv.org/abs/1806.07366.

[13] L S Pontryagin. *Mathematical theory of optimal processes: Mathematical theory of optimal processes L.s. pontryagin selected works volume 4*. Classics of Soviet Mathematics. Amsterdam, Netherlands: Harwood Academic, Mar. 1987.

[14] Ronald M. Errico. "What Is an Adjoint Model?" In: *Bulletin of the American Meteorological Society* 78.11 (Nov. 1997), pp. 2577–2591. DOI: 10.1175/1520-0477(1997)078<2577:wiaam>2.0.co;2. URL: https://doi.org/10.1175/1520-0477(1997)078%3C2577:wiaam%3E2.0.co;2.

[15] Tianqi Chen et al. "Training Deep Nets with Sublinear Memory Cost". In: *CoRR* abs/1604.06174 (2016). arXiv: 1604.06174. URL: http://arxiv.org/abs/1604.06174.

[16] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791. URL: https://doi.org/10.1109/5.726791.

[17] Ricky T. Q. Chen. *torchdiffeq*. Version 0.2.2. June 2021. URL: https://github.com/rtqichen/torchdiffeq.

[18] Junteng Jia and Austin R. Benson. *Neural Jump Stochastic Differential Equations*. 2019. DOI: 10.48550/ARXIV.1905.10403. URL: https://arxiv.org/abs/1905.10403.

[19] Patrick Kidger et al. *Neural Controlled Differential Equations for Irregular Time Series*. 2020. DOI: 10.48550/ARXIV.2005.08926. URL: https://arxiv.org/abs/2005.08926.

[20] Xiang Xie, Ajith Kumar Parlikad, and Ramprakash Srinivasan Puri. "A Neural Ordinary Differential Equations Based Approach for Demand Forecasting within Power Grid Digital Twins". In: *2019 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. 2019, pp. 1–6. DOI: 10.1109/SmartGridComm.2019.8909789.

[21] Zhaozhi Qian et al. *Integrating Expert ODEs into Neural ODEs: Pharmacology and Disease Progression*. 2021. DOI: 10.48550/ARXIV.2106.02875. URL: https://arxiv.org/abs/2106.02875.

[22] Matías Núñez et al. "Forecasting virus outbreaks with social media data via neural ordinary differential equations". In: (Jan. 2021). DOI: 10.1101/2021.01.27.21250642. URL: https://doi.org/10.1101/2021.01.27.21250642.

[23] Alan Yang et al. *Input-to-State Stable Neural Ordinary Differential Equations with Applications to Transient Modeling of Circuits*. 2022. DOI: 10.48550/ARXIV.2202.06453. URL: https://arxiv.org/abs/2202.06453.

[24] Patrick Kidger. *On Neural Differential Equations*. 2022. DOI: 10.48550/ARXIV.2202.02435. URL: https://arxiv.org/abs/2202.02435.

[25] Yaofeng Desmond Zhong, Biswadip Dey, and Amit Chakraborty. "Symplectic ODE-Net: Learning Hamiltonian Dynamics with Control". In: (2019). DOI: 10.48550/ARXIV.1909.12077. URL: https://arxiv.org/abs/1909.12077.

[26] Greg Brockman et al. *OpenAI Gym*. 2016. DOI: 10.48550/ARXIV.1606.01540. URL: https://arxiv.org/abs/1606.01540.

[27] Kookjin Lee and Eric J. Parish. "Parameterized neural ordinary differential equations: applications to computational physics problems". In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 477.2253 (Sept. 2021), p. 20210162. DOI: 10.1098/rspa.2021.0162. URL: https://doi.org/10.1098/rspa.2021.0162.

[28] Xing Chen et al. "Forecasting the outcome of spintronic experiments with Neural Ordinary Differential Equations". In: *Nature Communications* 13.1 (Feb. 2022). DOI: 10.1038/s41467-022-28571-7. URL: https://doi.org/10.1038/s41467-022-28571-7.

[29] Tianhan Zhang et al. *A deep learning-based ODE solver for chemical kinetics*. 2020. DOI: 10.48550/ARXIV.2012.12654. URL: https://arxiv.org/abs/2012.12654.

[30] Wujie Wang, Simon Axelrod, and Rafael Gómez-Bombarelli. *Differentiable Molecular Simulations for Control and Learning*. 2020. DOI: 10.48550/ARXIV.2003.00868. URL: https://arxiv.org/abs/2003.00868.

[31] W. Kutta. *Beitrag zur näherungsweisen Integration totaler Differentialgleichungen*. Teubner, 1901. URL: https://books.google.co.uk/books?id=K5e6kQEACAAJ.

[32] Laurentiu Spiridon and David D. L. Minh. "Hamiltonian Monte Carlo with Constrained Molecular Dynamics as Gibbs Sampling". In: *Journal of Chemical Theory and Computation* 13.10 (Sept. 2017), pp. 4649–4659. DOI: 10.1021/acs.jctc.7b00570. URL: https://doi.org/10.1021/acs.jctc.7b00570.

[33] Simon Duane et al. "Hybrid Monte Carlo". In: *Physics Letters B* 195.2 (1987), pp. 216–222. ISSN: 0370-2693. DOI: https://doi.org/10.1016/0370-2693(87)91197-X. URL: https://www.sciencedirect.com/science/article/pii/037026938791197X.

