

Using Discrete Variational Mechanics as a Prior in Deep Learning

Saúl José Rodrigues dos Santos

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisors: Prof. Rodrigo Martins de Matos Ventura
Monica Jeevan Ekal

Examination Committee

Chairperson: Prof. José Fernando Alves da Silva

Supervisor: Prof. Rodrigo Martins de Matos Ventura

Member of the Committee: Prof. Mário Alexandre Teles de Figueiredo

June 2022

To my family

Acknowledgments

First of all, I would like to thank my parents and sister for providing me the means to accomplish my academic education and for all the support during these years.

I would like to thank my supervisors Prof. Rodrigo Ventura and Monica Ekal for all the time spent during the meetings, guiding me throughout this work.

Finally, I would like to thank all my friends and family for all the good times during these last years.

Resumo

Com a aprendizagem profunda a ganhar mais atenção da comunidade para predição e controlo de sistemas físicos reais, aprender representações importantes está se a tornar agora mais que nunca relevante. É de extrema importância que representações de aprendizagem profunda sejam coerentes com a física. Desenvolvimentos recentes possibilitaram a combinação de experiência prévia sobre as equações diferenciais contínuas com redes neurais. Enquanto isto leva a modelos eficientes em termos de dados e com boa capacidade de generalização, isto pode não ser o caso para dados discretos, onde discretização do sistema contínuo adjacente é necessária. De facto, usando experiência prévia com integradores tradicionais propriedades como a simplecticidade e a conservação do momento das equações contínuas originais não são preservadas durante a discretização.

Neste trabalho apresentamos *Symplectic Momentum Neural Networks* (*SyMo*) como modelos vindos de um formulão discreta da mecânica para sistemas mecânicos não separáveis. A combinação de tal formulão leva os *SyMos* a serem forçados a preservar estruturas geométricas importantes como o momento e uma forma simpléctica bem como aprender com dados limitados. Nós estendemos os *SyMos* a incluir integradores variacionais na estrutura de aprendizagem, desenvolvendo uma camada implícita que calcula raízes. Isto dá origem a *End-to-End Symplectic Momentum Neural Networks* (*E2E-SyMo*). Através de resultados experimentais usando sistemas de referência como o pêndulo, o *cartpole* e o *acrobot* nós mostramos que tal combinação leva estes models a aprender com menos dados mas também fazem com que estes modelos preservem a forma simpléctica levando a melhor comportamento de longa duração.

Palavras-chave: Aprendizagem condicionada com física, Integradores Variacionais, Aprendizagem profunda, Mecânica Discreta

Abstract

With deep learning gaining attention from the research community for prediction and control of real physical systems, learning important representations is becoming now more than ever relevant. It is of extreme importance that deep learning representations are coherent with physics. Recent developments have enabled the combination of priors about the continuous-time differential equations with neural networks. While this leads to data efficient models with good generalization properties, this may not be the case for discrete data where discretization of the adjacent continuous system is required. In fact, when using such priors with traditional integrators, properties from the original continuous time equations such as symplecticity and momentum conservation are not preserved under the discretization flow.

In this work we introduce Symplectic Momentum Neural Networks (SyMo) as models from a discrete formulation of mechanics, for non-separable mechanical systems. The combination of such formulation leads SyMos to be constrained towards preserving important geometric structures such as momentum and a symplectic form, and learn from limited data. We extend SyMos to include variational integrators within the learning framework by developing an implicit root-find layer which leads to End-to-End Symplectic Momentum Neural Networks (E2E-SyMo). Through experimental results, using reference systems, such as the pendulum, the acrobot and the cartpole we show that such combination not only allows these models to learn from limited data but also provides the models with the capability of preserving the symplectic form leading to a better long-term behaviour.

Keywords: Physics-Informed learning, Variational Integrators, Deep Learning, Discrete Mechanics

Contents

| | |
|--|-----------|
| Acknowledgments | v |
| Resumo | vii |
| Abstract | ix |
| List of Tables | xv |
| List of Figures | xix |
| Glossary | xxi |
| 1 Introduction | 1 |
| 1.1 Motivation | 1 |
| 1.1.1 The limits and challenges of deep learning in robotics | 1 |
| 1.1.2 Physically Informed Learning - A new paradigm | 2 |
| 1.2 Regression architectures for model learning | 4 |
| 1.3 Contributions | 5 |
| 1.4 Thesis Outline | 6 |
| 2 Background | 7 |
| 2.1 Artificial Neuron Model | 7 |
| 2.2 Activation functions | 8 |
| 2.3 Artificial Neural Networks | 9 |
| 2.3.1 Concept of Layer in Deep Learning | 9 |
| 2.3.2 Model Representation | 10 |
| 2.3.3 Neural Networks as Universal Approximations of Functions | 10 |
| 2.3.4 Loss Functions for Regression | 11 |
| 2.3.5 Regularization Techniques | 11 |
| 2.4 Neural Network Training | 13 |
| 2.4.1 Automatic Differentiation | 13 |
| 2.4.2 Back-Propagation | 14 |
| 2.4.3 Optimizers | 16 |
| 2.5 Implicit Layers | 17 |
| 3 From Continuous to Discrete Mechanics | 19 |
| 3.1 Continuous Lagrangian Mechanics | 19 |

| | | |
|----------|---|-----------|
| 3.1.1 | Forced Continuous Mechanics | 20 |
| 3.1.2 | Lagrangian Dynamics for Mechanical Systems | 21 |
| 3.2 | Numerical Integration of Differential Equations | 22 |
| 3.3 | Geometric Numerical Integration | 23 |
| 3.3.1 | Symplectomorphisms | 23 |
| 3.3.2 | Discrete Mechanics and Variational Integrators | 24 |
| 3.3.3 | Discrete Mechanics | 25 |
| 3.3.4 | Forced Discrete Mechanics | 27 |
| 3.3.5 | Variational Integrators | 29 |
| 4 | Physics-Guided Deep Learning | 30 |
| 4.1 | Incorporating Lagrangian Mechanics into Deep Learning | 30 |
| 4.1.1 | Ensuring Symmetry and Positive Definiteness of the Inertia Matrix | 30 |
| 4.1.2 | Deep Energy-Based Modeling of Mechanical Systems | 31 |
| 4.1.3 | Discontinuity free Learning | 32 |
| 4.1.4 | Gradient Based End-To-End Learning of Lagrangian Mechanics | 34 |
| 4.2 | Learning ODEs from discrete trajectories | 35 |
| 4.2.1 | Neural Ordinary Differential Equations | 35 |
| 4.2.2 | Gradient Computation through the Adjoint Sensitivity Method | 36 |
| 4.3 | Modeling Mechanical Systems with Neural ODEs | 38 |
| 4.3.1 | Second Order Neural Ordinary Differential Equations | 38 |
| 4.3.2 | ODE solvers | 39 |
| 4.3.3 | Geometric Neural ODE | 40 |
| 4.3.4 | Lagrangian Neural ODE | 40 |
| 5 | Combining Discrete Variational Mechanics with Deep Learning | 43 |
| 5.1 | Symplectic-Momentum Neural Networks | 43 |
| 5.1.1 | Problem Formulation | 43 |
| 5.1.2 | Discretization | 44 |
| 5.1.3 | Parameter Optimization | 47 |
| 5.1.4 | Inference in SyMos | 48 |
| 5.2 | End-to End Symplectic-Momentum Neural Networks | 50 |
| 5.2.1 | Implicit Differentiation through Gray-Box RootFind Solvers | 51 |
| 5.2.2 | RootFind Layer | 51 |
| 5.2.3 | Parameter Optimization | 52 |
| 6 | Results | 54 |
| 6.1 | Dataset Generation | 54 |
| 6.2 | Methods and Metrics | 55 |
| 6.3 | Training Details | 55 |

| | | |
|---------------------|--|-----------|
| 6.4 | Task 1 - Pendulum | 56 |
| 6.4.1 | Generalization Capabilities | 56 |
| 6.4.2 | Long Term Integration | 58 |
| 6.4.3 | Learned Quantities | 59 |
| 6.4.4 | Prediction with Forcing | 59 |
| 6.5 | Task 2 - Acrobot | 60 |
| 6.5.1 | Generalization Capabilities | 60 |
| 6.5.2 | Test Trajectory | 61 |
| 6.6 | Task 3 - Cartpole | 63 |
| 6.6.1 | Generalization Capabilities | 63 |
| 6.6.2 | Test Trajectory | 64 |
| 6.6.3 | Prediction with Forcing | 65 |
| 6.6.4 | Training with noise | 66 |
| 7 | Conclusions | 68 |
| 7.1 | Future Work | 69 |
| Bibliography | | 69 |
| A | Implicit Differentiation in SyMos | 77 |
| A.1 | Uniqueness and Existence of solutions in Symplectic-Momentum Neural Networks | 77 |
| A.2 | Backward Pass for the Implicit RootFind Layer | 78 |
| B | Models | 81 |
| B.1 | Physical Pendulum | 81 |
| B.2 | Acrobot | 82 |
| B.3 | Cartpole | 83 |

List of Tables

| | | |
|-----|---|----|
| 2.1 | Common Non-linear activation functions for regression. | 8 |
| 6.1 | Overview of the methods discussed. | 55 |
| 6.2 | Quantitative results for the moderate data regime - pendulum. | 57 |
| 6.3 | Quantitative results for the moderate data regime - acrobot. | 61 |
| 6.4 | Quantitative results for the moderate data regime - cartpole. | 64 |
| B.1 | Parameters for the pendulum. | 82 |
| B.2 | Parameters for the acrobot | 83 |
| B.3 | Parameters for the cartpole. | 84 |

List of Algorithms

| | | |
|---|--|----|
| 1 | Back-Propagation Algorithm | 15 |
| 2 | Adam Optimizer | 17 |
| 3 | Newton's Method for inference in SyMo. | 50 |

List of Figures

| | |
|---|----|
| 1.1 Bias-Variance Tradeoff | 2 |
| 1.2 Energy behaviour of a variational integrator | 3 |
| 2.1 Analogy between biological neuron and its computer simulation. | 7 |
| 2.2 Feedforward Neural Network Architecture | 9 |
| 2.3 Underfitting vs Overfitting | 12 |
| 2.4 Computational graph. | 14 |
| 2.5 Back-propagation from a single hidden layer neural network perspective. | 15 |
| 2.6 Explicit vs Implicit Layers. | 17 |
| 3.1 Phase Space for the Simple Pendulum obtained from a non geometric integrator. | 24 |
| 3.2 Left and right discrete forces. | 28 |
| 4.1 Cholesky Decomposition of the Inertia Matrix. | 31 |
| 4.2 Modeling the energy of a mechanical system. | 32 |
| 4.3 Angle embedding on \mathbb{T}^m | 33 |
| 4.4 Angle embedding on $\mathbb{R}^n \times \mathbb{T}^m$ | 33 |
| 4.5 Reverse-mode differentiation of an ODE solution. | 38 |
| 4.6 Geometric Neural ODE. | 40 |
| 4.7 L-NODE - Lagrangian Neural ODE. | 42 |
| 5.1 Angle embedding and collocation point on \mathbb{T}^m | 46 |
| 5.2 Angle embedding and collocation point on $\mathbb{R}^n \times \mathbb{T}^m$ | 47 |
| 5.3 SyMo - Symplectic-Momentum Neural Networks. | 48 |
| 5.4 General RootFind Layer. | 52 |
| 5.5 E2E-SyMo - End-to-End Symplectic-Momentum Neural Networks. | 53 |
| 6.1 Train, test and integration loss for the pendulum. | 56 |
| 6.2 Energy and Inertia MSE for the pendulum.. | 57 |
| 6.3 Pendulum Phase Spaces | 58 |
| 6.4 Energy and Configuration Space MSE | 58 |
| 6.5 Learned Quantities - pendulum | 59 |
| 6.6 Trajectory and MSE for the test forced trajectory -pendulum. | 59 |

| | | |
|------|--|----|
| 6.7 | Energy and Inertia for the forced test trajectory - pendulum | 60 |
| 6.8 | Train, Test and Integration error for the acrobot. | 60 |
| 6.9 | Energy and Inertial error for the acrobot. | 61 |
| 6.10 | Trajectories for the test trajectory - acrobot | 62 |
| 6.11 | Energy and MSE for the test trajectory for the acrobot. | 62 |
| 6.12 | Learned quantities for the acrobot. | 62 |
| 6.13 | Train, test and integration loss for the cartpole. | 63 |
| 6.14 | Energy and Inertial Error for the cartpole. | 63 |
| 6.15 | Trajectories for the cartpole. | 64 |
| 6.16 | Energy and MSE for the test trajectory cartpole. | 65 |
| 6.17 | Learned quantities for the cartpole. | 65 |
| 6.18 | Forced trajectories for the cartpole. | 66 |
| 6.19 | Train, Test and Integration Loss for the noisy training data. | 66 |
| B.1 | Physical Pendulum. | 81 |
| B.2 | Acrobot | 82 |
| B.3 | Cartpole | 84 |

Acronyms

AD Automatic Differentiation.

DEL Discrete Euler-Lagrangian Equations.

DELAN Deep Lagrangian Neural Networks.

E2E-SyMo End-to-End Symplectic-Momentum Neural Networks.

GP Gaussian Processes.

MLP Multi-Layer Perceptron.

MPC Model Predictive Control.

MSE Mean Squared Error.

NODE Neural Ordinary Differential Equations.

ODE Ordinary Diferential Equation.

RK2 Runge-Kutta 2nd Order.

RK4 Runge-Kutta 4th Order.

SyMo Symplectic-Momentum Neural Networks.

VI Variational Integrators.

Chapter 1

Introduction

1.1 Motivation

1.1.1 The limits and challenges of deep learning in robotics

Even though deep learning was subject to an unparalleled boost during the last decade and being practically at the forefront of areas like Computer Vision [1] or even language comprehension [2], there is still a substantial gap in the field of robotics. Specially for model-based learning, where there is an inherent need for high accuracy, faster adaptation and proper ability to extrapolate the learned models beyond the domain where they were trained [3].

As the robot's tasks and environments become increasingly complex, endowing the controllers of the robots with the capability of learning a model of their kinematics and dynamics under changing circumstances, is becoming now more than ever mandatory. For this purpose, there is a inherent need for accurate long-term prediction, interpretability, and data-efficient learning, while still remaining flexible enough and capable of modeling complex behavior.

When specifying a model, one faces the question of whether to look for knowledge about the plant, neglecting hard-to-model effects (white-box approach) [4], or to go for a fully data-driven approach (black-box approach). The former would make assumptions about the structure of the model, such as its kinetic structure and inertial properties. The latter models the dynamics as any other function typical from machine learning, needing a large amount of data in order to generalize well due to model variance. The ideal is to combine both into a gray-box approach that uses neural networks alongside knowledge about the plant, achieving an equilibrium between model bias and variance, as summarized in figure 1.1.

A recent line of research has experienced a tremendous growth by considering the integration of traditional physics-based modeling approaches, such as conservation laws, with state-of-the-art machine learning techniques [4–8]. By doing so, some of the problems of the traditional deep learning techniques can be mitigated. In reality, traditional deep learning techniques rely on a purely data driven approach and hence are limited to the characteristics of the data. Even though, expressive enough to model arbitrary phenomena, these models are still considered black-box parameterizations, failing to provide

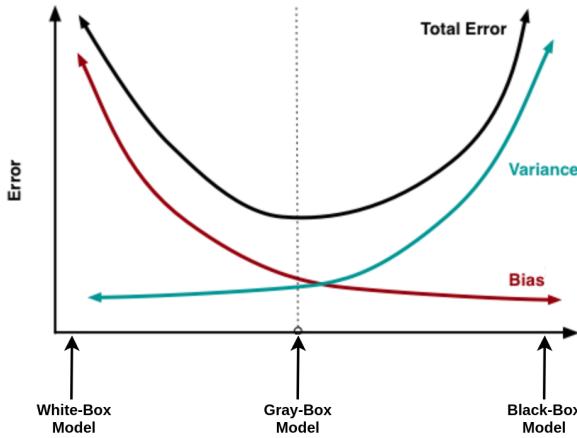


Figure 1.1: Current approaches to white-box modeling using analytic methods can suffer from high bias, while black-box modeling using neural networks can provide models with high variance. Gray-box models incorporate prior knowledge with neural networks offering models with reduced bias and capability to extrapolate to regimes beyond the models were trained. Adapted from ¹.

physically-rich models .

The application of even the state-of-the-art black box models has very frequently met with limited success in scientific domains due to their large data requirements, inability to produce physically consistent results, and their lack of generalizability to out-of-sample scenarios [9]. By needing large amounts of training data in order to generalize well to unseen states, another problem comes up which is the fact that such data can be prohibitively expensive to obtain on real robotics systems. Additionally, many robotic control frameworks, such as feedback linearization, leverage model structure derived from first principles [6].

By ignoring physical principles, black-box approaches fail to produce consistent long term results due to error accumulation, specially for long term simulation of robotic systems. Gray-box models, on the other hand, offer resistance to error accumulation.

1.1.2 Physically Informed Learning - A new paradigm

The need for more viable models led researchers to look to endowing traditional deep learning techniques with prior knowledge about the plant. In fact, this was foreseen in [3]. We will debate the introduction of inductive biases in neural networks in the context of differential equations, as they are the underlying structure present in mechanical systems.

In fact, in [10, 11] authors used the knowledge of specific differential equations to structure the learning problem and achieve lower sample complexity. With this, they were able to solve partial differential equations in a learning fashion with physical plausibility and good generalization properties. However, this approach assumes that the non-linear part of the differential is known a priori, which lead the same authors to extend his work [12] not only to learn the solution but also to discover the underlying nonlinear partial differential equation.

¹<https://scott.fortmann-roe.com/docs/BiasVariance.html>.

Following this line of research authors decided to propose an architecture that imposes Lagrangian mechanics in a structured approach that is optimized to minimize the violation of the differential equations of motion of robot manipulators [5, 13]. In [14] authors generalized to include all types of Lagrangian and extended to Hamiltonian mechanics in [7, 15, 16].

All the methods mentioned assume continuous-time equations of motion for dynamical systems which are given by a set of differential equations that can be derived from its Lagrangian or Hamiltonian via variational calculus. These equations encode the underlying physical properties, such as symmetries corresponding to conservation laws, i.e., energy and momentum and once they are used as constraints in the learning process then there will be conservation of those properties. However, for model-based methods, such as model-based control, these continuous models need to be numerically integrated in order to make usage of the forward model.

Typical numerical integrators, such as the Runge-Kutta 4th Order (RK4), despite having good local behavior do not account for the inherent geometric structure of the governing continuous-time equations and consequently fail to preserve those properties. That can be, however, accomplished by means of *geometric integration* [17]. If the underlying geometric structure is not preserved under the discretization process, then much likely the simulation may be following wrong patterns. Exploring the combination of such strong integrators with deep learning techniques is of extreme importance.

In [18], Chen et al. proposed Neural Ordinary Differential Equations (NODE), differentiable ODE solvers with $O(1)$ memory backpropagation, which minimizes the distance between the generated time series with the observed data. This lead researchers to propose algorithms that combine methods with prior knowledge about the underlying equations of motion with geometric discretization schemes [8, 15].