[34] Marcel Hirt, Michalis K. Titsias, and Petros Dellaportas. *Entropy-based adaptive Hamiltonian Monte Carlo*. 2021. DOI: 10.48550/ARXIV.2110.14625. URL: https://arxiv.org/abs/2110.14625.

[35] Michael Betancourt. *A Conceptual Introduction to Hamiltonian Monte Carlo*. 2017. DOI: 10.48550/ARXIV.1701.02434. URL: https://arxiv.org/abs/1701.02434.

[36] Sergei Prokhorenko et al. "Large scale hybrid Monte Carlo simulations for structure and property prediction". In: *npj Computational Materials* 4.1 (Dec. 2018). DOI: 10.1038/s41524-018-0137-0. URL: https://doi.org/10.1038/s41524-018-0137-0.

[37] Benedikt Rennekamp et al. "Hybrid Kinetic Monte Carlo/Molecular Dynamics Simulations of Bond Scissions in Proteins". In: *Journal of Chemical Theory and Computation* 16.1 (Nov. 2019), pp. 553–563. DOI: 10.1021/acs.jctc.9b00786. URL: https://doi.org/10.1021/acs.jctc.9b00786.

[38] M.P. Allen et al. *Computer Simulation of Liquids*. Oxford Science Publ. Clarendon Press, 1989. ISBN: 978-0-198-55645-9. URL: https://books.google.co.uk/books?id=O32VXB9e5P4C.

[39] Joshua A. Vita and Dallas R. Trinkle. "Exploring the necessary complexity of interatomic potentials". In: *Computational Materials Science* 200 (2021), p. 110752. ISSN: 0927-0256. DOI: https://doi.org/10.1016/j.commatsci.2021.110752. URL: https://www.sciencedirect.com/science/article/pii/S0927025621004791.

[40] Jörg Behler et al. "Pressure-induced phase transitions in silicon studied by neural network-based metadynamics simulations". In: *physica status solidi (b)* 245.12 (Dec. 2008), pp. 2618–2629. DOI: 10.1002/pssb.200844219. URL: https://doi.org/10.1002/pssb.200844219.

[41] Chris M. Handley and Paul L. A. Popelier. "Potential Energy Surfaces Fitted by Artificial Neural Networks". In: *The Journal of Physical Chemistry A* 114.10 (Feb. 2010), pp. 3371–3383. DOI: 10.1021/jp9105585. URL: https://doi.org/10.1021/jp9105585.

[42] Hagai Eshet et al. "iAb initio/iquality neural-network potential for sodium". In: *Physical Review B* 81.18 (May 2010). DOI: 10.1103/physrevb.81.184107. URL: https://doi.org/10.1103/physrevb.81.184107.

[43] Jörg Behler and Michele Parrinello. "Generalized Neural-Network Representation of High-Dimensional Potential-Energy Surfaces". In: *Physical Review Letters* 98.14 (Apr. 2007). DOI: 10.1103/physrevlett.98.146401. URL: https://doi.org/10.1103/physrevlett.98.146401.

[44] Frank Noé et al. "Machine Learning for Molecular Simulation". In: *Annual Review of Physical Chemistry* 71.1 (Apr. 2020), pp. 361–390. DOI: 10.1146/annurev-physchem-042018-052331. URL: https://doi.org/10.1146/annurev-physchem-042018-052331.

[45] K. T. Schütt et al. "SchNet – A deep learning architecture for molecules and materials". In: *The Journal of Chemical Physics* 148.24 (June 2018), p. 241722. DOI: 10.1063/1.5019779. URL: https://doi.org/10.1063/1.5019779.

[46] Albert Musaelian et al. "Learning Local Equivariant Representations for Large-Scale Atomistic Dynamics". In: *arXiv e-prints*, arXiv:2204.05249 (Apr. 2022), arXiv:2204.05249. arXiv: 2204.05249 [physics.comp-ph].

[47] Simon Batzner et al. "E(3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials". In: *Nature Communications* 13.1 (May 2022). DOI: 10.1038/s41467-022-29939-5. URL: https://doi.org/10.1038/s41467-022-29939-5.

[48] Jörg Behler. "Perspective: Machine learning potentials for atomistic simulations". In: *The Journal of Chemical Physics* 145.17 (Nov. 2016), p. 170901. DOI: 10.1063/1.4966192. URL: https://doi.org/10.1063/1.4966192.

[49] Jiang Wang et al. "Machine Learning of Coarse-Grained Molecular Dynamics Force Fields". In: *ACS Central Science* 5.5 (Apr. 2019), pp. 755–767. DOI: 10.1021/acscentsci.8b00913. URL: https://doi.org/10.1021/acscentsci.8b00913.

[50] Stefan Doerr et al. "TorchMD: A Deep Learning Framework for Molecular Simulations". In: *Journal of Chemical Theory and Computation* 17.4 (Mar. 2021), pp. 2355–2363. DOI: 10.1021/acs.jctc.0c01343. URL: https://doi.org/10.1021/acs.jctc.0c01343.

[51] Samuel S. Schoenholz and Ekin D. Cubuk. "JAX, M.D.: A Framework for Differentiable Physics". In: (2019). DOI: 10.48550/ARXIV.1912.04232. URL: https://arxiv.org/abs/1912.04232.