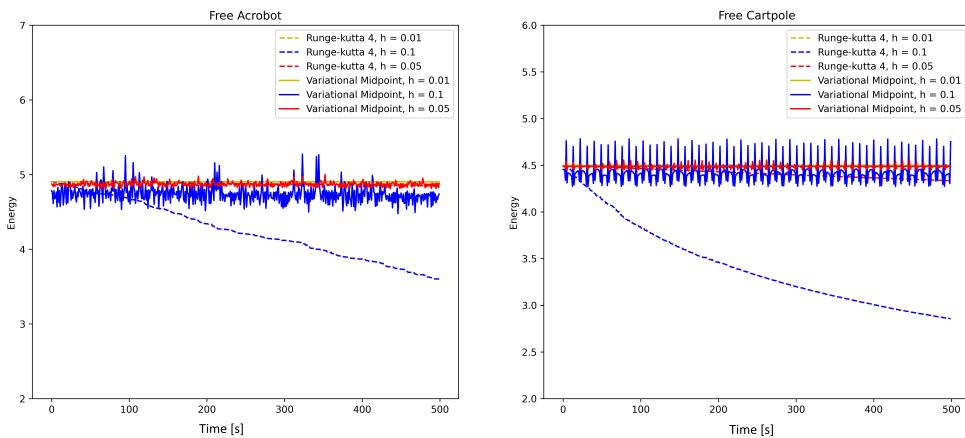


Figure 1.2: Energy computed with variational second-order midpoint rule and fourth-order Runge-Kutta. (a) Energy behaviour for the free acrobot. When unforced, the acrobot is a chaotic system, meaning that the system is highly sensitive to initial conditions. Even though second-order accurate, VI preserve the energy better than the fourth-order integrator RK4, specially for larger time-steps. (b) Same experiment but for a free cartpole. Note that the variational integrator, independently of the time-step, bounds the true energy, revealing only less accuracy for large time-steps, whereas the Runge-kutta dissipates energy due to numerical damping [19].

In [20] authors proposed learning algorithms that capture the dynamics of physical systems from

observed discrete trajectories based on a class of geometric discretization scheme called Variational Integrators (VI). These subsequent integrators have the advantage of conserving momentum and a symplectic form as well as bounding energy behavior providing considerable improvements over conventional integrators for long term simulations (see figure 1.2) and hence they are extremely useful for applications where energy is important such as space missions [21].

All the aforementioned works explored Hamiltonians and Lagrangians that do not have the inertia matrix dependent on the configuration, i.e., they are separable, and within this case one can get an explicit integrator and integrate it with neural networks, which relates the next configuration based on the previous [4, 6, 15]. For articulated mechanical systems the Lagrangians are generally non-separable, and in order to use these powerful simulators, it is necessary to solve an implicit equation. To make use of physical regularizers for deep learning methods, these need to be fully differentiable as the only way to be compatible with backpropagation algorithms, which is not accomplished with an implicit equation. This can, however, be accomplished by means of implicit differentiation that deals with the implicit equation.

1.2 Regression architectures for model learning

In order to acquire more autonomy and operate within the real world, it is desirable that robots are endowed with the following capabilities: getting information about their environments via sensors that deliver high-dimensional data; processing redundant or sparse information with low response delay and energy efficiency; behaving correctly under dynamic and changing conditions, which requires a self-learning ability [22].

In the subject of learning mechanical models, most machine learning techniques are used either to fit the inverse or the forward dynamics [23]. One very common and well studied approach for learning mechanical models of robots is Gaussian Processes (GP). GPs provide a probabilistic non-parametric modelling for black-box identification of mechanical systems. As they are one of the first methodologies to incorporate physical priors with regressor, GPs share the unique property of providing measurements of the prediction uncertainties, which allows this regressor to be used with Linear or Non-Linear Model Predictive Control (MPC) schemes [24]. Due to the $O(n^3)$ computation and $O(n^2)$ storage complexity of the standard GPs [3] several practical approaches have been implemented very recently such as *Sparse Gaussian Process Regression* [25–27] or even *Local Gaussian Processes* [28, 29], where rigid body dynamics are combined with a residual learned term via GPs to account for unmodelled dynamics.

These practical approaches made the GPs less computational expensive, which allows them to be implemented in real time making GPs strong candidates for online model learning with a certain degree of knowledge of the model's dynamics. This allows the adaptation of the model structure with data complexity [23] allowing for a range of controllers that exploit the model structure to be applied in an probabilistic way.

Neural Networks can, like Gaussian Processes, be used to model static nonlinearities and can consequently be used for dynamic system modelling. Examples of such methods can be seen in [30], where

recurrent neural networks are used for food cutting purposes alongside with model-predictive control or in [31] where a linearization of a feed forward neural network is used with MPC for a pneumatic soft robot.

1.3 Contributions

In this thesis, we address the limitations mentioned above by combining Lagrangian dynamics with deep learning techniques from observed discrete trajectories with the usage of neural ODEs with traditional integrators such as the RK4 and Runge-Kutta 2nd Order (RK2) for toy mechanical systems (pendulum, cartpole and acrobot). This is accomplished by learning the inertia matrix and potential energy of the dynamical system and then build the resulting Euler-Lagrange equations that form the system's dynamics via neural ODEs. This leads to Lagrangian neural ordinary differential equations (L-NODE).

Further, we propose to combine discrete mechanics for Lagrangians which have inertia dependent on the configuration (non-separable) with deep learning. Following the same parameterization but adapted to discrete mechanics, we build instead the Discrete Euler-Lagrangian Equations (DEL) and use them to make predictions via variational integration. This gives origin to Symplectic-Momentum Neural Networks (SyMo).

Finally, the novelty resides in performing end-to-end learning by including variational integrators within the learning framework. This is accomplished by adding an implicit layer that solves the root finding procedure to the implicit discrete Euler-Lagrange equations. With the help of the implicit function theorem, we perform implicit differentiation through the implicit layer without the need to back-propagate through the intermediate steps of the root finding algorithm. This gives origin to End-to-End Symplectic-Momentum Neural Networks (E2E-SyMo)

We want to make a comparison of the methods with second-order accurate variational integrators with the ones coming from NODEs, with and without prior, with traditional integrators and show that the preservation of important properties by the methods that combine discrete mechanics leads to better long term behaviour.

The main contributions of this thesis can be summarized as:

- Combine NODEs with Lagrangian mechanics with the RK2 and RK4 integrator by parameterizing the potential energy and the inertia matrix and build the Euler-Lagrange equations leading to L-NODEs.
- Development of Symplectic Momentum Neural Networks for non-separable systems by combining discrete variational mechanics with deep learning techniques and using the resulting learned implicit discrete Euler-Lagrange equations to make predictions via a root-finding algorithm.
- Development of End-to-End Symplectic Momentum Neural Networks by accommodating the root-finding process via implicit differentiation to the learning framework.

- Simulation results for the pendulum, acrobot and cartpole showing that the integration of physical priors from discrete and continuous mechanics leads to more data-efficient models when compared with the black-box NODEs.
- Analysis and comparison of the short and long term behaviour of the methods proposed (NODE, L-NODE, SyMo and E2E-SyMo).
- Our code with the experiments and models is made available in <https://github.com/ssantos97/SyMo>
- A paper [32] accepted at the 4th Annual Learning for Dynamics & Control Conference taking place in Stanford University on June 23-24, 2022.

1.4 Thesis Outline

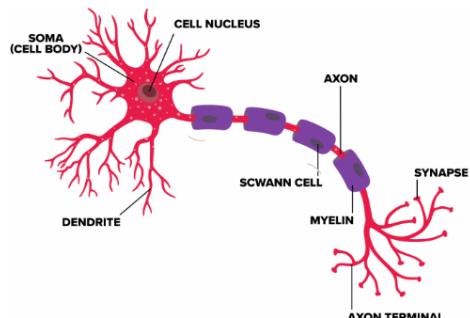
This work is organized as follows. In Chapter 2, the background regarding deep learning is introduced. It approaches the core deep learning definitions used throughout this work. Next, derivation of continuous mechanics and discrete mechanics, as well as some basics about numerical integration are presented in Chapter 3. Chapter 4 addresses the combination of continuous mechanics with deep learning methods through NODE and establish the baselines used throughout this work. Chapter 5 addresses the main contribution of this work where we combine discrete variational mechanics with deep learning techniques. We extend this combination to account for variational integration through a developed implicit root-find layer. The reader is referred to annex A.2 for detailed derivation of the implicit layer. Finally, in chapter 6 results are shown. One refers to annex B for the system’s dynamics of the toy models used within this work.

Chapter 2

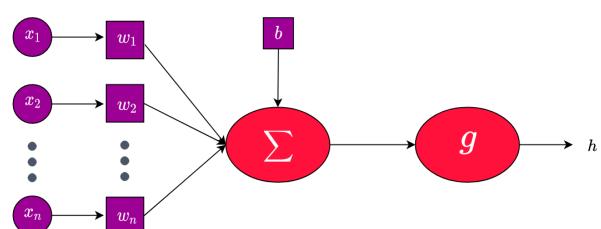
Background

2.1 Artificial Neuron Model

Even though artificial neurons are said to resemble the computations of biological neurons to create an agent that can learn, one must be prudent as the modern term "artificial neural networks" goes way beyond the neuroscientific perspective. It appeals to a more general principle of learning multiple levels of composition, which can then be applied in machine learning frameworks that are not necessarily neurally inspired [33]. Yet, we can easily see the similarities between an artificial and biological neurons, i.e., the artificial neurons receive some values from the sensory input or other neurons and after mathematical processing they generate an output that can be provided once more to another neuron (figure 2.1).



(a) Neuron anatomy. In a neuron there exists a nucleus, dendrites (inputs), weight associated with each input, and axons (outputs from the neurons).



(b) Artificial neuron - Perceptron. A perceptron is a single layer neural network that gives a single output.

Figure 2.1: Analogy between biological neuron and its computer simulation. Retrieved from ¹.

Let $x_i \in \mathbb{R}^{n_x}$ be the set of input sensory to the neuron, where n_x is the input size, and $w_i \in \mathbb{R}^{n_x}$ be the set of weights. Finally, let us consider the input feature $x_0 = 1$ and consider $w_0 = b$, where b represents the bias term then in mathematically language a single neuron input can be represented by a linear transformation of the weights that represent the importance of each connection between the neuron and the input itself:

¹<https://smartboost.com/blog/deep-learning-vs-neural-network>

$$z = w_1x_1 + w_2x_2 + w_i x_i + \dots + b = \sum_{n=1}^{n_x} w_i x_i + b. \quad (2.1)$$

Or in matrix form, assuming $X \in \mathbb{R}^{n_x}$ and $W \in \mathbb{R}^{1 \times n_x}$:

$$z = WX + b. \quad (2.2)$$

The inputs are then passed through a function, $g(x, w)$, called activation function used to describe the features by the following relationship:

$$h = g(WX + b), \quad (2.3)$$

where g is the activation function that induces an affine transformation controlled by the learned parameters.

2.2 Activation functions

Activation functions are of crucial importance in deep learning insofar as they are used to allow neural networks to get complex patterns from data. The role of the activation function is to transform an input signal into an output signal but also to keep the value of the neuron output within an acceptable range in order to avoid numerical issues, specially for very deep neural networks. As our work is based on regression task we now present some of the most common activation functions for this purpose.

| Name | Function | Derivative | Figure | Range |
|------------|---|---|--------|---------|
| Sigmoid | $\sigma(x) = \frac{1}{1+e^{-x}}$ | $\sigma'(x) = \sigma(x)(1 - \sigma(x))$ | | (0, 1) |
| tanh | $\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ | $\sigma'(x) = 1 - \sigma(x)^2$ | | (-1, 1) |
| ReLU | $g(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0. \end{cases}$ | $g'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0. \end{cases}$ | | [0, ∞) |
| Leaky ReLU | $g(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0. \end{cases}$ | $g'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0. \end{cases}$ | | [0, ∞) |
| Softplus | $g(x) = \ln(1 + e^x)$ | $g'(x) = \frac{1}{1+e^{-x}}$ | | (0, ∞) |

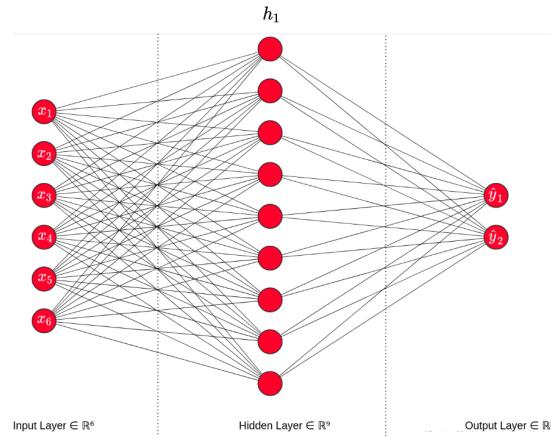
Table 2.1: Common Non-linear activation functions for regression.

Although these are the most popular choices for regression (table 2.1), many other functions have been successfully used as activation functions, including non monotonic functions such as gaussians

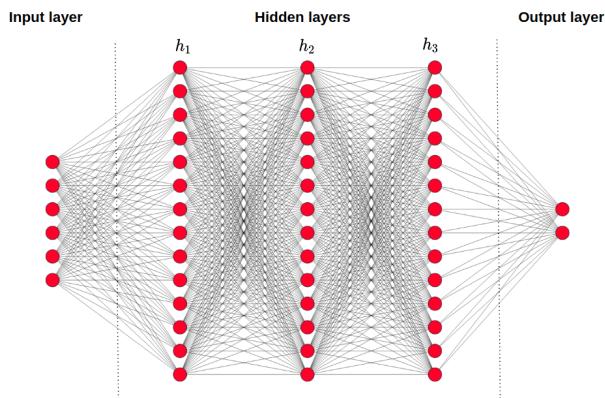
and sinusoids or even softmax for classification problems. Activation functions play a major role in neural networks as they inherit the differentiability of those which in turn will be of extreme importance in the learning process. The right choice of activation function will surely define the best accuracy in the final optimized model.

2.3 Artificial Neural Networks

This main idea of artificial neural networks is to combine multiple neurons in parallel in a sequential fashion. Artificial neural networks are then a sequence of layers composed with several individual neurons in parallel where each layer is connected through neuron edges to the adjacent layers (figure 2.2).



(a) Vanilla Neural Network - Multilayer Perceptron with one single layer



(b) Deep Multilayer Perceptron with 3 hidden layers. Although a Multilayer perceptron can be composed by several hidden layers.

Figure 2.2: Feedforward Neural Network Architecture

2.3.1 Concept of Layer in Deep Learning

Feedforward neural networks are one of the most well studied architectures in deep learning. As already mentioned, these models are called feedforward because information flows through the function

being evaluated from the input, through the intermediate computations used to define that function, and finally to the output. In this network architecture, neurons are organized in independent layers where each layer has some neurons that define the neural network width. On the other hand, the number of layers define the depth of the neural network and they are responsible for processing information coming from previous layers, hence the name “deep learning”.

We can split neural networks architecture into three chunks. First the input layer is the layer that transmits the input information of the incoming pattern to the hidden layers. The number of neurons in the input layer is generally determined by the number of input parameters or characteristics of the problem. Second, the hidden layers are responsible to process the input patterns in such a way that we get the desired output. The final chunk is called the output layer and is responsible for processing the information from the last hidden layer providing the desired output of the neural network (figure 2.2).

2.3.2 Model Representation

Throughout this report we follow the approach of [33] and [34] with an adapted notation. A Feed-forward neural network can be described by a series of functional transformations of the input variables x_1, \dots, x_{n_x} in the form

$$a_j^{[l]} = g^{[l]} \left(\sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]} \right) = g^{[l]}(z_j^{[l]}), \quad (2.4)$$

where $j = 1, \dots, m$ is the index corresponding to the neurons of the next layer. From now on, we shall refer to $w_{jk}^{[l]}$ as *weights* and to $b_j^{[l]}$ as *biases* in the layer $[l]^{th}$ of the neural network. The quantities a_j are known as *activations*. Each of them is transformed using a differentiable, nonlinear *activation function* $g(\cdot)$. This process keeps repeating through layers until we reach the last layer where output unit activations are transformed using an appropriate activation function to give a set of network outputs \hat{y}_k .

2.3.3 Neural Networks as Universal Approximations of Functions

Neural networks are considered universal function approximators according to the *universal approximation theorem*. This was first proved for multilayer perceptrons in [35] for the sigmoid activation function and then extended to an arbitrary activation function by [36].

Theorem 2.3.1. *Let $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ be a continuous activation function that is not a polynomial. Let $V = \mathbb{R}^d$ be a real finite dimensional vector space. Then any continuous map $f : V \rightarrow \mathbb{R}$ can be approximated, in the sense of uniform convergence on compact sets, by maps $\tilde{f} : V \rightarrow \mathbb{R}$ of the form*

$$\hat{f}(x_1, \dots, x_d) = \sum_{n=1}^N c_n \sigma \left(\sum_{s=1}^d w_{ns} x_s + b_n \right), \quad (2.5)$$

with some coefficients c_n , w_{ns} and b_n .

Theorem 2.3.1 [37] not only states that the result of the first layer \tilde{f} can approximate any well-behaved function but also that any neural network of greater depth can also approximate the function \tilde{f} by

using the same construction for the first layer and approximating the identity function for the subsequent layers. Given this, multilayer perceptrons can approximate a function to any desired degree of accuracy, provided sufficiently many hidden units are available. Thus, multilayer perceptrons are universal approximators [38]. Moreover, further works showed that multilayer perceptrons are capable of learning not only functions but also its derivatives given an appropriately smooth activation function [39].

The universal approximation theorem means that regardless of what function we are approximating, we know that a large enough multilayer perceptron will be able to represent this function. However, it is not guaranteed that the training algorithm will be able to learn that function. Even if the multilayer perceptron is able to represent the function, there is a possibility that the learning fails for two different reasons. First, the optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function. Second, the training algorithm can fail to approximate the desired function due to overfitting [33].

2.3.4 Loss Functions for Regression

Most deep learning algorithms involve optimization of some sort. Optimization is the task of either minimizing or maximizing some function with respect to a set of parameters. In deep learning context these parameters are generally represented by the weights and biases and the function to be minimized by a *loss function* or *cost function* [34]. Even though some machine learning publications assign different meaning to those terms, we will use them interchangeably.

Loss functions can be considered one of the most important components of neural networks. Therefore, the term *loss* is nothing but a quantification of how far the neural network output is from the desired value.

As our work involves regression tasks, we will focus only on loss functions commonly used within that scope. The most used one for regression tasks is called Mean Squared Error (MSE).

Let $y^{(i)} \in \mathbb{R}^{n_y}$ be the output label or target for the i^{th} example and $\hat{y}^{(i)} \in \mathbb{R}^{n_y}$ be the predicted output by the neural network corresponding to the i^{th} example then we can define the mean squared loss as:

$$L(\hat{y}, y) = \frac{1}{m \times n_y} \sum_{i=1}^m \|\hat{y}^{(i)} - y^{(i)}\|^2, \quad (2.6)$$

where m is the number of examples in the dataset. Intuitively, one can see that this error measure decreases to 0 when $\hat{y}^{(i)} = y^{(i)}$. The error increases whenever the Euclidean distance between the predictions and the labels increases. To obtain an optimal fit for the neural network, the learning algorithm needs then to adjust the parameters in such a way that the loss is reduced when the Neural Network is gaining experience from the training set $\{(x^{(i)}, y^{(i)})\} \in \mathbb{R}^m$.

2.3.5 Regularization Techniques

A central problem in deep learning is how to make an algorithm perform well not only on the training data but also on the test data. After Neural Network training, it is common to find an excellent perfor-

mance on the training set, but not nearly as good on the test set. This may be a sign of *overfitting*, meaning that the model has high *variance* but on the other hand a low *bias*. Bias and variance measure two different sources of error in an estimator. On one side, bias measures the expected deviation of the estimate from the true value of the function or parameter. Variance on the other, provides a measure of the deviation from the estimate or mean that any particular data sample is likely to cause, i.e, the variability of model predictions.

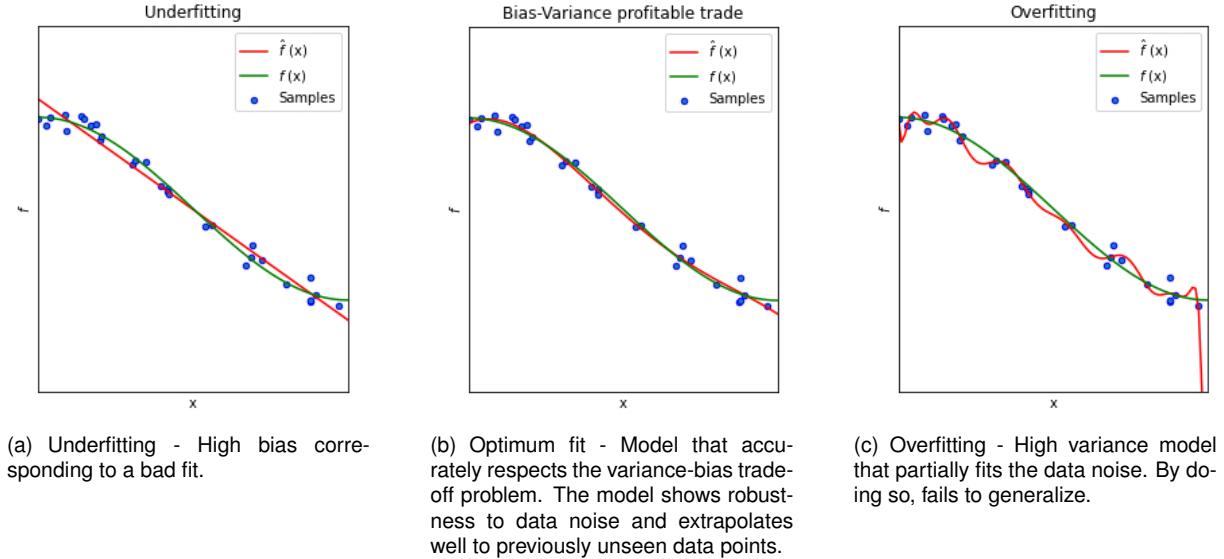


Figure 2.3: Underfitting vs Overfitting - $f(x)$ is the function that represents the data while $\hat{f}(x)$ is a fit to the data corresponding to underfitting, optimal fit and overfitting.

In order to overcome the aforementioned problems, many strategies used in deep learning research are designed to decrease the test error, i.e, increase the generalization capabilities and consequently decrease the variance, possibly at the expense of an increment in the training error. These strategies are known by *regularization*. Regularization in simple terms is the process of introducing additional information in order to solve an ill-posed problem or to prevent neural networks from overfitting by imposing a smoothness restriction in the model and consequently penalizing model complexity.

Some of these methodologies are characterized by adding extra constraints on the models, such as adding restrictions on the parameter values. Some add extra term in the cost function that can be thought as a constraint in the parameters. Others are designed to enforce specific forms of prior knowledge, which we will discuss in depth in the next chapters. All of them have in common the objective of improving the performance on the test set. An effective regularizer is the one that makes a profitable trade between reducing variance significantly while not drastically increasing the bias.

Ridge Regression

One of the simplest and most common kinds of parameter norm penalty is the L^2 parameter norm penalty commonly known as *weight decay* [34]. This type of regularization is also known by *Ridge regression* or even *Tikhonov regularization*. It is based on a regularization term $\Omega(\theta) = \frac{\lambda}{2} \|w\|_2^2$ that drives the weights closer to the origin. By doing so, the regularizer penalizes large weights, w , leading

to a reduction in the model variance. In other words, the weight decay prevents from overfitting by simplifying the model. Mathematically L^2 regularization can be represented by adding the penalty term to the cost function, $J(w, b)$,

$$J(\theta) = \frac{1}{m \times n_y} \sum_{i=1}^m \|\hat{y}^{(i)} - y^{(i)}\|^2 + \frac{\lambda}{2} \sum_{l=1}^L \|w^{[l]}\|_2^2, \quad (2.7)$$

where λ is the *regularization parameter* belonging to the set of hyperparameters, as we shall see later, that need to be tuned to get the best accuracy possible without overfitting.

2.4 Neural Network Training

So far, we have discussed Neural Networks as a general class of parametric nonlinear functions from a vector x of input variables to a vector y of output variables that approximate a desired function without explaining the approximation process. The approximation process is known by *training* or *learning*. Neural Network learning consists in the procedure of finding the parameter values that best fit the data in an iterative process of back and forth through the weights and biases of the layers of the Neural Network architecture. By simple means, learning consists in finding the parameters θ of a neural network that minimize a given cost function $J(\theta)$, given data.

Most training algorithms involve an iterative method for minimization of an error function, by adjusting the model parameter values in a sequence of steps. This procedure can be split into two stages. In the first stage, the derivatives of the error function, in this case the cost function, must be computed. This is done by back-propagation in such a way that the cost function is propagated backwards through the network. In the second stage, the derivatives are used to calculate the adjustments required for the weights and biases such that in the next step a decrease in the error function is verified.

In the majority of the available learning algorithms there is an inherent need for the gradient of the cost function with respect to the parameters, $\nabla_{\theta} J(\theta)$. Even though computing an analytical expression for the gradients is straightforward, evaluating numerically such an expression is computational expensive. For that purpose, a *back-propagation* algorithm [33, 34] was developed based on Automatic Differentiation (AD). In fact, AD is a family of techniques similar to but more general than back-propagation for efficiently and accurately evaluating derivatives of numeric functions, expressed as computer programs which are crucial for neural networks with, sometimes, millions of parameters.

2.4.1 Automatic Differentiation

The automatic calculation of gradients, automatic differentiation, completely simplifies the implementation of deep learning algorithms. Without automatic differentiation any change to the neural network model would require a new computation of the gradients manually. AD provides numerical values of derivatives (as opposed to derivative expressions) and it does so by using symbolic rules of differentiation while keeping track of derivative values as opposed to the resulting expressions [40]. Despite its relevance, automatic differentiation has been missing from the machine learning toolbox, a situation

gradually changing with its ongoing adoption under the names of *computational graphs* and differentiable programming, in several deep learning frameworks.

Automatic differentiation evaluates the derivatives of a function in such a way that exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations such as addition, subtraction, multiplication, division alongside many others and elementary functions like the exponentiation, logarithm or even trigonometric functions. Thereby, a neural network model can be represented by a set of elementary operations and functions in such a way that it forms what is called by a computational graph [33]. The computational graph language is in very simple terms a data structure that allows to efficiently apply the chain rule to compute gradients for all of the parameters of the neural network. Each node represents either a variable that can be a scalar, vector, matrix, tensor, or even a variable of another type or a function.

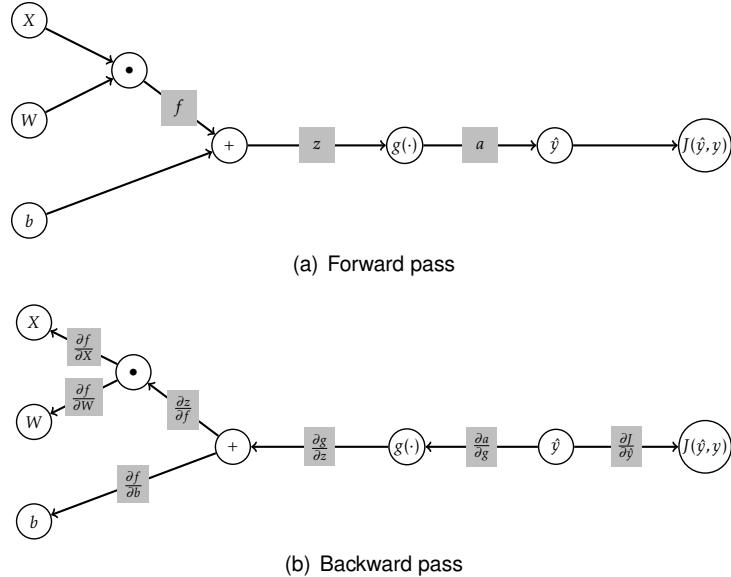


Figure 2.4: (a) Forward computational graph of a feedforward neural network with one hidden layer $\hat{y} = g(WX + b)$, where W and X are the weight and input matrix, respectively and b the bias vector. The activation function $g(\cdot)$ may be described by composing many operations together in a sub computational graph. Here, the value for the cost function is calculated. (b) The cost function adjoint is propagated backwards, providing the gradients with respect to the weights and inputs that are going to be used in the learning algorithm.

2.4.2 Back-Propagation

Modern complex neural network architectures can have up to millions of learnable parameters. Back-propagation is an efficient and accurate method to compute the gradients of such extensive and complex models based on a very simple principle that can be paralleled using multiple GPUs. From a computational point of view, training a neural network consists of two phases. First, there is a need for a forward pass to compute the value of the loss function. Second, a backward pass is required to compute the gradients of the learnable parameters [33] (see algorithm 1).

Many deep learning algorithms involve computing other derivatives, as we shall see further, either as part of the learning process, or to analyze the learned model. The back-propagation algorithm can

Algorithm 1: Back-Propagation Algorithm- Based on [33]. Firstly, with the forward pass starting from the input layer propagate forward through the network, computing the activities of the neurons at each layer and ending in computing the cost value. Secondly, compute the derivatives of the error function with respect to the output layer activities, propagating this error towards outer layers while calculating the gradients with respect to the parameters. Introducing the auxiliary quantity δ^l for the partial products as a measurement of the vector error in the l^{th} layer where each entry corresponds to each neuron in that layer. Even though, this demonstration uses only a single input example x as we shall see further practical applications should use a set of examples at once (minibatch).

Require: Network depth, L
Require: $W^{[l]}$, $l \in \{1, \dots, L\}$, the weight matrices of the model
Require: $b^{[l]}$, $l \in \{1, \dots, L\}$, the bias vectors of the model
Require: $x \in \mathbb{R}^{n_x}$, data input example represented as a column vector
Require: $y \in \mathbb{R}^{n_y}$, data output example represented as a column vector
Function Forward(W, b):
 Set $a^{(1)} = x^{(i)} \in \mathbb{R}^{n_x}$ // Activation of the first layer
 for $l = 2, \dots, L$ **do**
 $z^{[l]} = b^{[l]} + W^{[l]} a^{[l-1]}$ // Intermediate variable
 $a^{[l]} = g(z^{[l]})$ // Activation of the l^{th} layer
 Set $\hat{y} = a^{(L)}$ // Neural Network output
 $J = L(\hat{y}, y) + \Omega(\theta)$ // Regularized cost function
 return J

Function Backward(J):
 Compute $\nabla_{\hat{y}} J$ // Neural Network output error
 $\delta^{[L]} = \nabla_{\hat{y}} J \cdot g'(z^{[L]})$ // Output layer error
 for $l = L-1, \dots, 2$ **do**
 $\delta^{[l]} = \nabla J_{a^{[l]}} = (W^{[l+1]T} \delta^{[l+1]}) \odot g'(z^{[l]})$ // Vector error in the l^{th} layer
 $\nabla_{b^{[l]}} J = \delta^{[l]} + \lambda \nabla_{b^{[l]}} \Omega(\theta)$ // Gradients on biases
 $\nabla_{W^{[l]}} J = a^{[l-1]T} \delta^{[l]} + \lambda \nabla_{W^{[l]}} \Omega(\theta)$ // Gradients on weights
 return $\nabla_{b^{[L]}} J, \nabla_{W^{[L]}} J$

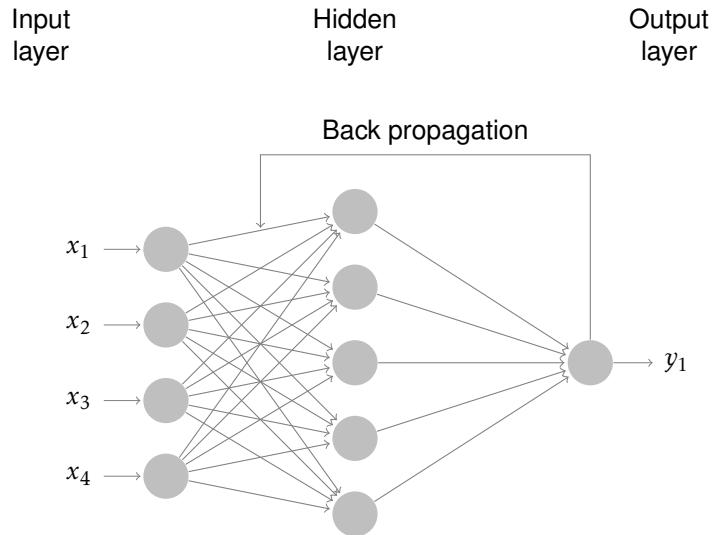


Figure 2.5: Back-propagation from a single hidden layer neural network perspective.

be also be applied to these variants, and is not restricted to computing the gradient of the cost function with respect to the parameters. The idea of computing derivatives by propagating information through a

network is very general, and can be used to compute values such as the Jacobian or even the Hessian of a function f approximated via a Neural Network.

2.4.3 Optimizers

We have discussed so far the basic concepts intrinsic to deep learning algorithms. First, we defined the concept of neurons that assembled into a layered organized structure form a neural network. Then we discussed the process to compute the gradients of the cost function with respect to the parameters in an efficient and accurate way, but there is one important component of building a neural network that approximates an arbitrary function that is as important as the others. This is where *optimizers* come in. They tie together the loss function and model parameters by updating the model in response to the output of the loss function with the objective of minimizing it. This optimization process is driven by the cost function, which in turn tells the optimizer when it's moving in the right or wrong direction.

Before we enter into particular learning algorithms, we should emphasize that most of these algorithms update the parameters based on an expected value of the cost function estimated using only a subset of the terms of the full cost function. These algorithms are usually called *minibatch*. Optimization algorithms that use the entire training set are called *batch* or deterministic gradient methods, since they process all of the training examples simultaneously in a large batch, like the very traditional Gradient Descent [34]. On the other hand, optimization algorithms that only use a single example at each iteration are called *stochastic*.

The need for minibatch algorithms is fully related with the fact that, on one hand, using batch methods implies computing the gradient over the entire dataset, averaging over a potential lot of information which not only leads to memory issues but also can lead to a bad loss function minimum, *saddle point*. On the other hand, using a single data example at a time induces noise that may be useful to escape from the saddle point. However, at the cost of inefficiency, since it requires evaluating every single example at once in a possibly very large dataset. Minibatch algorithms in turn seek for a trade-off between efficiency and accuracy leading to good generalization performance and significantly smaller memory footprint [41].

Adam - Adaptive Moment Estimation

Even though, stochastic gradient descent is very popular among machine learning research [33], learning with it can be often a slow procedure. For this purpose, several methods based on a momentum were developed. The momentum can be considered a variable that accumulates the gradients over iterations and that represents the direction and speed at which the parameters move through the parameter space. In this case, previous gradients will affect as well the current update.

The Adam algorithm [42] can be considered as an extension to stochastic gradient descent. It is computationally efficient and has low memory requirements. The algorithm updates exponential moving averages of the gradient, m_t , and the squared gradient, v_t , where the hyper-parameters β_1 and β_2 control the exponential decay rates of these moving averages. These moving averages are estimates of the 1st

moment, the mean, and the 2nd raw moment, the uncentered variance, of the gradients.

Algorithm 2: Adam - Retrieved from [42]. Here \mathbf{g}_t^2 indicates the element wise square $\mathbf{g}_t \odot \mathbf{g}_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ as a small constant used for numerical stabilization. All operations on vectors are element-wise. β_1^t and β_2^t are denoted by β_1 and β_2 to the power t .

```

Require:  $\alpha$ , step-size
Require:  $\beta_1, \beta_2 \in [0, 1]$ , exponential decay rates for the moment estimates
Require:  $\theta_0$ , initial parameter vector
Require:  $f(\theta)$ , Stochastic objective function with parameters  $\theta$ 

 $\mathbf{m}_0 \leftarrow 0$                                 // Initialize 1st moment vector
 $\mathbf{v}_0 \leftarrow 0$                                 // Initialize 2nd moment vector
 $t \leftarrow 0$                                     // Initialize timestep

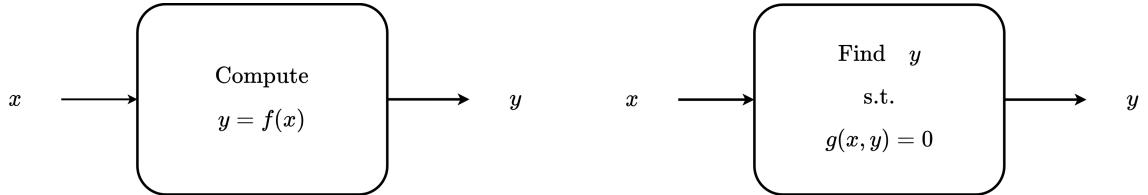
while  $\theta_t$  not converged do
     $t \leftarrow t + 1$ 
     $\mathbf{g}_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$            // Get gradients w.r.t. stochastic objective at  $t$ 
     $\hat{\mathbf{m}}_t \leftarrow \beta_1 \cdot \mathbf{m}_{t-1} + (1 - \beta_1) \cdot \mathbf{g}_t$       // Update biased first moment estimate
     $\hat{\mathbf{v}}_t \leftarrow \beta_2 \cdot \mathbf{v}_{t-1} + (1 - \beta_2) \cdot \mathbf{g}_t^2$     // Update biased second raw moment estimate
     $\hat{\mathbf{m}}_t \leftarrow \hat{\mathbf{m}}_t / (1 - \beta_1^t)$                       // Compute bias-corrected first moment estimate
     $\hat{\mathbf{v}}_t \leftarrow \hat{\mathbf{v}}_t / (1 - \beta_2^t)$                     // Compute bias-corrected second raw moment estimate
     $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{\mathbf{m}}_t / (\sqrt{\hat{\mathbf{v}}_t} + \epsilon)$  // Update parameters

```

2.5 Implicit Layers

Recently, deep learning methods started getting more complex as researchers started to add more functionalities to the layers. Deep learning layers are built largely like typical computer programs, where the code is directly written to generate the output of the layer as a function of its input.

An implicit layer², as we will use the term throughout this document, is a concept of layer in which instead of specifying how to compute the layer's output from the input, we specify the conditions that we want the layer's output to satisfy with respect to the input.



(a) **Explicit Layer:** the output of the layer is directly calculated by an affine transformation followed by a non-linear activation function.

(b) **Implicit Layer:** As an example a root finding algorithm can be used to compute the output of the layer.

Figure 2.6: Virtually all common layers are explicit in the sense that they provide a computation graph for computing the forward pass and then back propagate through the exact same graph. In contrast implicit layers, define a layer in such a way that the input-output pair has to satisfy some joint condition

Many examples can be found in the literature, specifically differentiable optimization layers [44, 45], where the output of the implicit layer is the solution to a constrained optimization problem based upon

²See [43] for some theoretical and algorithmic foundations.

previous layers, neural ordinary differential equations [18], in which the output of the implicit layer is the solution of an initial value problem computed with an ODE solver, without the need for back-propagating through all its intermediate operations. Further, fixed point iterations can also be used as an implicit layer (see figure (2.6)) giving rise to deep equilibrium models [46, 47] where the authors showcased the success of their implicit framework, for the task of sequence modeling. Implicit layers offer a set of desirable properties such as:

- **Powerful Representation:** Implicit Layers compactly represent complex operations such as solving differential equations, optimization problems or finding roots.
- **Memory Efficiency:** While one could back-propagate through the internal operations of an ODE solver or through the many iterations of a root finding algorithm, this is memory-expensive and can be avoided by means of the implicit function theorem.
- **Simplicity:** The implicit-based architectures are elegant and straightforward.
- **Abstraction:** Separating "what a layer should do" from "how to compute it", is an abstraction that has been extremely valuable in many other areas.

Implicit layers can simplify the notation of deep learning, and open up many new possibilities, in terms of novel architectures and algorithms, robustness analysis and design, interpretability, sparsity, and network architecture optimization [43].

Chapter 3

From Continuous to Discrete Mechanics

In this Chapter we start with a review of continuous Lagrangian mechanics. We derive the underlying equations of motion that govern any dynamical system with special emphasis in mechanical systems. Next a brief introduction to numerical integration is made and to some known numerical integrators. Finally, derivation of the discrete formulation of mechanics and respective integrators (Variational Integrators) conclude the Chapter.

3.1 Continuous Lagrangian Mechanics

We will now review the Lagrangian description of mechanics for mechanical systems as the base of this work. Any mechanical system can be represented by a set of generalized coordinates, $q_i \in \mathbb{R}^n$, that define completely the configuration of the structure relative to a reference configuration. This set of generalized coordinates in turn define a smooth configuration manifold Q . The Lagrangian mechanics define the Lagrangian, $\mathcal{L} : TQ \rightarrow \mathbb{R}$, as a function of the kinetic energy, T , and potential energy, V .