[52] A. P. Thompson et al. "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales". In: *Comp. Phys. Comm.* 271 (2022), p. 108171. DOI: 10.1016/j.cpc.2021.108171.

[53] Ada Sedova et al. "High-Performance Molecular Dynamics Simulation for Biological and Materials Sciences: Challenges of Performance Portability". In: (Nov. 2018). DOI: 10.1109/P3HPC.2018.00004. URL: https://www.osti.gov/biblio/1493977.

[54] Aram Davtyan et al. "AWSEM-MD: Protein Structure Prediction Using Coarse-Grained Physical Potentials and Bioinformatically Based Local Structure Biasing". In: *The Journal of Physical Chemistry B* 116.29 (May 2012), pp. 8494–8503. DOI: 10.1021/jp212541y. URL: https://doi.org/10.1021/jp212541y.

[55] Luca Monticelli et al. "The MARTINI Coarse-Grained Force Field: Extension to Proteins". In: *Journal of Chemical Theory and Computation* 4.5 (Apr. 2008), pp. 819–834. DOI: 10.1021/ct700324x. URL: https://doi.org/10.1021/ct700324x.

[56] Sergei Izvekov and Gregory A. Voth. "A Multiscale Coarse-Graining Method for Biomolecular Systems". In: *The Journal of Physical Chemistry B* 109.7 (Jan. 2005), pp. 2469–2473. DOI: 10.1021/jp044629q. URL: https://doi.org/10.1021/jp044629q.

[57] James L. Suter, Derek Groen, and Peter V. Coveney. "Chemically Specific Multiscale Modeling of Clay-Polymer Nanocomposites Reveals Intercalation Dynamics, Tactoid Self-Assembly and Emergent Materials Properties". In: *Advanced Materials* 27.6 (Dec. 2014), pp. 966–984. DOI: 10.1002/adma.201403361. URL: https://doi.org/10.1002/adma.201403361.

[58] Aditi Khot, Stephen B. Shiring, and Brett M. Savoie. "Evidence of information limitations in coarse-grained models". In: *The Journal of Chemical Physics* 151.24 (Dec. 2019), p. 244105. DOI: 10.1063/1.5129398. URL: https://doi.org/10.1063/1.5129398.

[59] Thomas E. Ouldridge, Ard A. Louis, and Jonathan P. K. Doye. "Structural, mechanical, and thermodynamic properties of a coarse-grained DNA model". In: *The Journal of Chemical Physics* 134.8 (Feb. 2011), p. 085101. DOI: 10.1063/1.3552946. URL: https://doi.org/10.1063%5C%2F1.3552946.

[60] Wujie Wang and Rafael Gómez-Bombarelli. "Coarse-graining auto-encoders for molecular dynamics". In: *npj Computational Materials* 5.1 (Dec. 2019). DOI: 10.1038/s41524-019-0261-5. URL: https://doi.org/10.1038/s41524-019-0261-5.

[61] Christoph Wehmeyer and Frank Noé. "Time-lagged autoencoders: Deep learning of slow collective variables for molecular kinetics". In: *The Journal of Chemical Physics* 148.24 (June 2018), p. 241703. DOI: 10.1063/1.5011399. URL: https://doi.org/10.1063%5C%2F1.5011399.

[62] Andreas Mardt et al. "VAMPnets for deep learning of molecular kinetics". In: *Nature Communications* 9.1 (Jan. 2018). DOI: 10.1038/s41467-017-02388-1. URL: https://doi.org/10.1038%5C%2Fs41467-017-02388-1.

[63] Brooke E. Husic et al. "Coarse graining molecular dynamics with graph neural networks". In: *The Journal of Chemical Physics* 153.19 (Nov. 2020), p. 194101. DOI: 10.1063/5.0026133. URL: https://doi.org/10.1063/5.0026133.

[64] Jiang Wang et al. "Ensemble learning of coarse-grained molecular dynamics force fields with a kernel approach". In: *The Journal of Chemical Physics* 152.19 (May 2020), p. 194106. DOI: 10.1063/5.0007276. URL: https://doi.org/10.1063/5.0007276.

[65] Joe G. Greener and David T. Jones. "Differentiable molecular simulation can learn all the parameters in a coarse-grained force field for proteins". In: *PLOS ONE* 16.9 (Sept. 2021). Ed. by Yang Zhang, e0256990. DOI: 10.1371/journal.pone.0256990. URL: https://doi.org/10.1371%5C%2Fjournal.pone.0256990.

[66] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[67] Peter I. Frazier. *A Tutorial on Bayesian Optimization*. 2018. DOI: 10.48550/ARXIV.1807.02811. URL: https://arxiv.org/abs/1807.02811.

[68] Alden H. Wright. "Genetic Algorithms for Real Parameter Optimization". In: *Foundations of Genetic Algorithms*. Elsevier, 1991, pp. 205–218. DOI: 10.1016/b978-0-08-050684-5.50016-1. URL: https://doi.org/10.1016/b978-0-08-050684-5.50016-1.

[69] Scott Clark and Patrick Hayes. *SigOpt Web page*. https://sigopt.com. 2019. URL: https://sigopt.com.

[70] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *AISTATS*. 2010.

[71] Daniel Karlsson and Olle Svanström. "Modelling Dynamical Systems Using Neural Ordinary Differential Equations". MA thesis. Chalmers University of Technology Department of Physics, 2019. URL: https://hdl.handle.net/20.500.12380/256887.