For a given time interval $[0, T]$, the *path space* is defined to be

$$\mathcal{C} = \mathcal{C}([0, T], Q) = \{q : [0, T] \rightarrow Q \mid q \text{ is a } C^2 \text{ curve}\}, \quad (3.1)$$

and the *action map* $\mathfrak{G} : \mathcal{C} \rightarrow \mathbb{R}$ to be

$$\mathfrak{G} = \int_0^T \mathcal{L}(q(t), \dot{q}(t)) dt. \quad (3.2)$$

The least-action principle states that the evolution of the generalized coordinates, $q(t)$, minimizes the action¹. With this in mind, proceeding to the computation of the variations of the action map using

¹To be more precise, the least-action principle states that the action should be extremized but not necessarily minimized. However, in practice, the action is almost always minimized (hence, the name least-action principle).

integration by parts and the condition $\delta q(0) = \delta q(T) = 0$,

$$\begin{aligned}\delta\mathfrak{G} &= \delta \int_0^T \mathcal{L}(q(t), \dot{q}(t)) dt = \int_0^T \left[\frac{\partial \mathcal{L}}{\partial q} \cdot \delta q + \frac{\partial \mathcal{L}}{\partial \dot{q}} \cdot \delta \dot{q} \right] dt \\ &= \int_0^T \left[\frac{\partial \mathcal{L}}{\partial q} + \frac{d}{dt} \left(\frac{\partial \mathcal{L}}{\partial \dot{q}} \right) \right] \delta q dt + \left[\frac{\partial \mathcal{L}}{\partial \dot{q}} \cdot \delta q \right]_0^T \\ &= \int_0^T \left[\frac{\partial \mathcal{L}}{\partial q} + \frac{d}{dt} \left(\frac{\partial \mathcal{L}}{\partial \dot{q}} \right) \right] dt,\end{aligned}\quad (3.3)$$

that can be summarized by Hamilton's principle, which seeks paths $q(t) \in \mathcal{C}(Q)$ which pass through $q(0)$ and $q(T)$ and also satisfy $\delta\mathfrak{G}(q) = 0$ leading the integrand of equation (3.3) to be zero for each t . Consequently we immediately get the well-known *Euler-Lagrange equations*,

$$\frac{\partial \mathcal{L}(q, \dot{q})}{\partial q} - \frac{d}{dt} \left(\frac{\partial \mathcal{L}(q, \dot{q})}{\partial \dot{q}} \right) = 0. \quad (3.4)$$

For the description of Lagrangian dynamics, we now extend the previous approach to dynamical systems that include external forcing resulting from dissipation, friction, loading and in particular control forces.

3.1.1 Forced Continuous Mechanics

Lagrangian systems with external forcing appear in many different contexts. From actuator control forces to dissipation and friction or even external loading on mechanical systems. With this in mind, in order to define control forces for Lagrangian systems, we shall now consider a control manifold $U \subset \mathbb{R}^m$ and define the *control path space* to be:

$$\mathcal{C}(U) = \mathcal{C}([0, T], U) = \{u : [0, T] \rightarrow U \mid u \in L^\infty\}, \quad (3.5)$$

where $u(t) \in U$ is called the control parameter and L^∞ represents the space of essentially bounded, measurable functions equipped with the essential supremum norm. Defining the Lagrangian force as a map $F_{\mathcal{L}} : TQ \times U \rightarrow T^*Q$, where T^*Q is the cotangent bundle defining the phase space, then any external force can be included [21].

The Lagrange-d'Alembert principle generalizes Hamilton's principle to Lagrangian systems with external forcing. The Lagrange-d'Alembert principle seeks paths $q \in \mathcal{C}(Q)$ satisfying

$$\delta \int_0^T \mathcal{L}(q(t), \dot{q}(t)) dt + \int_0^T F_{\mathcal{L}}(q(t), \dot{q}(t), u(t)) \delta dt = 0, \quad (3.6)$$

for all variations δq with $\delta q(0) = \delta q(T) = 0$. The second integrand may be thought as the virtual work acting on a mechanical system due to the force $f_{\mathcal{L}}$. Integrating by parts, one arrives at the *forced Euler-Lagrange equations*:

$$\frac{\partial \mathcal{L}(q, \dot{q})}{\partial q} - \frac{d}{dt} \left(\frac{\partial \mathcal{L}(q, \dot{q})}{\partial \dot{q}} \right) + F_{\mathcal{L}}(q(t), \dot{q}(t), u(t)) = 0. \quad (3.7)$$

These equations implicitly define a family of *forced Lagrangian flows* and *forced Lagrangian vector fields*

[21]. Equations (3.7) determine a system of n second-order differential equations. If we assume that the Lagrangian is *regular*, i.e., the $(n \times n)$ matrix $\left(\frac{\partial^2 \mathcal{L}}{\partial \dot{q} \partial \dot{q}}\right)$, is nonsingular, the local existence and uniqueness of solutions is guaranteed for any given initial condition by employing the implicit function theorem.

3.1.2 Lagrangian Dynamics for Mechanical Systems

Dynamical systems are systems with state $x \in \mathbb{R}^{2n}$ that evolve over time. The rate of change of the state \dot{x} is specified by some function $\dot{x} = f(x, u)$, and $u \in \mathbb{R}^m$ is the control input. Mechanical systems are a subset of dynamical systems describing the evolution of extended bodies in the physical world, and can describe many robotic systems of interest. The state, x , of a mechanical system is composed by a set of generalized coordinates, $q \in \mathbb{R}^n$, and their correspondent rate of changes, $\dot{q} \in \mathbb{R}^n$ called generalized velocities. The state is formed by both the generalized velocities and coordinates. Describing the equations of motion for mechanical systems has been extensively studied and several formalisms do exist in the literature [48]. The Lagrangian is chosen to be

$$\mathcal{L} = T - V. \quad (3.8)$$

The kinetic energy of mechanical systems takes the well-known quadratic form,

$$T(q, \dot{q}) = \frac{1}{2} \dot{q}^T H(q) \dot{q}, \quad (3.9)$$

where $H \in \mathbb{R}^{n \times n}$ is called *generalized inertia matrix* [48], which is symmetric and positive definite and hence makes the Lagrangian regular. The Lagrangian assumes then the form of

$$\mathcal{L} = T(q, \dot{q}) - V(q) = \frac{1}{2} \dot{q}^T H(q) \dot{q} - V(q). \quad (3.10)$$

Recall that this Lagrangian is not unique and every \mathcal{L} that yields the correct equations of motion is valid². The implicit forced Euler-Lagrange equations can be transformed into an explicit equation by taking into account Lagrangians of the form of (3.10). Applying calculus of variations over \mathcal{L} yields the Euler-Lagrange equation with non-conservative forces described by

$$H(q)\ddot{q} + C(q, \dot{q}) + g(q) = F(q, \dot{q}, u), \quad (3.11)$$

where $g(q) = \frac{\partial V(q)}{\partial q} \in \mathbb{R}^n$ is the *gravitational potential vector*, while $C \in \mathbb{R}^n$ is called the vector of *Coriolis* and *Centrifugal* forces given by:

$$C(q, \dot{q}) = \dot{H}(q)\dot{q} - \frac{1}{2} \left(\frac{\partial}{\partial q} (\dot{q}^T H(q) \dot{q}) \right)^T. \quad (3.12)$$

²In fact, $\mathcal{L}' = \alpha \mathcal{L} + \beta$, for any constant α and β , and $\mathcal{L}' = \mathcal{L} + \frac{d}{dt}f(q, t)$ lead to the same equations of motion by performing calculus of variations for the respective modified Lagrangians. We point this out as this will explain, in part, the results obtained.

Further, the Euler-Lagrange equation can be rearranged in order to obtain the forward model by solving equation (3.11) for \ddot{q} in terms of $H(q)$, $V(q)$ and $F(q, \dot{q}, u)$,

$$\ddot{q} = f(q, \dot{q}, u) = H^{-1}(F(q, \dot{q}, u) - C(q, \dot{q}) - g(q)), \quad (3.13)$$

i.e.,

$$f(q, \dot{q}, u) = H^{-1}\left(F(q, \dot{q}, u) - \dot{H}(q)\dot{q} + \frac{1}{2}\left(\frac{\partial}{\partial q}\left(\dot{q}^T H(q)\dot{q}\right)\right)^T - g(q)\right). \quad (3.14)$$

Note that the positive definiteness of the inertia matrix ensures invertibility and with that equation (3.14) can explicitly be used to compute the generalized accelerations. Record that equation (3.14) defines an ordinary differential equation (ODE) that can describe any multi-body mechanical system with holonomic constraints [48].

3.2 Numerical Integration of Differential Equations

In many engineering applications, the solution of ODEs such as the one in equation (3.14) is required. Typical examples are present in the field of robotics. Specifically within model-based control where the system's continuous equations of motion need to be transferred into discrete counterparts in a process called time discretization. This process is accomplished by means of numerical integrators. Defining the space state by $\dot{x} = [\dot{q}, \ddot{q}]$ and letting an initial value problem (IVP) be specified as follows:

$$\frac{dx}{dt} = f(x, t), \quad x(t_0) = x_0, \quad (3.15)$$

a numerical integrator (or ODE solver) approximates the true solution x of an ODE of the form $\dot{x} = f(x(t))$ at discrete time steps t_0, t_1, \dots, t_T . The simplest integrator, **Euler's Method**, starts from the initial state x_0 at time t_0 and proceeds to the estimation of the function $f(t)$ at uniformly spaced time points $t_n = t_0 + h$ with the recursive expression:

$$x_{t+h} = x_t + hf(x_t, t). \quad (3.16)$$

In fact, the Euler method is the simplest numerical integrator of a broad family of ODE solvers known as Runge–Kutta methods [49]. However, Euler's method can easily lead to unstable solutions for larger time-steps as in practice consists of approximating the original function by drawing a sequence of curve segments of the function. This can be improved by using a half step and sample the derivative there instead. This is accomplished by an explicit second-order Runge-Kutta 2nd Order (RK2) known by **Midpoint Rule**. The midpoint method has the form as follows:

$$x_{t+h} = x_t + f\left(x_t + \frac{h}{2}f(x_t, t), t + \frac{h}{2}\right). \quad (3.17)$$

Note that the midpoint rule still relies on Euler's method to determinate the solution at the mid-point of the interval, leading to an extra order of accuracy with respect to Euler's method.

Higher order numerical integrators can also be obtained by means of more collocation points. Mak-

ing use of four collocation points leads to the most widely known member of the Runge–Kutta family, generally referred to as **fourth-order Runge–Kutta** (RK4). For instance, the RK4 method is a fourth-order method that can be formulated by evaluating the integrand $f(x, u)$, four times per step (one for each collocation point).

Given the system's generalized coordinates q_t and generalized velocity \dot{q}_t at some time t , we predict $q_{t'}$ and $\dot{q}_{t'}$ at some time $t' = t + \Delta t$. Defining the time derivative of the state space by $[\dot{q}, \ddot{q}_t]^T$ the RK4 takes the following form [4]:

$$\begin{aligned} k_1 &= \Delta t \cdot f(x_t, u), \\ k_2 &= \Delta t \cdot f(x_t + k_1/2, u), \\ k_3 &= \Delta t \cdot f(x_t + k_2/2, u), \\ k_4 &= \Delta t \cdot f(x_t + k_3, u), \\ x_{t+\Delta t} &= x_t + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \end{aligned} \tag{3.18}$$

The development of efficient and accurate numerical integrators is object of considerable research [17, 19, 50, 51].

3.3 Geometric Numerical Integration

Geometric integration is a class of numerical methods that exploits the intrinsic geometric structure (within a round-off error) present within the equations of motion of dynamical systems [51]. By doing so, these methods provide high quality numerical integration that respects, in part, the original continuous structure of the equations of motion. Many of the preserved geometric properties are of crucial importance in many physical applications: preservation of energy, momentum, angular momentum, phase-space volume, symmetries, time-reversal symmetry, symplectic structure and dissipation are examples of those [51]. As a consequence, the numerical solutions computed by these integrators are not only quantitatively accurate but they are also qualitatively superior to those classical integrators such as the Runge-Kutta family of integrators.

A brief introduction of the field of geometric integration can be found on [17]. In fact, Ge and Marsden [52] state that a geometric integration can either preserve energy and momentum, or symplectic structure and momentum, but not all three at the same time. For this purpose, the symplectic-momentum and energy-momentum terms are often found in the literature to designate the integrators. Even though symplectic-momentum integrators do not conserve energy exactly, they have been shown to exhibit good long-time energy behavior [51].

3.3.1 Symplectomorphisms

Symplectomorphisms [53] are one of the most important features of the time evolution of Hamilton's equations. Symplectic maps represent a transformation of the space where all possible states of a given system are represented (*phase space*), with each possible state corresponding to one unique point in

the phase space. Further, this transformation is volume preserving, i.e., a symplectic transformation of the phase flow conserves the symplectic two-form

$$d\mathbf{q} \wedge d\mathbf{p} \equiv \sum_{i=1}^N (dq_i \wedge dp_i), \quad (3.19)$$

where \wedge denotes the wedge product between two differential forms. The rules of wedge products can be found in [54]. $p = \partial\mathcal{L}/\partial\dot{q}$ is the momentum, that for Lagrangian systems of the form of (3.10) is simply $p = H(q)\dot{q}$. A numerical scheme is said to be a symplectic integrator if it conserves this two-form (figure 3.1), which in the two-dimensional case, can be understood as the area element of the surface.

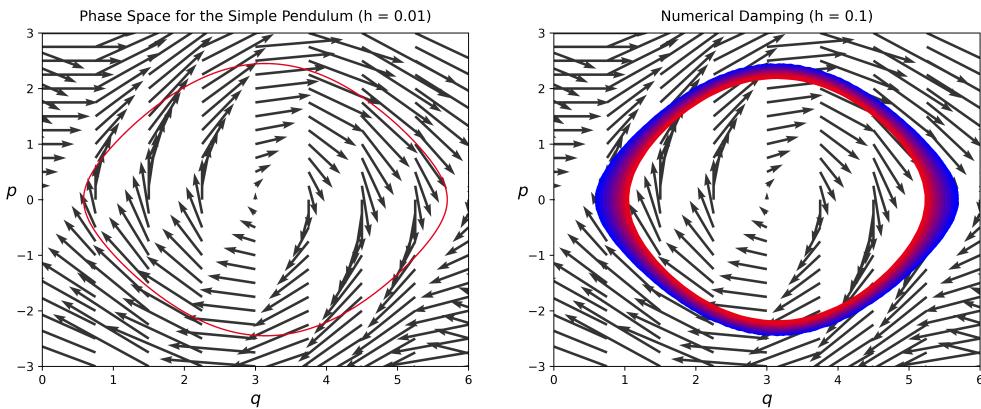


Figure 3.1: Phase Space for the Simple Pendulum obtained from the RK4 integrator. Both plots were integrated for 10000 time-steps. Note that for small time-steps the RK4 method preserves the symplectic form (left) while for larger time-steps there is a relevant dissipation of the circle area due to numerical damping (right).

3.3.2 Discrete Mechanics and Variational Integrators

In robotics or in any other engineering application, numerical integrators for mechanical systems are generally derived by discretizing differential equations, which typically begin with continuous-time formulation of the dynamics that are numerically integrated to yield a discrete-time approximation of the continuous-time dynamics. Despite good local behavior, these traditional integrators do not account for the inherent geometric structure of the governing continuous-time equations [19]. Therefore, they inject numerical dissipation and do not preserve invariants of the system, which for long time simulations may lead to a destabilization of the integration but also compromise the model energy evolution [19, 51].

In contrast, the theory of discrete variational mechanics describes a variational approach to discrete mechanics and mechanical integrators. The idea behind the variational approach is to discretize the variational formulation of the system dynamics, rather than the differential equations [19, 51]. These dynamics may be either conservative or dissipative and forced. This approach is known in the literature by Discrete Mechanics and the resulting integrators by Variational Integrators (VI).

Advantages of variational integrators. The variational approach around discrete mechanics automatically ensures a number of desirable properties. In fact, VI's have the property of conserving momen-

tum and a symplectic form [51]. These integrators are usually placed on a class of geometric integrators known by symplectic-momentum integrators [51] as opposed to energy-momentum integrators. Energy cannot, in general, be exactly preserved by symplectic integration [55]. However, VI's exhibit bounded energy behavior, achieving near energy conservation. These properties make Variational Integrators an appealing choice for long-term simulation of physical systems which are either conservative or near-conservative. Along with these properties, VI's handle elegantly holonomic constraints, external forces, impacts, and non-smooth phenomenon [19, 56, 57].

Discrete Variational Principles have been used in the context of Optimal Control and Discrete Mechanics for space mission design [21]. Discrete Variational principles have also been used for trajectory optimization [58, 59] or even parameter estimation [60, 61].

3.3.3 Discrete Mechanics

In this section, we present the discrete analog of continuous mechanics. Full derivation can be seen in [19, 21]. In an alternative approach of numerical integration one can instead discretize the action integral forcing the errors to respect the conservation of properties, which, in turn, leads to a discrete derivation of the equations that govern the system's dynamics.

For the derivation of the continuous Euler-Lagrange equations (3.4), (3.7) we considered a mechanical system with configuration manifold Q , velocity phase space TQ and the Lagrangian as a map $\mathcal{L} : TQ \rightarrow \mathbb{R}$. The idea behind the discrete level is to replace TQ with the manifold $Q \times Q$ and account for two nearby points as being the discrete analogue of the velocity vector. In the discrete variational mechanics paradigm, we now define the configuration manifold as Q and the discrete state space as $Q \times Q$. This means that the path $q : [0, T] \rightarrow Q$ is replaced by a discrete path $q_d : \{t_k\}_{k=0}^N \rightarrow Q$, where $q_k = q_d(kh)$ is an approximation to $q(kh)$. Even though, here, the type of information that characterizes a given dynamical system changed the corresponding amount of information remains the same.

A *discrete Lagrangian*, \mathcal{L}_d , is defined as a function $\mathcal{L}_d : Q \times Q \rightarrow \mathbb{R}$ that approximates the action integral along the exact solution curve segment q between q_k and q_{k+1}

$$\mathcal{L}_d(q_k, q_{k+1}) \approx \int_{t_k}^{t_{k+1}} \mathcal{L}(q(t), \dot{q}(t)) dt. \quad (3.20)$$

Here, it is then required to introduce a time-step, that relates the discrete and continuous mechanics, $h \in \mathbb{R}$, as the time difference between two adjacent points, i.e., $h = t_{k+1} - t_k$.

Considering the time grid, $\{t_k = kh \mid k = 0, \dots, N\} \subset \mathbb{R}$, $Nh = T$, it is possible to define the *discrete path space* as follows:

$$\mathcal{C}_d = \mathcal{C}_d(\{t_k\}_{k=0}^N) = \{q_d : \{t_k\}_{k=0}^N \rightarrow Q\}. \quad (3.21)$$

The *discrete action map*, $\mathfrak{G}_d : \mathcal{C}_d \rightarrow \mathbb{R}$, along this sequence is now calculated by summing the individual discrete Lagrangians on each adjacent-time pair:

$$\mathfrak{G}_d(q_d) = \sum_{k=0}^{N-1} \mathcal{L}_d(q_k, q_{k+1}). \quad (3.22)$$

Similar to the continuous case, the discrete Hamiltonian's principle, instead, seeks discrete curves, $q_{k=0}^N$, that satisfy zero variation of the discrete action for arbitrary discrete variations, δq_d :

$$\delta \mathfrak{G}_d(q_d) = \sum_{k=0}^{N-1} [D_1 \mathcal{L}_d(q_k, q_{k+1}).\delta q_k + D_2 \mathcal{L}_d(q_k, q_{k+1}).\delta q_{k+1}] = 0. \quad (3.23)$$

The slot derivative D_i represents the derivative of the function \mathcal{L} with respect to the i^{th} argument. Rearranging equation (3.23),

$$\delta \mathfrak{G}_d(q_d) = \sum_{k=1}^{N-1} [D_1 \mathcal{L}_d(q_k, q_{k+1}).\delta q_k + D_2 \mathcal{L}_d(q_{k-1}, q_k).\delta q_k] + D_1 \mathcal{L}_d(q_0, q_1).\delta q_0 + D_2 \mathcal{L}_d(q_{N-1}, q_N).\delta q_N = 0, \quad (3.24)$$

and requiring vanishing variations at the end points, $\delta q_0 = \delta q_N = 0$, then we obtain the discrete equivalent of equation (3.4), known by Discrete Euler-Lagrangian Equations (DEL):

$$D_1 \mathcal{L}_d(q_k, q_{k+1}) + D_2 \mathcal{L}_d(q_{k-1}, q_k) = 0. \quad (3.25)$$

These equations implicitly define the discrete Lagrangian map $F_{L_d}: Q \times Q \rightarrow Q \times Q$ mapping (q_{k-1}, q_k) to (q_k, q_{k+1}) . A sequence of discrete points $\{q_k\}$ is said to be a solution of the discrete Lagrangian map if they satisfy the discrete Euler-Lagrangian equations for all $k = 1, \dots, N - 1$.