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

[73] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018. URL: http://github.com/google/jax.

[74] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[75] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[76] Yiping Lu et al. *Beyond Finite Layer Neural Networks: Bridging Deep Architectures and Numerical Differential Equations*. 2017. DOI: 10.48550/ARXIV.1710.10121. URL: https://arxiv.org/abs/1710.10121.

[77] Daan Frenkel and Berend Smit. "Chapter 2 - Statistical Mechanics". In: *Understanding Molecular Simulation (Second Edition)*. Ed. by Daan Frenkel and Berend Smit. Second Edition. San Diego: Academic Press, 2002, pp. 9–22. ISBN: 978-0-12-267351-1. DOI: https://doi.org/10.1016/B978-012267351-1/50004-3. URL: https://www.sciencedirect.com/science/article/pii/B9780122673511500043.

[78] Will Grathwohl et al. "Scalable Reversible Generative Models with Free-form Continuous Dynamics". In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=rJxgknCcK7.

[79] Michele Invernizzi and Michele Parrinello. "Making the best of a bad situation: A multiscale approach to free energy calculation". In: *Journal of Chemical Theory and Computation* 15.4 (2019), pp. 2187–2194. URL: https://doi.org/10.1021/acs.jctc.9b00032.

[80] Ravishankar Shivarama and Eric P. Fahrenthold. "Hamilton's Equations With Euler Parameters for Rigid Body Dynamics Modeling". In: *Journal of Dynamic Systems, Measurement, and Control* 126.1 (Mar. 2004), pp. 124–130. DOI: 10.1115/1.1649977. URL: https://doi.org/10.1115%5C%2F1.1649977.

[81] David Fincham. "Leapfrog Rotational Algorithms". In: *Molecular Simulation* 8.3-5 (Jan. 1992), pp. 165–178. DOI: 10.1080/08927029208022474. URL: https://doi.org/10.1080/08927029208022474.

[82] Igor P. Omelyan. "A New Leapfrog Integrator of Rotational Motion. The Revised Angular-Momentum Approach". In: *Molecular Simulation* 22.3 (May 1999), pp. 213–236. DOI: 10.1080/08927029908022097. URL: https://doi.org/10.1080/08927029908022097.

[83] Miyabi Hiyama, Tomoyuki Kinjo, and Shi-aki Hyodo. "Angular Momentum Form of Verlet Algorithm for Rigid Molecules". In: *Journal of the Physical Society of Japan* 77.6 (June 2008), p. 064001. DOI: 10.1143/jpsj.77.064001. URL: https://doi.org/10.1143/jpsj.77.064001.

[84] Bernd Fischer and Lothar Reichel. "A stable Richardson iteration method for complex linear systems". In: *Numerische Mathematik* 54.2 (Mar. 1989), pp. 225–242. DOI: 10.1007/bf01396976. URL: https://doi.org/10.1007/bf01396976.

[85] Giovanni Bussi and Alessandro Laio. "Using metadynamics to explore complex free-energy landscapes". In: *Nature Reviews Physics* 2.4 (Mar. 2020), pp. 200–212. DOI: 10.1038/s42254-020-0153-0. URL: https://doi.org/10.1038/s42254-020-0153-0.

[86] Luigi Bonati, Yue-Yu Zhang, and Michele Parrinello. "Neural networks-based variationally enhanced sampling". In: *Proceedings of the National Academy of Sciences* 116.36 (Aug. 2019), pp. 17641–17647. DOI: 10.1073/pnas.1907975116. URL: https://doi.org/10.1073/pnas.1907975116.

[87] Nikhila Ravi et al. "Accelerating 3D Deep Learning with PyTorch3D". In: *arXiv:2007.08501* (2020). URL: https://doi.org/10.48550/arXiv.2007.08501.

[88] Christopher Rackauckas and Qing Nie. "DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia". In: *The Journal of Open Research Software* 5.1 (2017). Exported from https://app.dimensions.ai on 2019/05/05. DOI: 10.5334/jors.151. URL: https://app.dimensions.ai/details/publication/pub.1085583166%20and%20http://openresearchsoftware.metajnl.com/articles/10.5334/jors.151/galley/245/download/.

[89] Sina Stocker et al. "How Robust are Modern Graph Neural Network Potentials in Long and Hot Molecular Dynamics Simulations?" In: (Apr. 2022). DOI: 10.26434/chemrxiv-2022-mc4gb. URL: https://doi.org/10.26434/chemrxiv-2022-mc4gb.

[90] Michael Poli et al. *Graph Neural Ordinary Differential Equations*. 2021. arXiv: 1911.07532 [cs.LG].

A Appendix

A.1 Backpropagation algorithm

For instance, to update weight w_{33}^1 in Figure 18a, the gradient to find is

$$\frac{\partial L}{\partial w_{33}^1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \sigma_1^3} \frac{\partial \sigma_1^3}{\partial a_1^3} \frac{\partial a_1^3}{\partial w_{33}^1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \sigma_1^3} \frac{\partial \sigma_1^3}{\partial a_1^3} \cdot w_{31}^2 \frac{\partial \sigma_3^2}{\partial a_3^2} \sigma_3^1(a_3^1) \tag{56}$$

as the pre-activation values of the two relevant neurons are

$$\begin{aligned} a_1^3 &= \sum_i w_{i1}^2 \sigma_i^2 + b_1^3 = w_{11}^2 \sigma_1^2 + w_{21}^2 \sigma_2^2 + w_{31}^2 \sigma_3^2 + b_1^3 \\ a_3^2 &= \sum_i w_{i3}^1 \sigma_i^1 + b_3^2 = w_{13}^1 \sigma_1^1 + w_{23}^1 \sigma_2^1 + w_{33}^1 \sigma_3^1 + b_3^2 \end{aligned} \tag{57}$$

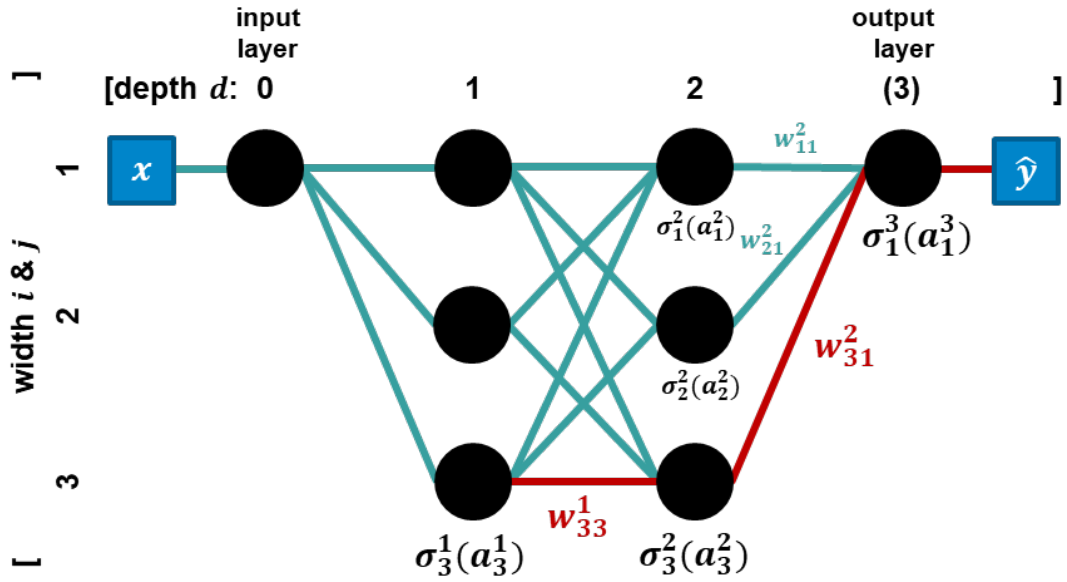


Figure 18: Small neural net of 2 hidden layers ($d = 1$ and $d = 2$) and 3 neurons in each hidden layer ($w = 1$, $w = 2$ and $w = 3$). Red shows the path of the backpropagation algorithm for calculating the gradient of the loss function with respect to w_{33}^1 solved in Eqs. 56 and 57

A.2 LambdaLR Scheduler

This scheduler sets the learning rate to the initial learning rate multiplied by a given function $\lambda(b) = \alpha^b$, where α is the scheduling factor and b is the number of scheduling steps already taken. That is defined by the integer division $b = n//\beta$, where n is the training epoch number and β is the scheduling frequency. Therefore, the learning rate is $\eta(n) = \lambda(n)\eta(0) = \alpha^b \eta(0)$.

A.3 Augmented Dynamics of the Adjoint Method

Below we reproduce the derivation of the adjoint state from the original neural ODE paper [12]. When one views θ and t as states with constant differential equations as

$$\frac{\partial \theta(t)}{\partial t} = \mathbf{0} \qquad\qquad \frac{\partial t(t)}{\partial t} = 1 \tag{58}$$

Combining these into an *augmented state* gives

$$\frac{d}{dt} \begin{bmatrix} \mathbf{Z} \\ \theta \\ t \end{bmatrix} (t) = f_{aug}([\mathbf{Z}, \theta, t]) := \begin{bmatrix} f([\mathbf{Z}, \theta, t]) \\ \mathbf{0} \\ 1 \end{bmatrix} \tag{59}$$

The adjoint state of the augmented dynamics then follows

$$\mathbf{A}_{aug} := \begin{bmatrix} \mathbf{A} \\ \mathbf{A}_\theta \\ \mathbf{A}_t \end{bmatrix} \qquad \mathbf{A}_\theta(t) := \frac{\partial L}{\partial \theta(t)} \qquad \mathbf{A}_t(t) := \frac{\partial L}{\partial t(t)} \tag{60}$$

This formulates the augmented ODE as an autonomous (time-invariant) ODE. The Jacobian of f then has the form

$$\frac{\partial f_{aug}}{\partial [\mathbf{Z}, \theta, t]} = \begin{bmatrix} \frac{\partial f}{\partial \mathbf{Z}} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} (t) \tag{61}$$

where $\mathbf{0}$ is a matrix of zeros with the appropriate dimensions. Plugging this into the differential equation of the adjoint

$$\frac{\partial \mathbf{A}(t)}{\partial t} = -\mathbf{A}(t) \frac{\partial f(\mathbf{Z}(t), \theta)}{\partial \mathbf{Z}(t)} \tag{62}$$

gives us

$$\frac{\partial \mathbf{A}_{aug}(t)}{\partial t} = -\begin{bmatrix} \mathbf{A}(t) & \mathbf{A}_\theta(t) & \mathbf{A}_t(t) \end{bmatrix} \frac{\partial f_{aug}}{\partial [\mathbf{Z}, \theta, t]}(t) = -\begin{bmatrix} \mathbf{A} \frac{\partial f}{\partial \mathbf{Z}} & \mathbf{A} \frac{\partial f}{\partial \theta} & \mathbf{A} \frac{\partial f}{\partial t} \end{bmatrix} (t) \tag{63}$$