Discrete Lagrangian maps are symplectic [21]. The discrete Lagrangian one-forms $\Theta_{L_d}^+$ and $\Theta_{L_d}^-$ are called *discrete Lagrangian one-forms* and in coordinates are

$$\Theta_{L_d}^+(q_0, q_1) = D_2 \mathcal{L}_d(q_0, q_1) dq_1 = \frac{\partial \mathcal{L}_d}{\partial q_1^i} dq_1^i, \quad (3.26)$$

$$\Theta_{L_d}^-(q_0, q_1) = D_1 \mathcal{L}_d(q_0, q_1) dq_0 = \frac{\partial \mathcal{L}_d}{\partial q_0^i} dq_0^i. \quad (3.27)$$

The discrete Lagrangian maps inherits symplecticity from the continuous Lagrangian flows. This means that the *discrete Lagrangian symplectic form* $\Omega_{L_d} = d\Theta_{L_d}^+ = d\Theta_{L_d}^-$ with coordinate expression

$$\Omega_{L_d}(q_0, q_1) = \frac{\partial^2 \mathcal{L}_d}{\partial q_0^i \partial q_1^j} dq_0^i \wedge dq_1^j, \quad (3.28)$$

is preserved under the discrete Lagrangian map (equation 3.25). By preserving the same two form on state space as the true system, integrators derived from the discrete Lagrangian formulation guarantee excellent energy behaviour, even for long term simulations [19].

The discrete Lagrangian map is momentum-preserving [19]. For mechanical systems it is often common to specify the initial conditions as a position and a momentum, p_k , instead of two positions,

$$p_{k,k+1}^+ = p^+(q_k, q_{k+1}) = \mathbb{F}^+ \mathcal{L}_d(q_k, q_{k+1}), \quad (3.29)$$

$$p_{k,k+1}^- = p^-(q_k, q_{k+1}) = \mathbb{F}^- \mathcal{L}_d(q_k, q_{k+1}), \quad (3.30)$$

for the momentum at the two endpoints of each interval $[t_k, t_{k+1}]$. The discrete Euler-Lagrange equations can be written as

$$D_2 \mathcal{L}_d(q_{k-1}, q_k) = -D_1 \mathcal{L}_d(q_k, q_{k+1}), \quad (3.31)$$

or even

$$\mathbb{F}^+ \mathcal{L}_d(q_{k-1}, q_k) = \mathbb{F}^- \mathcal{L}_d(q_k, q_{k+1}), \quad (3.32)$$

which is simply

$$p_{k-1,k}^+ = p_{k,k+1}^-. \quad (3.33)$$

Thus, the discrete Lagrangian map defined by the discrete Euler-Lagrange equations enforces momentum preservation at time k . This means that the momentum at time k should be the same when evaluated from the lower interval $[k-1, k]$ or the upper interval $[k, k+1]$. In fact, the discrete Lagrangian map has as conserved quantity a Lagrangian momentum map. One refers [19] for further details and proofs.

3.3.4 Forced Discrete Mechanics

Discrete Lagrangian Control Forces. Similar to the replacement of the path space by a discrete one, now the control path must be replaced by a *discrete control path* [21]. For this purpose, let's consider a refined grid $\Delta\tilde{t}$, that is generated via a set of control points $0 \leq c_1 \leq \dots \leq c_s \leq 1$ as $\Delta\tilde{t} = \{\tilde{t}_{kl} = t_k + c_l h \mid k = 1, \dots, N-1, l = 1, \dots, s\}$ which allows us to define the discrete control path by:

$$C_d(U) = C_d(\Delta\tilde{t}, U) = \{u_d : \Delta\tilde{t} \rightarrow U\}.$$

Let the intermediate control samples u_k on $[t_k, t_{k+1}]$ as $u_k = (u_{k1}, \dots, u_{ks}) \in U^s$ be the values of the controls or any other force that guides the system from $q_k = q_d(t_k)$ to $q_{k+1} = q_d(t_{k+1})$, where $u_{kl} = u_d(\tilde{t}_{kl})$ for $l \in \{1, \dots, s\}$.

With this in mind, the continuous force, $F(q, \dot{q}, u) : TQ \times U \rightarrow T^*Q$, is approximated by a left and right discrete force, $f_d^+, f_d^- : Q \times Q \times U^s \rightarrow T^*Q$, that can be written in coordinates as:

$$f_d^+(q_k, q_{k+1}, u_k) = (q_{k+1}, f_d^+(q_k, q_{k+1}, u_k)), \quad (3.34)$$

$$f_d^-(q_k, q_{k+1}, u_k) = (q_k, f_d^-(q_k, q_{k+1}, u_k)). \quad (3.35)$$

Combining the two discrete forces into a single form $f_d : Q \times Q \times U^s \rightarrow T^*(Q \times Q)$ and assuming u_k fixed:

$$f_d(q_k, q_{k+1}).(\delta q_k, \delta q_{k+1}) = f_d^+(q_k, q_{k+1}).\delta q_{k+1} + f_d^-(q_k, q_{k+1}).\delta q_k \quad (3.36)$$

We can think of the last term as an approximation of the continuous virtual work done by the external forcing during a time-step h , i.e.,

$$f_d^+(q_k, q_{k+1}, u_k).\delta q_{k+1} + f_d^-(q_k, q_{k+1}, u_k).\delta q_k \approx \int_{t_k}^{t_{k+1}} F_L(q(t), \dot{q}(t), u(t)) \delta q dt. \quad (3.37)$$

Similar with discrete Lagrangians, the discrete forces will also depend on the time-step h . To fully unravel what the left and right discrete forces represent, one can interpret from figure 3.2 the left discrete force, f_{k-1}^+ , as the contribution of the continuous force acting during the time interval $[t_{k-1}, t_k]$ on the node correspondent to the generalized coordinate q_k . Similarly, the right discrete force f_k^- maps the contribution of the continuous force acting during the time interval $[t_k, t_{k+1}]$ on q_k .

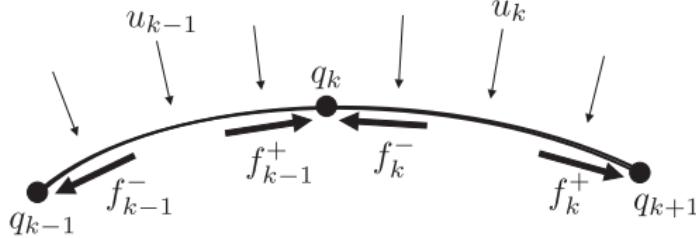


Figure 3.2: Left and right discrete forces. Retrieved from [21].

Discrete Lagrange-d'Alembert principle. Defined the discrete forces, similarly to the continuous Lagrangian mechanics with external forces, it is now possible to modify the discrete Hamiltonian's principle valid for conservative systems to the discrete Lagrange-d'Alembert principle, which seeks discrete curves $\{q_k\}_{k=0}^N$ satisfying

$$\delta \sum_{k=0}^{N-1} \mathcal{L}_d(q_k, q_{k+1}) + \sum_{k=0}^{N-1} [f_d^+(q_k, q_{k+1}).\delta q_{k+1} + f_d^-(q_k, q_{k+1}, u_k).\delta q_k] = 0, \quad (3.38)$$

for all variations $\{\delta q_k\}_{k=0}^N$ vanishing at the end points, $\delta q_0 = \delta q_N = 0$. This leads to the forced Discrete Euler-Lagrangian Equations:

$$D_1 \mathcal{L}_d(q_k, q_{k+1}) + D_2 \mathcal{L}_d(q_{k-1}, q_k) + f_d^+(q_{k-1}, q_k, u_{k-1}) + f_d^-(q_k, q_{k+1}, u_k) = 0. \quad (3.39)$$

Unlike the unforced case, the symplectic form will not be preserved in the presence of forcing [19].

The forced discrete Lagrangian map preserves the momentum evolution [19]. Again, the forced discrete Euler-Lagrange equations can be formulated as

$$D_2 \mathcal{L}_d(q_{k-1}, q_k) + f^+(q_{k-1}, q_k, u_k) = -D_1 \mathcal{L}_d(q_k, q_{k+1}) - f^-(q_k, q_{k+1}, u_k) \quad (3.40)$$

that automatically gives the definitions for the momentums

$$p_k = D_2 \mathcal{L}_d(q_{k-1}, q_k) + f^+(q_{k-1}, q_k, u_k) \quad (3.41)$$

$$p_{k-1} = -D_1 \mathcal{L}_d(q_k, q_{k+1}) - f^-(q_k, q_{k+1}, u_k) \quad (3.42)$$

which is the same as the unforced case but with the discrete forces added. Consequently, the evolution of the momentum is preserved. This is a direct consequence of the Discrete forced Noether's theorem and we leave [19] for the interested reader. Since the Lagrangian momentum map is preserved, even with external forcing, variational integrators preserve the energy rate very accurately.

3.3.5 Variational Integrators

Variational Integrators are a class of algorithms with the goal of solving the implicit DEL equation:

$$g(q_{k+1}) = D_1 \mathcal{L}_d(q_k, q_{k+1}) + D_2 \mathcal{L}_d(q_{k-1}, q_k) + f_d^+(q_{k-1}, q_k, u_{k-1}) + f_d^-(q_k, q_{k+1}, u_k) = 0 \quad (3.43)$$

As opposed to the numerical integration procedure applied to the Euler-Lagrange equations, in the context of variational integrators, a root finding algorithm such as Newton's shall be used. In fact, starting with two previous configurations, q_{k-1} and q_k , and a sequence of adjacent control inputs, u_{k-1} , u_k and u_{k+1} , the function defined by equation (3.43) can be solved using an iterative method to obtain the configuration, q_{k+1} , at the next time step. Additionally, the Implicit Function Theorem guarantees that such a function exists [19] provided that the derivative of $g(q_{k+1})$ w.r.t. q_{k+1} ,

$$J_g(q_{k+1}) = D_2 D_1 \mathcal{L}_d(q_k, q_{k+1}) + D_2 f_d^-(q_k, q_{k+1}, u_k) \quad (3.44)$$

is non-singular. This means that the derivative of the DEL equation is invertible and hence there is a unique solution for q_{k+1} . This is automatically guaranteed by the non-degeneracy of Lagrangians of the form of (3.10), for the case where the forces do not depend on the configuration, where the inertia matrix is symmetric positive definite. See Annex A.1 for a detailed explanation.

Chapter 4

Physics-Guided Deep Learning

In this chapter we discuss key topics that will be used throughout this work. We begin by reviewing the combination of Machine Learning concepts with inductive biases that are the foundations of our work. Further, this chapter will also consist on the establishment of the baselines used throughout this work.

4.1 Incorporating Lagrangian Mechanics into Deep Learning

Incorporating a priori knowledge of known physical laws into a learning algorithm helps reducing the necessary complexity of the learner and generally improves performance [5]. For instance, using equation (3.14) as a physical prior, represented by Lagrangian mechanics directly into the learning framework works as an implicit regularizer, resulting in an end-to-end training, and provides robust models capable of extrapolating to scenarios beyond the regime where this model was trained, while simultaneously ensuring physical plausibility. Authors in [4–6], proposed to combine equation (3.14) with deep learning techniques. Rather than using a black-box neural network to map the state space to the accelerations directly, here feed-forward Neural Networks are, instead, used for parameterizing the unknown potential energy and inertia functions, $V(q; \psi)$ and $H(q; \beta)$, with ψ and β being the set of learning parameters. Note that in equation (3.14), the inversion of the inertia matrix is necessary. As it is symmetric positive definite, this matrix is always invertible, ensuring then that the dynamics can be used as forward model and hence ensuring solution uniqueness. Ensuring the symmetry and positive definiteness of H is then crucial as this constraint enforces positive kinetic energy for all non-zero velocities.

4.1.1 Ensuring Symmetry and Positive Definiteness of the Inertia Matrix

The methodology for enforcing the positive definiteness of the inertia matrix using a neural network with parameters $\beta \cup \psi = \theta$ is based in [5]. By using trivial algebra properties one can easily note that the

symmetric inertia matrix, $H(q)$, is positive definite if

$$\dot{q}^T H(q) \dot{q} > 0 \quad \forall \dot{q} \in \mathbb{R}_0^n. \quad (4.1)$$

While we cannot guarantee that the learned $H(q)$ is positive definite by learning its elements directly from a feed forward network we can, instead, make use of the Cholesky decomposition of a symmetric positive definite matrix (see Figure 4.1). In fact, any symmetric positive definite matrix can be represented as the product of a lower-triangular matrix, L , with real and positive diagonal entries. Assuming $\hat{L}(q; \beta)$ is parameterized by a neural network, which predicts its $\frac{N^2+N}{2}$ elements of its Cholesky factor, with N being the number of degrees of freedom of the mechanical system, then the Cholesky decomposition of this matrix is a decomposition of the form

$$\hat{H}(q; \beta) = \hat{L}(q; \beta) \hat{L}(q; \beta)^T. \quad (4.2)$$

The symmetry and the positive semi-definiteness of \hat{H} are then automatically guaranteed, while simultaneously reducing the number of parameters needed to obtain the predicted inertia matrix. The positive definiteness is obtained by enforcing the diagonal elements of L , to be positive. This restriction is easily assured by using non-linearities with non-negative range, such as the ReLu or Softplus. Further, a small positive scalar, ϵ , is added to the diagonal elements of $\hat{H}(q; \beta)$ to ensure positiveness.

The neural network outputs a vector of predictions, of which the first N elements are used as the diagonal of $\hat{L}(q; \beta)$, l_d , and the remaining $\frac{N^2-N}{2}$ are used for the off-diagonal elements, l_o .

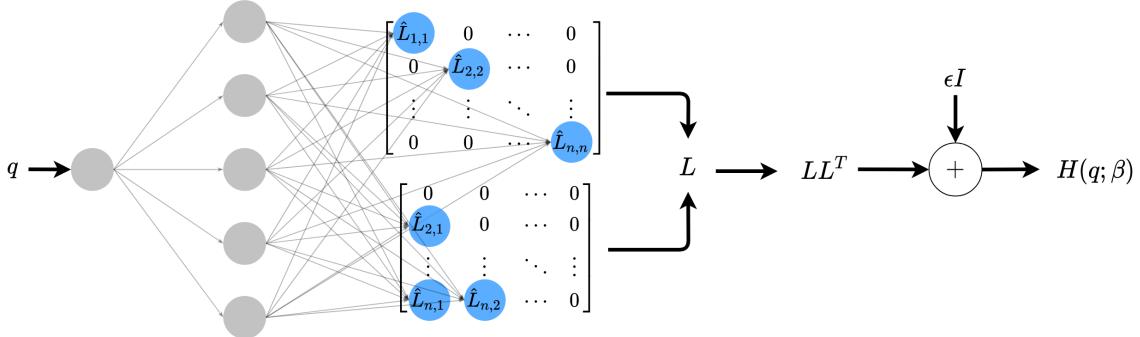


Figure 4.1: Computational graph for the Cholesky decomposition of the model's inertia matrix. Throughout this work we will be using $\epsilon = 1e^{-9}$. I is the identity matrix.

Note that, no matter the random weight initialization used, H , being implicitly enforced to be symmetric positive-definite, will always be invertible.

4.1.2 Deep Energy-Based Modeling of Mechanical Systems

Given the constraint enforcement of Section 4.1.1, one is now able to develop a neural network based architecture to parameterize multi-body rigid dynamics of mechanical systems of the form of equation (3.14). Further, by parameterizing the inertia matrix and the potential energy by neural networks we are, in fact, learning the system's energy (hence the term "Deep Energy-Based Modeling"). Throughout this

work we opt to share parameters between L and V as both not only rely on the same inputs but also less parameters are needed¹. Note, from equation (3.14), that we are required to take gradients of $H(q; \beta)$ as well as $V(q; \psi)$, with respect to q , in order to compute the system's acceleration, and later Hessians of the Lagrangian, we require the non-linearities in the neural network to be at least twice-differentiable. All of the activation functions are then hyperbolic tangents (\tanh) (Table 2.1) except for the hidden-to-output activation corresponding to the diagonal elements of L , where we make use of Softplus² to enforce non-negativity. The remaining hidden-to-output activations are set by a linear activation. See figure 4.2 for better understanding.

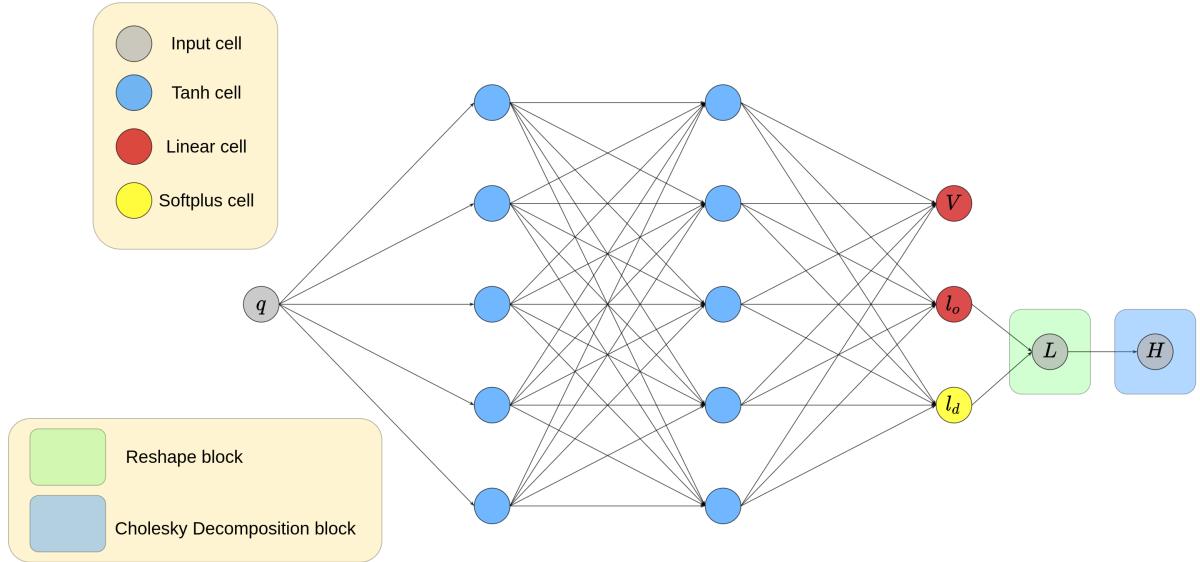


Figure 4.2: Gray-Box architecture for modeling the energy of mechanical systems.