The first element gives the regular adjoint differential equation in Eq. 62. The second element can be used to obtain the total gradient with respect to the parameters θ by integrating over the full interval and setting $\mathbf{A}(t_N) = \mathbf{0}$ as

$$\frac{\partial L}{\partial \theta} = \mathbf{A}_\theta(t_0) = -\int_{t_N}^{t_0} \mathbf{A}(t) \frac{\partial f(\mathbf{Z}(t), t, \theta)}{\partial \theta} dt \tag{64}$$

The last element then can also get the gradients with respect to t_0 and t_N as

$$\frac{\partial L}{\partial t_N} = \mathbf{A}(t_N) f(\mathbf{Z}(t_N), t_N, \theta) \qquad \frac{\partial L}{\partial t_0} = \mathbf{A}_t(t_0) = \mathbf{A}_t(t_N) - \int_{t_N}^{t_0} \mathbf{A}(t) \frac{\partial f(\mathbf{Z}(t), t, \theta)}{dt} \tag{65}$$

A.4 Continous Normalizing Flows and Hamiltonian Monte Carlo

Neural ODEs are not phase space preserving if there is a transformation of variables, meaning they do not conserve probability densities during integration. These models are referred to as continous normalizing flows and their probability density follows a second order differential equation called the instantaneous change of variables formula [12].

$$\frac{\partial \log p(\mathbf{Z}(t))}{dt} = -\operatorname{Tr}\left(\frac{df}{d\mathbf{Z}(t)}\right) \tag{66}$$

$$\log p(\mathbf{Z}(t_1)) = \log p(\mathbf{Z}(t_0)) - \int_{t_0}^{t_1} \operatorname{Tr}\left(\frac{df}{d\mathbf{Z}(t)}\right) dt \tag{67}$$

Similarly to the augmented dynamics in Eq. 59, we integrate the trace in Eq. 67 by concatenating a zero to the neural ODE input [78]:

$$\log p(\mathbf{Z}(t_1)) = \log p(\mathbf{Z}(t_0)) - \int_{t_0}^{t_1} \operatorname{Tr}\left(\frac{df}{d\mathbf{Z}}(t)\right) dt \tag{68}$$

The solutions to the dynamics are given as

$$\begin{bmatrix} \mathbf{Z}(t_0) \\ \log p(\mathbf{Z}(t_1)) - \log p(\mathbf{Z}(t_0)) \end{bmatrix} = \int_{t_1}^{t_0} \begin{bmatrix} f(\mathbf{Z}(t), t, \theta) \\ -\operatorname{Tr}\left(\frac{\partial f}{\partial \mathbf{Z}(t)}\right) \end{bmatrix} \tag{69}$$

where the initial values are

$$\begin{bmatrix} \mathbf{Z}(t_1) \\ \log p(\mathbf{Z}) - \log p(\mathbf{Z}(t_1)) \end{bmatrix} = \begin{bmatrix} \mathbf{Z} \\ 0 \end{bmatrix} \tag{70}$$

One can then use the second element from the solution in Eq. 69 to give the acceptance probability as

$$a\big(\mathbf{Z}(t_1) | \mathbf{Z}(t_0)\big) = \min\left(1, \frac{\exp\left(-\beta H(\mathbf{Z}(t_1))\right)}{\exp\left(-\beta H(\mathbf{Z}(t_0))\right)} + \frac{p(\mathbf{Z}(t_1))}{p(\mathbf{Z}(t_0))}\right) \tag{71}$$

A.5 Quaternions

Quaternions are a tuple of 4 numbers

$$\mathbf{q} = \begin{bmatrix} w & x & y & z \end{bmatrix} = \begin{bmatrix} w & \mathbf{v} \end{bmatrix} = w + x\mathbf{i} + x\mathbf{j} + q_3\mathbf{k} \tag{72}$$

where w is the scalar part and $\mathbf{v}$ is the vector part with components x, y and z. In rotations, quaternions are always normalised by the norm

$$||\mathbf{q}|| = \sqrt{w^2 + x^2 + y^2 + z^2} \tag{73}$$

Unit quaternions describe a rotation in 3D. To rotate a vector $\mathbf{r}$ (for instance, angular velocity or momentum), a conjugation is performed to give the rotated vector as

$$\mathbf{r}' = \mathbf{q} \otimes \begin{bmatrix} 0 & \mathbf{r} \end{bmatrix} \otimes \mathbf{q}^* \tag{74}$$

where the conjugate quaternion is defined as

$$\mathbf{q}^* = \begin{bmatrix} w & -x & -y & -z \end{bmatrix} \tag{75}$$

The inverse of a quaternion describes the opposite rotation and for a unit quaternion is given by

$$\mathbf{q}^{-1} = \frac{\mathbf{q}^*}{||\mathbf{q}||} = \mathbf{q}^* \tag{76}$$