4.1.3 Discontinuity free Learning

In general, rigid motions consist of rotations and translations. Rotations involve angular coordinates and so discontinuities. In fact, by considering observations of angular variables in the range of $[-\pi, \pi]$ for $\phi = \pi - \epsilon$, small changes in orientation might produce large changes in the value of ϕ (i.e., a rotation by ϵ causes ϕ to swing around $-\pi$ and π). This might be problematic when enforcing constraints implicitly such as Lagrangian dynamics, specially while learning the next states based on the present state when discontinuities are present. On the other side, if we treat each angular coordinate as a variable in \mathbb{R}^1 (line), the learning algorithm will not be able to infer that, for instance, 0 and 6π represents the same angle. Further, for long training trajectories large angle magnitudes will induce high variance to the model and make the training unstable.

¹In the literature one may find one different neural network for each task L and V (see [4, 6, 15]), while it is also possible to find what can be considered as a multi-headed neural network, where the first layers share parameters, diverging only on the activation function of the output values in the last layer [5]. We opted for the latter as it showed good results.

²The reason why we discard ReLu is related with the fact that its first derivative is 0 for negative inputs (see Table 2.1).

Learning from Embedded Angle Data on \mathbb{T}^m . Assuming observations of the form of $(q, \dot{q}, u)_{t0, \dots, tn}$, where $q \in \mathbb{R}^m$ are angular coordinates, then q resides on the manifold \mathbb{T}^m (m -torus) given by $\mathbb{T}^m = \mathbb{S}^1 \times \dots \times \mathbb{S}^m$ with \mathbb{S} being the circle manifold. From a data-driven perspective, the data that respects this geometry is simply a two dimensional embedding $(\cos q, \sin q)$. In particular, we will embed each angle $q \in [-\infty, \infty]$ into a new set of coordinates $s(q) = (c_q, s_q) = (\cos q, \sin q)$ by a transformation s , that represents the embedding, before passing into the network as [15]. With this in mind the neural network architecture of figure 4.2 assumes then the form of $H(c_q, s_q; \beta)$ and $V(c_q, s_q; \psi)$. Note that if gradient operations are necessary we still perform them on q by means of automatic differentiation.

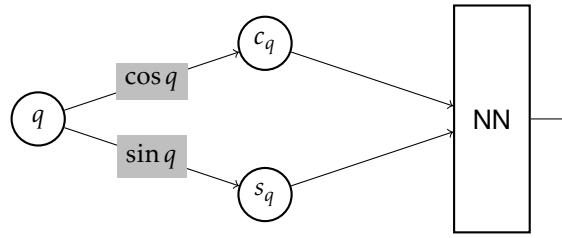


Figure 4.3: Computational Graph of the angle embedding used to feed the Neural Network block. Note that adding the embedding to the computational graph does not change the nature of the underlying equations. In fact, we just need the gradients to be computed with respect to the neural network inputs, (c_q, s_q) and by the chain rule calculate them with respect to q .

Learning from Hybrid Spaces $\mathbb{R}^n \times \mathbb{T}^m$. In most of physical systems, translational and rotational coordinates coexist. For instance, in robotics, mechanical systems are usually modelled as a set of interconnected rigid bodies interacting under their mutual potential. This mutual potential depends on both the position and the attitude of the bodies by means of rotational and translational coordinates. These coordinates form together the coupled dynamics of what is known by the full body problem, with generalized coordinates residing on $\mathbb{R}^n \times \mathbb{T}^m$. Here, we assume generalized coordinates $q = (r, \phi) \in \mathbb{R}^n \times \mathbb{R}^m$. Following the same line of reasoning the embedding will now be of the form $s(q) = (r, \cos \phi, \sin \phi) \in \mathbb{R}^n \times \mathbb{T}^m$ from observations $(r, \phi, \dot{r}, \dot{\phi})$ with neural network outputs $H(r, c_\phi, s_\phi; \beta)$ and $V(r, c_\phi, s_\phi; \psi)$.

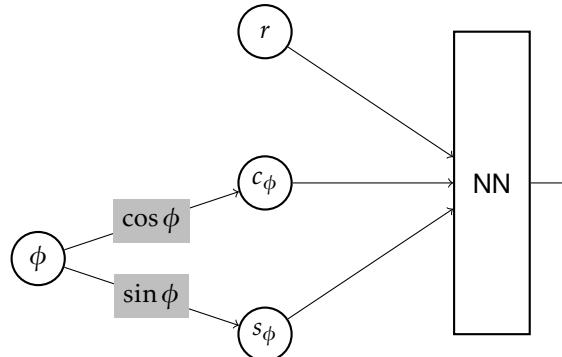


Figure 4.4: Computational Graph of the angle embedding used to feed the Neural Network block. Similarly with the previous case adding the embedding to the computational graph does not change the nature of the underlying equations.

4.1.4 Gradient Based End-To-End Learning of Lagrangian Mechanics

With the derivations given above is now straightforward to incorporate the structure introduced by the Euler-Lagrange ODE into the learning problem and learn the parameters in a gradient based end-to-end fashion by exploiting recent automatic differentiation frameworks. This approach was first introduced in [5] and it consists in learning the forward model defined by the second order ordinary differential equation (ODE) and with the embedding proposed

$$\hat{f}(q, \dot{q}, u; \theta) = [\hat{H}(s(q))]^{-1} \underbrace{\left(F(q, \dot{q}, u) - \dot{\hat{H}}(s(q))\dot{q} + \frac{1}{2} \left(\frac{\partial}{\partial q} (\dot{q}^T \hat{H}(s(q)) \dot{q}) \right)^T + \frac{\partial \hat{V}(s(q))}{\partial q} \right)}_{C(q, \dot{q})}, \quad (4.3)$$

with \hat{H} and \hat{V} approximated by a MLP, where \hat{H} is defined by equation (4.2).

Second order derivatives. First, we need to get rid of the temporal derivatives present in the Centrifugal and Coriolis term. Note that the generalized coordinates are a function of time $q(t)$, and hence the temporal derivative is trivial to obtain by means of the chain rule. The Centrifugal and Coriolis term can then be written as follows

$$C(q, \dot{q}; \beta) = \frac{\partial \hat{H}(s(q); \beta)}{\partial q} \dot{q} \dot{q} - \frac{1}{2} \left(\frac{\partial}{\partial q} (\dot{q}^T \hat{H}(s(q); \beta) \dot{q}) \right)^T, \quad (4.4)$$

and again recurring to the chain rule and taking into account the embedding proposed, $s(q)$, this is simply:

$$C(q, \dot{q}; \beta) = \frac{\partial \hat{H}(s(q); \beta)}{\partial s} \cdot \frac{\partial s(q)}{\partial q} \dot{q} \dot{q} - \frac{1}{2} \left(\frac{\partial}{\partial s} (\dot{q}^T \hat{H}(s(q); \beta) \dot{q}) \cdot \frac{\partial s(q)}{\partial q} \right)^T. \quad (4.5)$$

Following the same principle the potential field is given by:

$$g(q, \psi) = \frac{\partial \hat{V}(s(q); \psi)}{\partial s} \cdot \frac{\partial s(q)}{\partial q}. \quad (4.6)$$

Observe that the output of V and H is not directly compared with their respective targets. This leads some authors to use the unsupervised learning term for the respective learning of these terms [7]. Instead, only the value of f is needed as ground truth, correspondent to the system's accelerations.

Parameter Optimization. With this in mind, the parameters $\theta = \beta \cup \psi$ can be obtained by minimizing the violation of the physical law described by the Lagrangian Mechanics for control affine systems:

$$(\beta^*, \psi^*) = \underset{\beta, \psi}{\operatorname{argmin}} \mathbb{L}\left(\hat{f}(q, \dot{q}, u; \beta, \psi), \ddot{q}\right) \quad (4.7)$$

$$\text{with } \hat{f}(q, \dot{q}, u; \alpha, \psi) = (\hat{L} \hat{L}^T)^{-1} \left(u - \frac{\partial}{\partial q} (\hat{L} \hat{L}^T) \dot{q} \dot{q} + \frac{1}{2} \left(\frac{\partial}{\partial q} (\dot{q}^T \hat{L} \hat{L}^T \dot{q}) \right)^T + \frac{\partial \hat{V}(s(q); \psi)}{\partial q} \right) \quad (4.8)$$

$$\text{s.t. } 0 < x^T \hat{L} \hat{L}^T x \quad x \in \mathbb{R}_0^n \quad (4.9)$$

Note that neither \hat{L} nor \hat{V} are functions of \dot{q} or u and, hence, the chosen hypothesis space by the learning algorithm will only depend on q , with the velocities, control inputs and derivative prior playing the role of regularizers. This neural network architecture is known in the literature by Deep Lagrangian Neural Networks [5].

This learning architecture is defined using a *continuous-time* formulation of Lagrangian mechanics. In this work we intend to learn Lagrangian dynamics of mechanical systems from discrete trajectories. A common choice is just to solve the dynamics, f , with an explicit numerical integrator and use the next state as regression target with consequent back-propagation through the integrator and respective dynamics [4, 6]. However, doing so leads to unnecessary and expensive gradient operations through the intermediate steps of the numerical integrator, specially when making use of priors with gradient operations during the forward pass, which require computationally expensive back-propagation operations. Equation (3.14) is an example of such computationally expensive priors. In the next section, we instead present a simple and compact approach by means of implicit differentiation.

4.2 Learning ODEs from discrete trajectories

Recent developments in neural networks have enabled the mimicking of the energy conservation law by learning the underlying continuous-time differential equations [5, 7, 14]. However, this may not be possible in discrete time, which is often the case in practical learning and computation [62]. While from the optimization problem (4.7) one can learn the Lagrangian dynamics given a dataset $\mathcal{D} = \{q_k, \dot{q}_k, u_k, \ddot{q}_k \mid k \in \{1, \dots, N\}\}$ of control actuation, states, and their corresponding time-rates-of-change. It happens that the generalized accelerations needed for the regression target are generally not directly measured, instead they are estimated by finite-difference approximation, which amplifies high-frequency noise present in the data samples [4].

Now we focus on the problem of learning ordinary differential equations from discrete data. Consider an ODE: $\dot{x} = f(x)$. Assuming unknown analytical expression of the function f and we wish to approximate it with a neural network. If we have a dataset $\mathcal{D} = \{x_t, u_t, x_{t+\Delta t} \mid t \in \{1, \dots, N\}\}$ of states, inputs and next states, we pretend to include discretization methods into the learning framework. This can be accomplished by a conversion from a continuous-time dynamical system to a discrete-time dynamical system $x_{t+\Delta t} = f_d(x_t, u_t)$ performed by means of an integration scheme. The inclusion of those into the learning architecture is discussed in detail below.

4.2.1 Neural Ordinary Differential Equations

In [18] authors introduced Neural Ordinary Differential Equations as differentiable ODE solvers with $\mathcal{O}(1)$ -memory back propagation. Neural ODEs are a relatively new class of models that transform data continuously through infinite-depth architectures, which has made them particularly suitable for learning the dynamics of complex physical systems. They can be used as a replacement of residual networks, for computing normalizing flows, and finally for modeling and making predictions from time-series data

[18]. The latter is the one that we are concerned the most with within this work. In fact, we are interested in combining these architectures to model and learn dynamical systems from data with and without prior knowledge of the underlying dynamics and assess the modelling capabilities against the approaches that will be further discussed.

How could we learn $f(x)$ from discrete data and use it to make predictions? Let the right hand side of the ODE be a parametric function $f(x(t), t; \theta)$ of θ , x and t and let $x_0 \dots x_T$ be an observed trajectory of states measured at uniformly spaced time points $t_0 \dots t_T$. The set of parameters θ that best represent the dynamics of the given observed data can be found by minimizing the mean squared error $\sum_{i=1}^T \|x_i - \hat{x}_i(\theta)\|_2^2$ between the ground truth trajectory $\{x_i\}_{i=0}^T$ and the trajectory $\{\hat{x}_i\}_{i=0}^T$ generated with our integrator of choice,

$$\{\hat{x}\}_{i=0}^T = \text{ODESolve}(x_0, f(x(t), t; \theta), \{t_i\}_{i=0}^T). \quad (4.10)$$

This problem formulation defines neural ODEs as neural network architectures that need the dynamics function to take in the current state $x(t)$ of the ODE, the current time t , and some parameters θ , and output $\frac{\partial x(t)}{\partial t}$, which has the same shape as $x(t)$. In fact residual neural networks can be seen as a single step of forward Euler discretization. For instance, a single process of a feedforward ResNet for a time-step $h = 1$ is given by:

$$x_{t+\Delta t} = x_t + f(x(t), t; \theta). \quad (4.11)$$

Each step taken by the ODE solver can be interpreted as a layer output in neural ODEs, similar to how the steps through each block of ResNets can be seen as a step of Euler's integration of differential equations [63].

The continuous derivative can be parameterized by a neural network with parameters θ defined by a function f :

$$\frac{\partial x(t)}{\partial t} = f(x(t), t; \theta). \quad (4.12)$$

Given an initial value problem (IVP) of the form of

$$\dot{x}(t) = f(x, t; \theta), \quad x(0) = x_0, \quad (4.13)$$

the output of a Neural ODE can be seen as the solution of the ODE solver at time T given by:

$$x(T) = x_0 + \int_0^T f(x(t), t; \theta) dt = \text{ODESolve}(x_0, f(x(t), t; \theta), \{t_i\}_{i=0}^T). \quad (4.14)$$

A neural ODE can be seen as a continuous-time or continuous-depth model with the ODE solver commanding the number of layers.

4.2.2 Gradient Computation through the Adjoint Sensitivity Method

The main challenge in training neural ODEs is performing backpropagation through the ODE solver. While differentiating through the individual operations of the ODE solver is straightforward, that incurs a high memory cost and introduces additional numerical error [18]. Note that the number of individual

steps necessary when integrating between two time points depends on the numerical integrator chosen. With this in mind and since gradients of the loss function with respect to the set of learning parameters are required for training, Chen et al. [18] introduced a method to overcome this by means of implicit methods³. Specifically, they propose to compute the gradients using the *adjoint sensitivity method* [64]. This approach computes gradients by solving a second state backwards in time, independently of the chosen ODE solver.

Consider optimizing the mean squared error loss function \mathbb{L} , whose input is the result of an ODE solver,

$$\mathbb{L}(x(T)) = \sum_{i=1}^T \|x - \hat{x}(\theta)\|_2^2 = \|x - \text{ODESolve}(x_0, f(x(t), t; \theta), \{t_i\}_{i=0}^T)\|_2^2, \quad (4.15)$$

then the next step is to determine the gradient of the loss with respect to the hidden state $x(t)$ at each instant t . This gradient is defined by a quantity called by the *adjoint*,

$$a(t) = \frac{\partial \mathbb{L}}{\partial x(t)}. \quad (4.16)$$

Furthermore, it is proven in appendix B.1 in [18] that the dynamics of the adjoint define another differential equation defined by:

$$\frac{da(t)}{dt} = -a(t)^T \frac{\partial f(x(t), t; \theta)}{\partial x}. \quad (4.17)$$

This differential equation can then be solved for $a(t)$ backwards in time, from t_T to t_0 , similar to regular backpropagation, with the gradient with respect to the initial value being given by

$$a(t_0) = a(t_T) + \int_T^0 \frac{da(t)}{dt} dt = a(t_T) - \int_T^0 a(t)^T \frac{\partial f(x(t), t; \theta)}{\partial x(t)} dt, \quad (4.18)$$

with initial condition given by the adjoint at time T

$$a(t_T) = \frac{\partial \mathbb{L}}{\partial x(t_T)}. \quad (4.19)$$

We can generalize equation (4.17) to obtain gradients of the loss with respect to θ (appendix B.2 of [18]) by

$$\frac{d\mathbb{L}}{d\theta} = - \int_T^0 a(t)^T \frac{\partial f(x(t), t; \theta)}{\partial \theta} dt. \quad (4.20)$$

All the gradients needed to backpropagate through the ODE solver can then be concatenated alongside the original state to form an augmented state by solving the ODE for the augmented state backwards in time. Most ODE solvers require multiple steps between two adjacent time-steps, in this case the gradients can be calculated by solving the augmented ODE backwards between each time step in the same way as for the adjoint state. These gradients are then summed up after each solved step (figure 4.5).

³Note that neural ODEs are in fact characterized by an implicit layer. Instead of specifying how to compute the layer's output from the input, we specify the conditions that we want the layer's output to satisfy, i.e. the solution of an initial value problem by a numerical integrator.

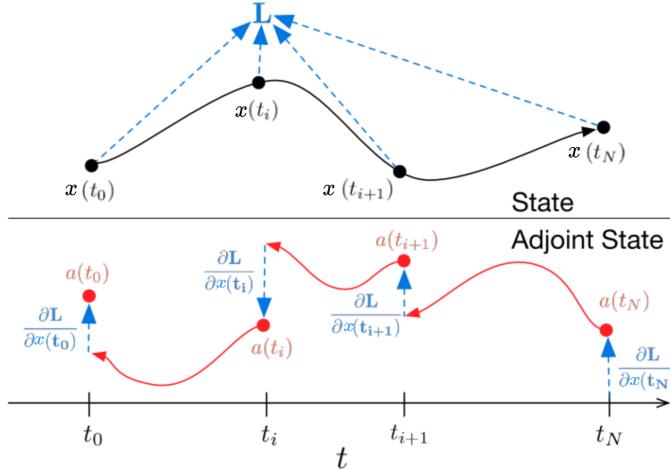


Figure 4.5: Reverse-mode differentiation of an ODE solution. The adjoint sensitivity method solves an augmented ODE backwards in time. The augmented system contains not only the original state but also the sensitivity of the loss with respect to the state and parameters. If the loss depends directly on the state at multiple observation times, the adjoint state must be updated in the direction of the partial derivative of the loss with respect to each observation. Retrieved from [18].

4.3 Modeling Mechanical Systems with Neural ODEs

The fact that neural ODEs approximate the derivative of a system makes these models powerful tools for system identification of dynamical systems. Given a time series data set described by an ODE, one can learn the underlying dynamics of the system with or without physical priors by integrating the model between any two data points in the data set and then perform implicit backpropagation via the adjoint sensitivity method to update the model parameters.

One of the reasons for the great success of neural ODEs, is in part related to the fact that these models are continuously defined and they can be integrated between any two time points, eliminating the common problem of modelling data with irregular time steps. Neural ODEs are then time series models continuously-defined whose dynamics can incorporate data which arrives at arbitrary times in a straightforward way. Further, they are memory efficient since there is no need to store all intermediate operations while backpropagating [18].

4.3.1 Second Order Neural Ordinary Differential Equations

The first challenge towards leveraging neural ODEs to learn state-space models of mechanical systems is the fact that the dynamics arising from such dynamical systems are defined by a second order ordinary differential equation. However, the dynamics can, instead, be seen as a system of coupled first-order ODEs with state $x = [q, \dot{q}]^T$:

$$\dot{x} = f^{(x)}(x, u; \theta) = \begin{bmatrix} \dot{q} \\ f^{(a)}(x, u, \theta) \end{bmatrix} \quad (4.21)$$

where $f^{(a)}(x, u, \theta)$ is a neural network with parameters θ that maps the states and controls to the generalized accelerations. This formulation allows the reuse of the adjoint sensitivity method for second order ODEs.

Augmenting the dynamics. The second challenge is the incorporation of the control term into the dynamics. Note that a function of this form cannot be directly fed into a neural ODE since the domain and range of f have different dimensions. However, assuming the control terms to be fixed throughout the trajectory, i.e., $\dot{u} = 0$ and given as observations, we can add an extra degree of freedom and ensure that the domain and range have the same shape.

Several approaches have extended neural ODEs to this particular cases through augmentation strategies. In particular 0-augmentation (ANODEs) [65] and Input Layer augmentation (IL-NODEs) in [66]. Here, we consider that our data consists of trajectories $(x, u)_{t_0, \dots, t_T}$ with fixed control inputs. The state augmentation can then be formulated as:

$$\begin{bmatrix} \dot{x} \\ \dot{u} \end{bmatrix} = \begin{bmatrix} f^{(\dot{x})}(x, u; \theta) \\ 0 \end{bmatrix} = f(x, u; \theta), \quad (4.22)$$

where we remove the time dependence of the dynamics due to the autonomous nature of the ODEs present in this study.