Note that quaternion multiplication is *non-commutative* ($\mathbf{q}_1 \otimes \mathbf{q}_2 \neq \mathbf{q}_2 \otimes \mathbf{q}_1$) and is defined by Hamilton's product as

$$\mathbf{q}_1 \otimes \mathbf{q}_2 = \begin{bmatrix} w_1 w_2 - \mathbf{v}_1 \cdot \mathbf{v}_2 & w_1\mathbf{v}_1 + w_2\mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2 \end{bmatrix} \tag{77}$$

This can be re-written

$$\mathbf{q}_1 \otimes \mathbf{q}_2 = \begin{bmatrix} w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2 \\ w_1 x_2 + w_2 x_1 + y_1 z_2 - y_2 z_1 \\ w_1 y_2 + w_2 y_1 - x_1 z_2 + x_2 z_1 \\ w_1 z_2 + w_2 z_1 + x_1 y_2 - x_2 y_1 \end{bmatrix} \tag{78}$$

From inspection, this gives the matrix product:

$$\mathbf{q}_1 \otimes \mathbf{q}_2 = \begin{bmatrix} w_2 & -x_2 & -y_2 & -z_2 \\ x_2 & w_2 & z_2 & -y_2 \\ y_2 & -z_2 & w_2 & x_2 \\ z_2 & y_2 & -x_2 & w_2 \end{bmatrix} \begin{bmatrix} w_1 \\ x_1 \\ y_1 \\ z_1 \end{bmatrix} \tag{79}$$

This is where $\mathbf{G}$ matrix from Eq. 41 comes from.

A.6 Loss Evolution for 2D Wolfe-Quapp

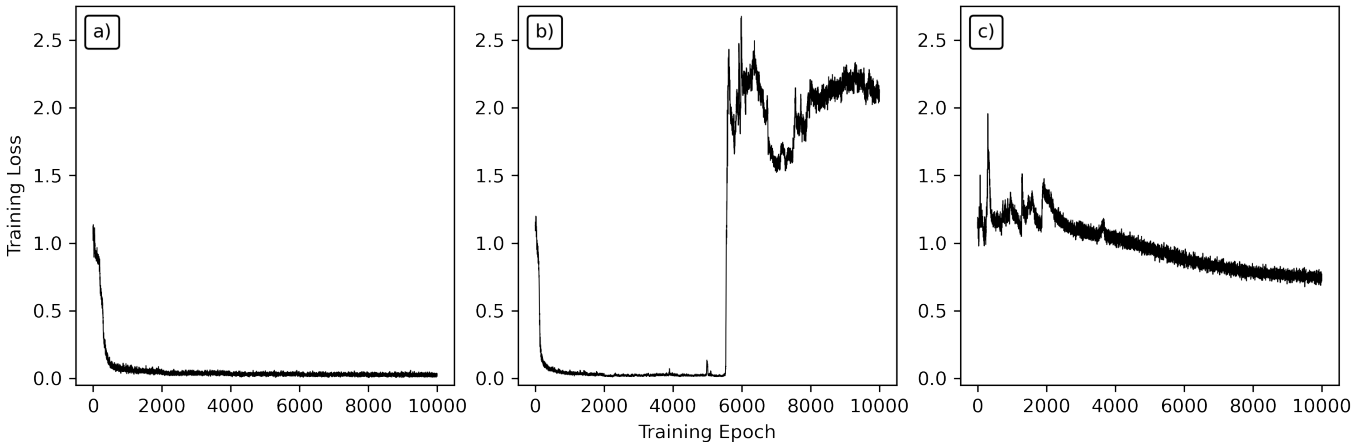


Figure 19: Training loss of independent learning runs. a) Expected monotonically decreasing loss. b) Training instability with an exploding gradient half-way through training that was not corrected for by running for more training epochs. c) Training instability with a slow convergence bounded way above the desired final loss, hence a complete re-initialisation of training is required.

A.7 Input File for 7-Particle Hexagon LAMMPS Simulations

```
# Input file for LAMMPS 2 hexagonal rigid bodies
# Author: Shanil Panara
# Date: 13/01/2022
# Notes:
#   use of MOLECULE package for binding particles in a rigid body
#   NVT simulation

##################################### SETUP ######################################
# ------------------------------ INITIALISATION ------------------------------
units       lj                              # could also be: real, metal, si
dimension   3                               # create box
boundary    p p p                           # all three boundaries are non-periodic
atom_style  full                            # can be used for CG or molecular bodies

# ------------------------------- VARIABLES ----------------------------------
# System Parameters
variable sigma   equal 1.0                  # sigma in lj potential
variable epsilon equal 1.0                  # sigma in lj potential
variable cut     equal 1.12246              # cutoff in lj potential (lj distance units)
variable temp    equal 0.5                  # temperature (lj temperature units)
variable K       equal 10*v_temp/v_cut/v_cut    # spring constant (force/distance units)
variable r0      equal 2                    # equilibrium distance (lj distance units)
variable seed    equal 5                    # random seed value
variable name    string NVE-temp-${temp}_K-${K}_r-${r0}_s-${seed}

# Output File Names
variable traj       string  ${name}-traj.dump
variable rb_info    string  ${name}-info.dat
variable sim_log    string  ${name}-sim.log
variable input_log  string  ${name}-input.log

log ${input_log}                            # start a new log file to log input parameters/lammps response

# Simulation Parameters
variable log_freq equal 100                 # log frequency
variable runsteps equal 10000000            # N of steps to run for
variable timestep equal 0.00001             # timestep

# -------------------------- ATOMS/MOLECULE DEFINITION --------------------------
read_data hex.conf                          # Add a hexagon molecule
read_data hex.conf add append shift 4.0 0.0 0.0
# Add a second identical molecule, shifted so it doesn't overlap

# Group ID can be used in other commands, including fix, compute, dump or velocity
group       hex_1 id 1 2 3 4 5 6 7          # label the atoms 1-7 as a group
group       hex_2 id 8 9 10 11 12 13 14     # label the atoms 8-14 as another group

# -------------------------- INTERATOMIC POTENTIALS ----------------------------
# Pairwise potentials
pair_style  lj/cut  ${cut}                  # lj truncated - cutoff at 1.12 sigma
pair_coeff  *   *   1.0 ${sigma}        # epsilon | sigma - * * means all atoms interact with each other
fix hex_spring_NVT hex_1 spring couple hex_2 1.0 0.0 0.0 0.0 ${r0}
# adds harmonic spring between rigid bodies
                            # f_hex_spring[1,2,3,4] = force_x, force_y, force_z, magnitude of the force


################### SYSTEM INITIALISATION WITH AN NVT SIMULATION ##################
# -------------------------------------------------------------------------------
# Add energy to the system by running an NVT simulation, use a very stiff spring to
# keep rigid bodies close. Then, remove (unfix) the thermostat and spring before
# initialising and running the more important NVE simulation
# -------------------------------------------------------------------------------

# System fixes: spring and nose-hoover thermostat
fix hex_spring_NVT hex_1 spring couple hex_2 1 0 0 0 ${r0} # adds harmonic spring between RBs
fix rigid_1_NVT hex_1 rigid/nvt/small molecule temp ${temp} ${temp} 1.0 tparam 10 3 3
fix rigid_2_NVT hex_2 rigid/nvt/small molecule temp ${temp} ${temp} 1.0 tparam 10 3 3
```

```
# Basic output to terminal
thermo ${log_freq}
thermo_style custom step f_hex_spring_NVT[*] temp pe ke etotal

# Set temperature and run simulation
velocity hex_1 create ${temp} ${seed}
run 0
velocity all scale ${temp}
run 100000

unfix rigid_1_NVT
unfix rigid_2_NVT
unfix hex_spring_NVT

################################ NVE SIMULATION ###################################
# ------------------------------ SYSTEM FIXES ---------------------------------
fix rig_hex_1 hex_1 rigid/nve/small molecule
fix rig_hex_2 hex_2 rigid/nve/small molecule
fix hex_spring hex_1 spring couple hex_2 ${K} 0.0 0.0 0.0 ${r0} # spring between RBs
# f_hex_spring[1,2,3,4] = force_x, force_y, force_z, magnitude of the force

# ---------------------------- PROPERTY COMPUTES ------------------------------
# Calculate rigid body properties - local data, so must be outputted using dump ... "local"
compute com_1   all rigid/local rig_hex_1 xu yu zu  # RB1 unscaled center of mass x, y, z
compute com_2   all rigid/local rig_hex_2 xu yu zu  # RB2 unscaled center of mass x, y, z
compute vel_1   all rigid/local rig_hex_1 vx vy vz  # RB1 center of mass velocities vx, vy, vz
compute vel_2   all rigid/local rig_hex_2 vx vy vz  # RB2 center of mass velocities vx, vy, vz
compute q_1     all rigid/local rig_hex_1 quati quatj quatk quatw   # RB1 quaternion components
compute q_2     all rigid/local rig_hex_2 quati quatj quatk quatw   # RB2 quaternion components
compute av_1    all rigid/local rig_hex_1 omegax omegay omegaz      # RB1 angular velocities
compute av_2    all rigid/local rig_hex_2 omegax omegay omegaz      # RB2 angular velocities
compute am_1    all rigid/local rig_hex_1 angmomx angmomy angmomz   # RB1 angular momenta
compute am_2    all rigid/local rig_hex_2 angmomx angmomy angmomz   # RB2 angular momenta
compute i_1    all rigid/local rig_hex_1 inertiax inertiay inertiaz # RB1 inertia
compute i_2    all rigid/local rig_hex_2 inertiax inertiay inertiaz # RB2 inertia

# ---------------------------- CONFIGURE OUTPUTS ------------------------------
thermo_style custom step f_hex_spring[*] temp pe ke etotal
dump    1 all custom ${log_freq} ${traj}       id type x y z
dump    2 all local  ${log_freq} ${rb_info}  c_com_1[*] c_com_2[*] &
    c_vel_1[*] c_vel_2[*] c_q_1[*] c_q_2[*] c_av_1[*] c_av_2[*]    &
    c_am_1[*] c_am_2[*] c_i_1[*] c_i_2[*]

log ${sim_log} # create new log file with simulation information only
# ------------------------------ RUN SIMULATION -------------------------------
timestep ${timestep}
run ${runsteps}

print "Simulation stored with name: ${name}"

################################### NOTES #########################################
# Each new simulation must be run in a new directory
```