The training trajectories can consist of multiple steps, $T > 1$, or single steps, $T = 1$. Here we only consider the latter. First to be coherent with the models we will discuss in further sections. Secondly because to do the recurrent training⁴ either augmentation strategies would be required and, to the best of our knowledge, no prior work has been done on that in terms of constrained system identification of mechanical systems, or the control inputs had to be constant throughout the complete trajectory.

The augmentation strategy of equation (4.22) has been previously applied in [15, 67]. Even though the advantages of neural ODEs are more notorious in the recurrent training [8], making use of the adjoint sensitivity method still makes these models computationally attractive as the back-propagation through the intermediate steps of the numerical integrator chosen is not required.

4.3.2 ODE solvers

Incorporating the differential ODE solver introduces new hyperparameters: solver types, and time horizon. The latter, as already mentioned, is set to $T = 1$. For the solver types, we use the fourth-order Runge-Kutta (RK4) with 3/8 rule and the midpoint method (see section 3.2). The framework for neural ODEs is implemented by Chen et al, the authors of the neural ODEs [18], and can be found in their Github repository⁵.

Modern ODE solvers such as the class of adaptive integrators provide guarantees about the growth of the approximation error and adapt their evaluation strategy on the fly to achieve the supposed ac-

⁴Back-propagating through multiple steps of the training integrator is comparable to back-propagating through time in recurrent networks [8]

⁵<https://github.com/rtqichen/torchdiffeq>

curacy. This will lead to a hypothetical increase in the number of function evaluations, both for making predictions and calculate the gradients. Empirical tests showed that using an adaptive integrator such as the "dopri5" lead to similar results as the RK4 but required more time during training. Hence, the choice on this thesis relies on fixed-step algorithms.

4.3.3 Geometric Neural ODE

One of the baselines used throughout this work is based on a standard fully connected feed-forward neural network, followed by a numerical ODE solver, where its input relies on the intended geometric embedding (hence the designation Geometric Neural ODE (NODE)). Here, we consider a dataset $\mathcal{D} = \{..., (x_t, u_t, x_{t+h}), ... | i \in \{1, ..., N\}\}$ with N samples of present states, control inputs and next states and we intent to perform system identification of mechanical systems augmenting the dynamics according with equation (4.22). The dynamics, $f^{(a)}(x, u; \theta)$, are approximated by feed-forward neural network which together with the augmented dynamics form the augmented function $f(x, u; \theta)$.

Optimization is simply minimizing the mean squared error loss $\mathbb{L} = \sum_{i=1}^N \|\hat{x}_{i,t+h} - x_{i,t+h}\|_2^2$ over integration between two adjacent time-steps. A detailed architecture can be found in figure 4.6.

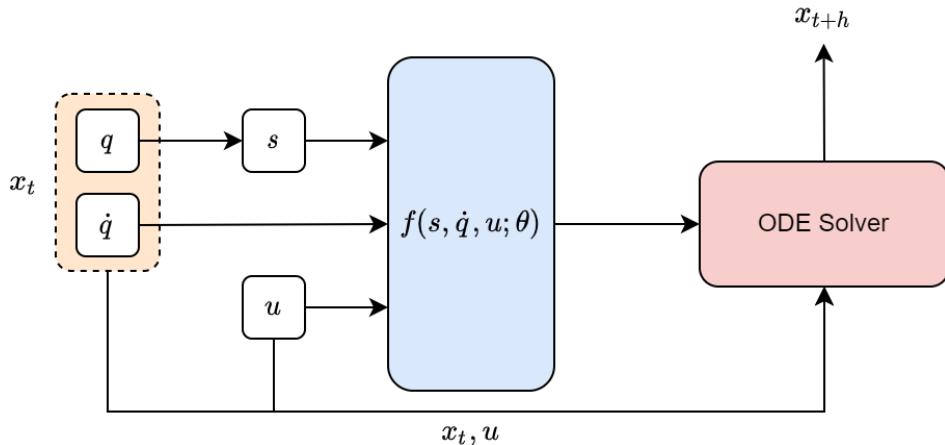


Figure 4.6: Geometric Neural ODE with the embedding proposed. Given a present state we embed the configuration space with a transformation s , and alongside with the velocity and control space we feed the MLP that represents the parameterization of the dynamics that shall be solved through time via an numerical integrator between two adjacent time steps. Backpropagation makes use of the adjoint sensitivity method that solves an augmented ODE backwards between two time steps.

4.3.4 Lagrangian Neural ODE

So far we have discussed the usage of continuous Lagrangian mechanics as an implicit constraint for an end-to-end learning framework of Lagrangian mechanics. We have also discussed neural ODEs that allow the incorporation of numerical discretization schemes into the learning in an efficient and compact way. Here, we want to combine the prior of the underlying dynamics with Neural ODEs. This is performed

by enforcing equation (3.14) into the learning problem, i.e, mimicking the Euler-Lagrange equations by learning from data the lower triangular matrix, $\hat{L}(s; \beta)$, that is further used to build the inertia matrix, $\hat{H}(q; \beta)$ via Cholesky decomposition, and the potential energy, $\hat{V}(s; \psi)$, both relying on the appropriate embedding.

Performing the gradient operations with automatic differentiation in $\hat{H}(s; \beta)$ and $\hat{V}(s; \psi)$ according with equation (4.5) and (4.6) we obtain the potential field, $g(q; \psi)$, and the Coriolis and Centrifugal terms, $C(q, \dot{q}; \beta)$. The resulting equations yield a second order ODE that maps the state and control inputs to the generalized accelerations according with equation (3.14). Therefore, we intend to learn the dynamics with parameterizations implicitly constrained with the following prior,

$$f^{(a)}(x, u; \theta) = \left(\hat{L}(s(q); \beta) \hat{L}(s(q); \beta)^T \right)^{-1} \left(u - \frac{d}{dt} (\hat{L}(s(q); \beta) \hat{L}(s(q); \beta)^T) \dot{q} \right. \\ \left. + \frac{1}{2} \left(\frac{\partial}{\partial q} \left(\dot{q}^T \hat{L}(s(q); \beta) \hat{L}(s(q); \beta)^T \dot{q} \right) \right)^T + \frac{\partial \hat{V}(s(q); \psi)}{\partial q} \right), \quad (4.23)$$

such that the full augmented dynamics fed to the ODE solver are formulated by:

$$f(x, u; \theta) = \begin{bmatrix} \frac{dq}{dt} \\ \frac{d\dot{q}}{dt} \\ \frac{du}{dt} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ f^{(a)}(x, u; \theta) \\ 0 \end{bmatrix}. \quad (4.24)$$

The regression predictions can be obtained by integrating the augmented dynamics between two adjacent time steps with an appropriate numerical integrator via NODEs,

$$\hat{x}_{t+h} = \text{ODESolve}(x_t, f(x_t, u; \theta), t, h). \quad (4.25)$$

Given a dataset $\mathcal{D} = \{..., (x_t, u_k, x_{t+h})_i, ... \mid i \in \{1, \dots, N\}\}$ with N samples of present states, control inputs and next states, the optimization problem consists in finding the set of parameters θ that minimizes the mean squared error,

$$\underset{\theta^*}{\text{minimize}} \quad \frac{1}{N} \sum_{i=1}^N \|\hat{x}_{i,t+h} - x_{i,t+h}\|_2^2, \quad (4.26)$$

with back-propagation being performed by the adjoint sensitivity method. We will designate this architecture henceforth by L-NODE. A more illustrative architecture can be found on figure 4.7.

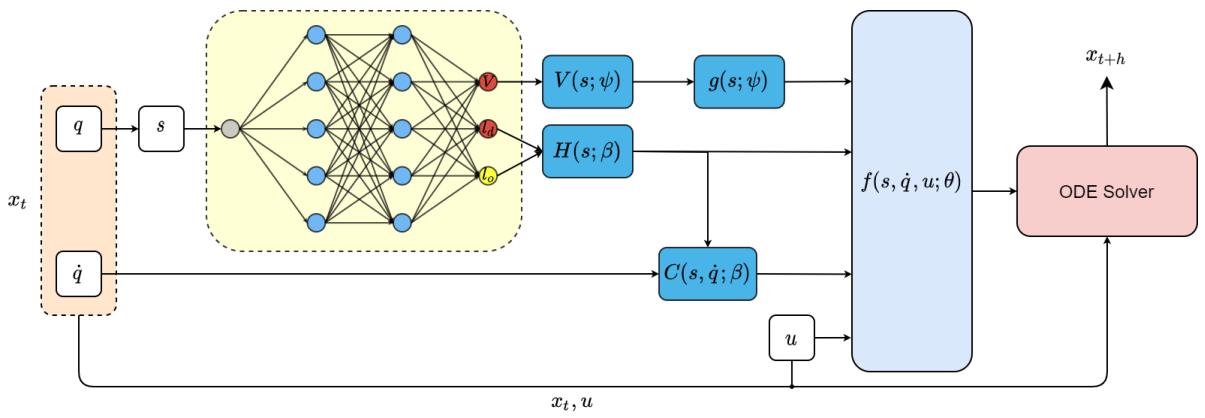


Figure 4.7: Lagrangian Neural ODE with the embedding proposed. Given a present state we create an embedding s in the configuration space that is further fed into the neural network that parameterizes a low triangular matrix and the potential energy. The lower triangular obtained from l_0 and l_d is reshaped by means of Cholesky decomposition to obtain the positive definite inertia matrix. The C represents the centrifugal and Coriolis effects that depend on the inertia matrix. The term g is the potential field obtained by performing automatic differentiation with respect to q . These terms together, form the function f that shall be solved through time via a numerical integrator between two time steps. Backpropagation makes use of the adjoint sensitivity method that solves an augmented ODE backwards between two time steps.

Chapter 5

Combining Discrete Variational Mechanics with Deep Learning

In this chapter we propose a novel neural network architecture that we call by Symplectic-Momentum Neural Networks (SyMo) due to the fact that the resulting integrators are symplectic-momentum integrators. This neural network architecture is derived directly from the context of Discrete Mechanics and its purpose is to combine variational integrators for mechanical systems with configuration-dependent inertia (non-separable Hamiltonians) with traditional deep learning techniques. We extend SyMos to accomodate the implicit root-finding operation by developing an implicit layer that solves the root. This leads to End-To-End Symplectic Momentum Neural Networks (E2E-SyMo). Using such strong priors with state-of-the-art learning algorithms is of extreme importance to obtain physical-coherent learning parameterizations.

5.1 Symplectic-Momentum Neural Networks

5.1.1 Problem Formulation

Variational integrators for mechanical systems present an implicit nature, and hence they are not so amenable for an end-to-end learning as the models derived in previous sections arising from the continuous formulation of Lagrangian mechanics. However, following the same line of reasoning of such and, instead of learning the dynamics from the Euler-Lagrangian equations, we can learn the dynamics from the implicit Discrete Euler-Lagrangian equations and using those to make predictions implicitly.

Let the observations be of the form $(q_{k-1}, q_k, q_{k+1}, u_{k-1}, u_k, u_{k+1})$, where q_{k+1} are the regression targets. The input space can, in turn, be defined by a set of two adjacent points in the configuration space and three adjacent discrete control inputs $x = (q_{k-1}, q_k, u_{k-1}, u_k, u_{k+1}) \in \mathcal{X}$ and the output space to be $y = (q_{k+1}) \in \mathcal{Y}$. We intend to build a neural network architecture that captures dependencies between the input and output space by using the Discrete Euler-Lagrange equation as a function that correlates both spaces.

Given input and output spaces $\mathcal{X} \times \mathcal{Y}$, we want to build a neural network based architecture over the (x, y) pairs that encodes a function, $g(x, y; \theta)$, that relates those pairs with the discrete Euler-Lagrange equations.

Recall that the DEL is given in equation (3.43). Here, we want to build $g(x, y; \theta)$ based on a parameterization of the discrete Lagrangian, by the set of learning parameters θ ,

$$g(x, y; \theta) = D_1 \mathcal{L}_d(q_k, q_{k+1}; \theta) + D_2 \mathcal{L}_d(q_{k-1}, q_k; \theta) + f_d^+(q_{k-1}, q_k, u_{k-1}) + f_d^-(q_k, q_{k+1}, u_k) = 0, \quad (5.1)$$

then learning consists in minimizing the violation of the DEL equation given the input and output space. This can be obtained by adjusting the free parameters, θ to minimize the loss function:

$$\theta^* = \arg \min_{\theta} \left(\frac{1}{N} \sum_{i=1}^N \|g(x_i, y_i; \theta)\|_2^2 \right) \quad (5.2)$$

The building up process of the discrete Lagrangian is derived from the quadrature rule used for discretizing the equations of motion. Our choice will be discussed in next section.

Inference consists in finding configurations of the variables q_{k+1} , obtained by implicitly solving the parameterized DEL, through the Newton's root finding algorithm:

$$\hat{q}_{k+1} = \text{RootFind}(g(q_{k+1}; \theta^*)). \quad (5.3)$$

Existence and uniqueness of solution is guaranteed under the regularity of the Lagrangian, which is accomplished for Lagrangians of the form of (3.10) (see A.1 for further details).

5.1.2 Discretization

Learning the discrete equations of motion with equation (5.1), as an implicit constraint, requires building a discrete Lagrangian that can be seen as an approximation of the action integral (equation 3.20).

While one could learn \mathcal{L}_d directly from data, since no prior about the form of the Lagrangian is fed into the learning algorithm, during the process of root finding necessary for inference the learned g might not guarantee solution existence, specially for out-of-sample scenarios and for low data training regimes since the regularity condition is not implicitly enforced¹. With that in mind, we enforce the Lagrangian regularity condition and with that solution existence by making use of the same neural network parameterization of Section 4.1, adapted to the discrete formulation of mechanics (section 3.3).

Choice of a Discrete Lagrangian. To keep a trade-off between accuracy and efficiency we make

¹ Interesting approaches but not sufficient (to the best of our knowledge) can be found on the literature. For instance in [68] authors present a neural network architecture that is convex with respect to a subset of its inputs. However, to enforce the regularity condition we need to enforce that the learned L_d is strictly convex w.r.t. to the velocities. While this could empirically solve the solution existence problem it is still not a sufficient condition.

use of the midpoint rule for approximating the discrete Lagrangian

$$\mathcal{L}_d(q_k, q_{k+1}; \theta) = h \mathcal{L}\left((1 - \alpha)q_k + \alpha q_{k+1}, \frac{q_{k+1} - q_k}{h}; \theta\right) \approx \int_{t_k}^{t_{k+1}} \mathcal{L}(q(t), \dot{q}(t)) dt, \quad (5.4)$$

where $\alpha \in [0, 1]$ is an algorithm parameter. For $\alpha = 0.5$, the above approximation corresponds to the midpoint rule, leading to second order accuracy [51]. From now on, we will denote by

$$q_{k+1/2} = \frac{1}{2}q_k + \frac{1}{2}q_{k+1}, \quad (5.5)$$

$$q_{k-1/2} = \frac{1}{2}q_{k-1} + \frac{1}{2}q_k, \quad (5.6)$$

the collocation points for approximating the Lagrangian integral during the time-step that goes from q_{k-1} to q_k and from q_k to q_{k+1} , respectively.

Note that the order of accuracy of the integrator is directly related with the order of accuracy of the quadrature rule [19], which leads the variational integrator derived to be second-order accurate. However, higher order variational integrators could be obtained by choosing more accurate quadrature rules [19].

Building regular Discrete Lagrangians with Neural Networks. For Lagrangians of Mechanical Systems (see equation (3.10)) and using the neural network architecture aforementioned to parameterize the inertia matrix, $\hat{H}(q_{k\pm 1/2}; \beta)$, at the collocation points $q_{k-1/2}$ and $q_{k+1/2}$, through the lower triangular matrix $\hat{L}(q_{k\pm 1/2}; \beta)$, and the potential energy, $\hat{V}(q_{k\pm 1/2}; \psi)$, the discrete Lagrangians for the three adjacent time-steps are built upon the special structure of the approximated Lagrangian integral,

$$\mathcal{L}_d(q_{k-1}, q_k; \theta) = h \left[\left(\frac{q_k - q_{k-1}}{h} \right) \hat{H}(q_{k-1/2}; \beta) \left(\frac{q_k - q_{k-1}}{h} \right)^T - \hat{V}(q_{k-1/2}; \psi) \right], \quad (5.7)$$

$$\mathcal{L}_d(q_k, q_{k+1}; \theta) = h \left[\left(\frac{q_{k+1} - q_k}{h} \right) \hat{H}(q_{k+1/2}; \beta) \left(\frac{q_{k+1} - q_k}{h} \right)^T - \hat{V}(q_{k+1/2}; \psi) \right], \quad (5.8)$$

where the velocities are approximated based on two adjacent points on the configuration space, for instance, $\dot{q}_{k+1} = \frac{q_{k+1} - q_k}{h}$. Recall that, according to equation (4.2), the inertia matrix is built based on its Cholesky factor, \hat{L} ,

$$\hat{H}(q_{k\pm 1/2}; \beta) = \hat{L}(q_{k\pm 1/2}; \beta) \hat{L}(q_{k\pm 1/2}; \beta)^T, \quad (5.9)$$

with \hat{L} enforced as a lower-triangular matrix with positive diagonal entries, which enforces symmetry and positive definiteness of the inertia. Note that this enforcement of the regularity condition of the continuous Lagrangian will automatically render regular discrete Lagrangians for sufficient small time-steps, h , and close adjacent generalized coordinates [19]. Once the discrete Lagrangian is regular, then existence and uniqueness are guaranteed under the special conditions of annex A.1. Note that now the learning is also a function of the time-step.

Discrete Forces. Similar to the discrete Lagrangian the forcing is approximated with the an appro-

priate rule, i.e, given a sequence of three adjacent control inputs, u_{k-1}, u_k, u_{k+1} we can approximate the virtual work (equation 3.37) by

$$\int_{t_k}^{t_{k+1}} F(q(t), \dot{q}(t), u(t)) \delta q dt \approx hF\left((1-\alpha)q_k + \alpha q_{k+1}, \frac{q_{k+1} - q_k}{h}, (1-\alpha)u_k + \alpha u_{k+1}\right)(1-\alpha)\delta q_k + \\ \alpha hF\left((1-\alpha)q_k + \alpha q_{k+1}, \frac{q_{k+1} - q_k}{h}, (1-\alpha)u_k + \alpha u_{k+1}\right)\delta q_{k+1}, \quad (5.10)$$

which from equation (3.37) automatically leads to the right and left discrete forces,

$$f_d^-(q_k, q_{k+1}, u_k, u_{k+1}) = (1-\alpha)hF\left((1-\alpha)q_k + \alpha q_{k+1}, \frac{q_{k+1} - q_k}{h}, (1-\alpha)u_k + \alpha u_{k+1}\right), \quad (5.11)$$

$$f_d^+(q_{k-1}, q_k, u_{k-1}, u_k) = \alpha hF\left((1-\alpha)q_{k-1} + \alpha q_k, \frac{q_k - q_{k-1}}{h}, (1-\alpha)u_{k-1} + \alpha u_k\right). \quad (5.12)$$

Similarly, we use the midpoint rule ($\alpha = 0.5$), leading to full second-order of accuracy, in the forces and discrete Lagrangian. The collocation points for the control inputs can be denoted by:

$$u_{k+1/2} = \frac{1}{2}u_k + \frac{1}{2}u_{k+1}, \quad (5.13)$$

$$u_{k-1/2} = \frac{1}{2}u_{k-1} + \frac{1}{2}u_k, \quad (5.14)$$

which, in turn, leads to the following discrete forces for control affine systems

$$f_d^-(q_k, q_{k+1}, u_k, u_{k+1}) = \frac{h}{2}F\left(q_{k+1/2}, \frac{q_{k+1} - q_k}{h}, u_{k+1/2}\right) = \frac{h}{4}(u_{k+1} + u_k), \quad (5.15)$$

$$f_d^+(q_{k-1}, q_k, u_{k-1}, u_k) = \frac{h}{2}F\left(q_{k-1/2}, \frac{q_k - q_{k-1}}{h}, u_{k-1/2}\right) = \frac{h}{4}(u_k + u_{k-1}). \quad (5.16)$$

Learning from Embedded Data. In section 4.1.3, we considered observations in R^n and further applied appropriated embeddings to the observations that follow angular coordinates, making the learning free of singularities. Here, we make use of the same embedding but on the collocation points, $q_{k-1/2}$ and $q_{k+1/2}$, instead.

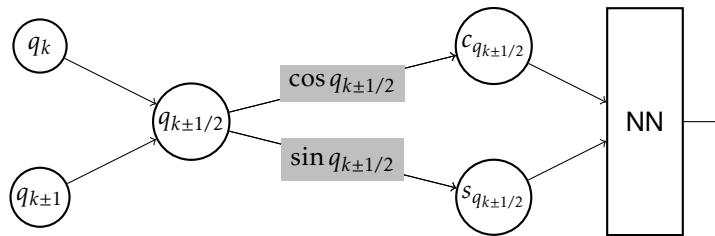


Figure 5.1: Computational Graph of the collocation point followed by the angle embedding from R^m to T^m used to feed the Neural Network block. Gradients are still calculated w.r.t. q_k or q_{k+1} by the chain rule.

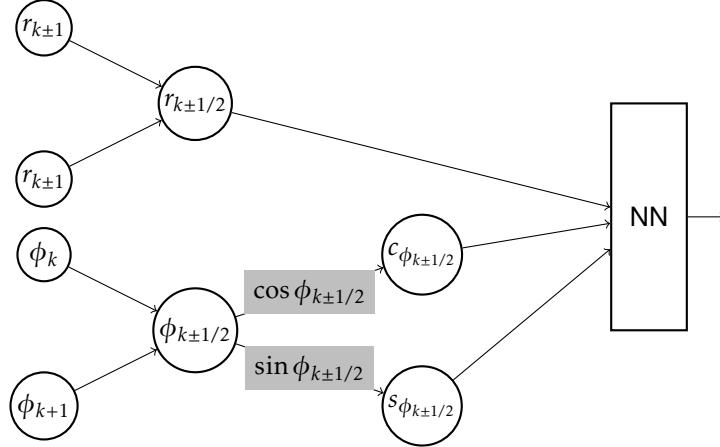


Figure 5.2: Computational Graph of collocation point and the embedding from $\mathbb{R}^n \times \mathbb{R}^m$ to $\mathbb{R}^n \times \mathbb{T}^m$ used to feed the Neural Network block. Similarly the embedding does not change the underlying discrete dynamics as long as the gradients are performed according with the original equations.

5.1.3 Parameter Optimization

Aggregating all concepts explained above, we are now able to define the optimization scheme for SyMos. We assume observations of the form of $(q_{k-1}, q_k, q_{k+1}, u_{k-1} u_k, u_{k+1})$. These are divided into the input space $x = (q_{k-1}, q_k, u_{k-1} u_k, u_{k+1})$, and output space $y = q_{k+1}$.

For each forward pass the computational graph considers two calls to the same neural network. For instance, we need the approximation for the Lagrangian integral between two adjacent time-steps, i.e, we need one call for $q_{k-1/2}$ and another one for $q_{k+1/2}$. These calls are then made by the following computations,

$$\mathcal{L}_d(q_k, q_{k+1}; \theta) = h \left[\left(\frac{q_{k+1} - q_k}{h} \right) \hat{L}(s_{k+1/2}; \beta) \hat{L}(s_{k+1/2}; \beta)^T \left(\frac{q_{k+1} - q_k}{h} \right)^T - \hat{V}(s_{k+1/2}; \psi) \right], \quad (5.17)$$

$$\mathcal{L}_d(q_{k-1}, q_k; \theta) = h \left[\left(\frac{q_k - q_{k-1}}{h} \right) \hat{L}(s_{k-1/2}; \beta) \hat{L}(s_{k-1/2}; \beta)^T \left(\frac{q_k - q_{k-1}}{h} \right)^T - \hat{V}(s_{k-1/2}; \psi) \right], \quad (5.18)$$

with $s_{q_{k+1/2}}$ and $s_{q_{k-1/2}}$ being functions of the pairs (q_{k+1}, q_k) and (q_k, q_{k-1}) respectively, and mapping the configuration space to the embedding space.

Formally we are now able to write the Discrete-Euler Lagrange equation in terms of the learning parameters, input and output spaces by:

$$g(x, y, h; \theta) = \frac{\partial}{\partial q_k} [\mathcal{L}_d(q_{k-1}, q_k; \theta) + \mathcal{L}_d(q_k, q_{k+1}; \theta)] + f_d^-(q_k, q_{k+1}, u_k, u_{k+1}) + f_d^+(q_{k-1}, q_k, u_{k-1}, u_k) = 0, \quad (5.19)$$

with \mathcal{L}_d given by equations (5.17) and (5.18) and the right and left discrete forces by equations (5.15) and (5.16).

Note that the time-step is multiplied in all terms of the summation. This leads, the time-step to be simply a scaling factor. The fact that the learning depends on the time-step is then not critical for making predictions with different time-steps. Further, similar with the case of Neural ODEs, the training can also be based on trajectories with irregular time-steps.

Learning consists in minimizing the squared Discrete Euler-Lagrange equations,

$$\underset{\theta^*}{\text{minimize}} \quad \frac{1}{N} \sum_{i=1}^N \|g(x_i, y_i; \theta)\|_2^2. \quad (5.20)$$

The conceptual architecture can be found in figure 5.3.

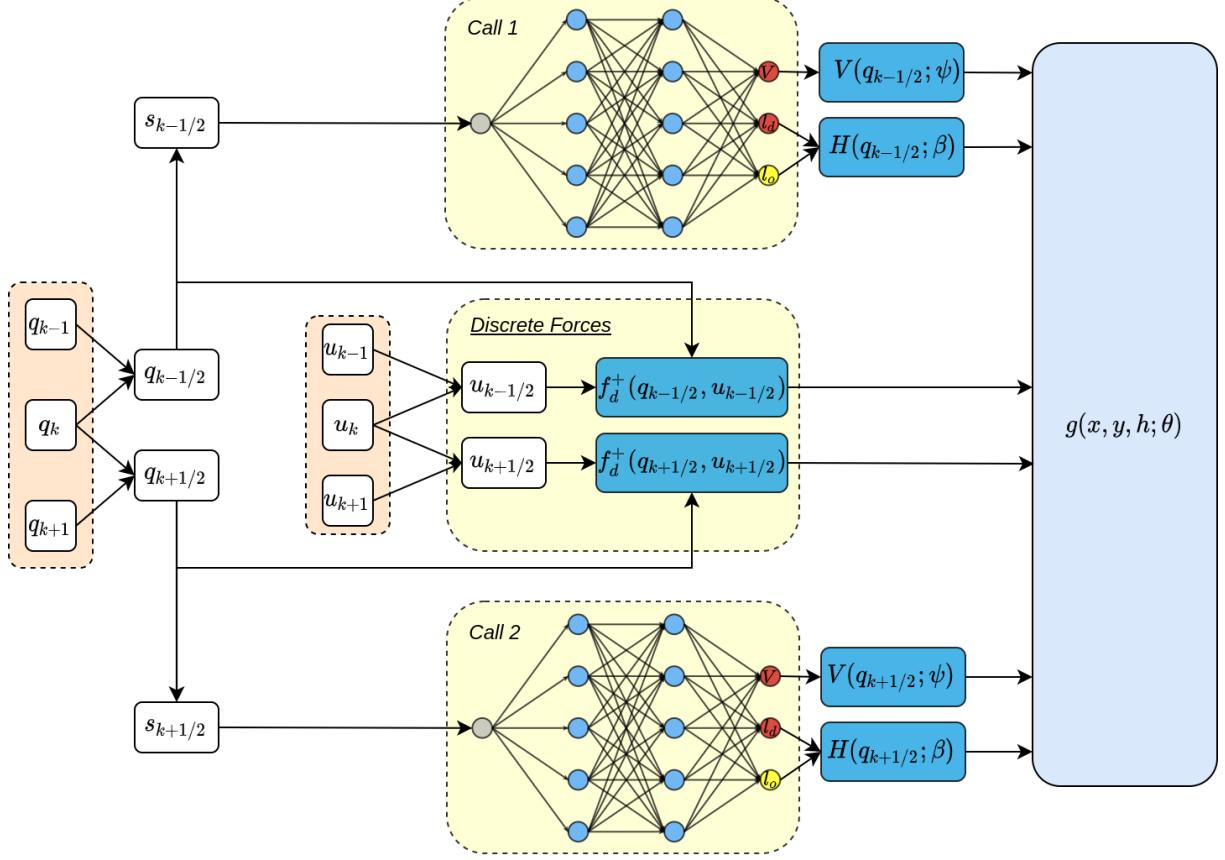


Figure 5.3: Symplectic Momentum Neural Network with the embedding proposed. Given the input and output state, we form the collocation points (for $\alpha = 0.5$) and apply to them the embedding $s_{k\pm 1/2}$. Given the embeddings, we perform two calls to the neural network to get the inertia and potential energy correspondent to the two adjacent time-steps. These terms, alongside the discrete forces form then the Discrete Euler-Lagrange equations, $g(x, y, h; \theta)$ as a function of the learning parameters.

5.1.4 Inference in SyMos

Once the discrete Lagrangian density \mathcal{L}_d is approximated by a neural network, based on the parameterization proposed, the learned discrete dynamics are ready to be served to predict new observations for q_{k+1} . As already mentioned, the discrete Euler-Lagrange equations are defined by an implicit mapping, and consequently a root-finding algorithm shall be used. The simplest method is Newton's algorithm. Newton's method seeks for producing successively better approximations to the roots based on a sufficiently close initial guess, x_0 . The process is repeated as

$$x_{n+1} = x_n - \frac{g(x_n)}{\nabla g(x_n)}, \quad (5.21)$$

until a given tolerance or a maximum number of iterations is reached. The method generally converges, provided a close enough initial guess to the unknown zeros, and that the derivative $\nabla g(x)$ is non-singular.

Even though quasi-Newton methods solve the root-finding problem for the DEL in $\mathcal{O}(n)$, these methods generally need more iterations to converge. The choice between Newton and quasi-Newton methods rely on whether we intend "a few costly steps" or "many many cheap steps". This leads us, to be agnostic in terms of the solver used for inference, specially for mechanical systems with a low number of degrees of freedom and rely on Newton's method.

Initial Guess. Given the convergence problem for distant initial guesses, we employ the explicit Euler integration, i.e, the initial guess for the next configuration, $q_{k+1}^{(0)}$, is given by $q_{k+1}^{(0)} = q_k + h \cdot \dot{q}_k$, where \dot{q}_k is approximated by $\dot{q}_k = \frac{1}{h}(q_k - q_{k-1})$, which simply leads to:

$$q_{k+1}^{(0)} = 2q_k - q_{k-1}. \quad (5.22)$$

With this initial guess, we are now able to formulate the inference procedure in SyMos. Full method can be seen in algorithm 3. Inference consists in solving the $B.n$ roots in parallel. With B being the number of samples and n the number of degrees of freedom of the system. The tolerance has both these parameters into account, which makes inference fair for different number of samples (inference batches). We define an adaptive tolerance $\epsilon = T \cdot \sqrt{B.n}$, where T is the intended fixed tolerance. We use the Frobenius norm, denoted by $\|\cdot\|_F$, to check if the DEL are close enough from zero up to a given tolerance. The norm is defined by $\|g\|_F = \left(\sum_{i=1}^B \sum_{j=1}^n g_{ij}^2\right)^{1/2}$.

Algorithm 3: Newton's Method for inference in SyMo.

```

Function DiscreteLagrangian( $q_k, q_{k+1}$ ):
     $s_{k+1/2} \leftarrow s(q_{k+1/2})$                                 // Embedding point
     $L(q_{k+1/2}; \beta), V(q_k, q_{k+1}; \psi) \leftarrow f(s_{k+1/2}; \theta)$  // Neural Network output
     $H(q_k, q_{k+1}; \beta) = L((q_{k+1/2}; \beta) L(q_{k+1/2}; \beta)^T$ 
     $\mathcal{L}_d(q_k, q_{k+1}; \theta) = h \left[ \left( \frac{q_{k+1}-q_k}{h} \right) H(q_k, q_{k+1}; \beta) \left( \frac{q_{k+1}-q_k}{h} \right)^T - V((s_{k+1/2}; \psi) \right]$ 
    return  $\mathcal{L}_d(q_k, q_{k+1}; \theta)$ 

Function DiscreteEulerLagrange( $q_{k+1}$ ):
     $q_{k-1/2} = 0.5q_{k-1} + 0.5q_k$                                 // Collocation point at first time-step
     $q_{k+1/2} = 0.5q_k + 0.5q_{k+1}$                                 // Collocation point at second time-step
     $\mathcal{L}_d(q_{k-1}, q_k; \theta) \leftarrow \text{DiscreteLagrangian}(q_{k-1}, q_k)$ 
     $\mathcal{L}_d(q_k, q_{k+1}; \theta) \leftarrow \text{DiscreteLagrangian}(q_k, q_{k+1})$ 
     $g(q_{k+1}; \theta) = D_2 \mathcal{L}_d(q_{k-1}, q_k; \theta) + D_1 \mathcal{L}_d(q_k, q_{k+1}; \theta) + f_d^-(q_k, q_{k+1}, u_k, u_{k+1}) + f_d^+(q_{k-1}, q_k, u_{k-1}, u_k)$ 

Procedure RootFind:
     $\epsilon = T \cdot \sqrt{n \cdot B}$ ;                                // Adaptive tolerance
     $q_{k+1}^0 = 2q_k - q_{k-1}$                                 // Initial Guess
    while num_iteration < maxiter do
         $g(q_{k+1}^{(i)}; \theta) \leftarrow \text{DiscreteEulerLagrange}(q_{k+1}^{(i)})$ 
         $J_g = \nabla g$                                          // Get gradient
         $q_{k+1}^{(i+1)} = q_{k+1}^{(i)} - [J_g(q_{k+1}^{(i)})]^{-1} g(q_{k+1}^{(i)})$  // Update estimate
        if  $\|g^{(i+1)}(q_{k+1}^{(i)}; \theta)\|_F \leq \epsilon$  then
            return  $q_{k+1}^{(i+1)}$ 
    return  $q_{k+1}^{(i+1)}$ 

```

5.2 End-to End Symplectic-Momentum Neural Networks

So far, we have established a neural architecture that takes into account the underlying discrete equations of motion arising from the discrete variational principles [19, 51], and using a root finding algorithm based on the architecture to implicitly integrate the dynamics in a time marching process. However, we have not discussed yet the process of including the resulting integrators into the learning process. These integrators, as already mentioned, are designated by variational integrators and consist in solving the DEL, which for non-separable Lagrangians, such as the ones generally present in mechanical systems, result in an implicit integrator.

Conventional deep learning techniques can not handle efficiently implicit methods. While it would be possible to implement solution procedures, especially those involving iterative updates, such as root-finding algorithms, directly within the Automatic Differentiation library, similar with the intermediate operation of an ODE solver, it would be necessary to store the computation graph for the complete solution procedure, along with the value of temporary iterates created during this solution. This process would result in extremely computationally expensive learning of complex neural network architectures, such as the one formulated in last section, that requires already gradient operations to form the DEL.

5.2.1 Implicit Differentiation through Gray-Box RootFind Solvers

Luckily, all the process aforementioned can be avoided by means of implicit differentiation. In fact, in calculus, implicit differentiation makes use of the chain rule to differentiate implicitly defined functions. This allows SyMos to be extended for an end-to-end learning. Assuming that the regression target is now the roots of the implicit discrete Euler-Lagrange equations, q_{k+1} , the loss can be defined by

$$\mathbb{L}_\theta = \frac{1}{N} \sum_{i=1}^N \|\hat{q}_{k+1} - q_{k+1}\|_2^2. \quad (5.23)$$

It is now necessary to compute the gradients of the output, with respect to the parameters θ . This is trivial by means of implicit differentiation. We provide below an alternative procedure that assumes constant memory when compared to back-propagating through the iterations of the gray-box² RootFind solver. Further, it assumes no knowledge of the method used for finding the roots.

Technically, in order to ensure that we can actually apply the implicit function theorem, we require that certain conditions must be satisfied, so that the implicit function $g(x, y)$ has, indeed, a solution y for some function $f(x)$, such that $y = f(x)$. These conditions are stated in what is known as the implicit function theorem (theorem A.2.1).

Theorem 5.2.1. Gradient of the RootFind solution. *Let $q_{k+1} \in \mathbb{R}^n$ be the solution to the physical constrained parameterized RootFind procedure based on the implicit DEL mapping $(q_{k-1}, q_k) \rightarrow (q_k, q_{k+1})$, defined by $g(q_{k+1}, x; \theta) \in \mathbb{R}^n$ (equation 5.19). The gradients of a scalar loss function $\mathbb{L}(q_{k+1}, x; \theta)$ (equation 5.23) with respect to the parameters θ are obtained by vector-Matrix products as follows:*

$$\frac{\partial \mathbb{L}}{\partial \theta} = -\frac{\partial \mathbb{L}}{\partial q_{k+1}} \left[\frac{\partial \mathcal{L}(q_k, q_{k+1}; \theta)}{\partial q_{k+1} \partial q_k} + \frac{\partial f_d^-(q_k, q_{k+1}, u_k, u_{k+1})}{\partial q_{k+1}} \right]^{-1} \frac{\partial g(q_{k+1}, x; \theta)}{\partial \theta} \quad (5.24)$$

The proof is provided in Appendix A.2. The insight provided by Theorem 5.2.1 is at the core of End-to-End Symplectic-Momentum Neural Networks (E2E-SyMo). Note that, the backward gradient through the “infinite” number of intermediate operations can be represented as one step of matrix multiplication that involves the Jacobian of the DEL at the root.

5.2.2 RootFind Layer

Forward Pass. Opposite to conventional neural networks where the output is the activation from the L-th layer, here, the output are the roots of the parameterized DEL equations (figure 5.4) solved by any root-finding algorithm, i.e., the output is

$$q_{k+1} = \text{RootFind}_{q_{k+1}}(g(q_{k+1}, x; \theta)). \quad (5.25)$$

When the Root Find Solver is simply the Newton method, predictions in E2E-SyMos are simply performing the insight given in algorithm 3.

²Here, we use the term gray-box due to the fact that the root-finding procedure computes the roots of a parameterized implicit equation that is physical constrained.

Backward Pass. The backward pass simply consists in applying the insight given by theorem 5.2.1 (figure 5.4). The gradients from equation (A.5) can then be used by any deep learning optimizer. For instance, a simple Stochastic Gradient Descent update performed on model parameters θ would be

$$\theta \leftarrow \theta - \alpha \cdot \frac{\partial \mathbb{L}}{\partial \theta} = \theta + \alpha \cdot \underbrace{\frac{\partial \mathbb{L}}{\partial q_{k+1}} \left[\frac{\partial \mathcal{L}_d(q_k, q_{k+1}; \theta)}{\partial q_{k+1} \partial q_k} + \frac{\partial f_d^-(q_k, q_{k+1}, u_k, u_{k+1})}{\partial q_{k+1}} \right]^{-1} \frac{\partial g(q_{k+1}, x; \theta)}{\partial \theta}}_{J_g}. \quad (5.26)$$

Note that this result is independent of the root-finding algorithm chosen or the internal structure of the implicit function g_θ . For instance, Newton's method requires the inverse Jacobian J_g at the root q_{k+1} . While using such method, the Jacobian needs to be computed in $\mathcal{O}(n^3)$, by automatic differentiation with a subsequent inversion in $\mathcal{O}(n^3)$. Both are computationally expensive operations, and for systems with a large number of degrees of freedom, training a neural network of this kind is intractable. In practice, Newton's method is barely used, and instead Quasi-Newton methods can be employed in $\mathcal{O}(n)$ without the need to explicitly compute J_g or its respective inverse. In that case approximations of J_g or J_g^{-1} can be used [46].

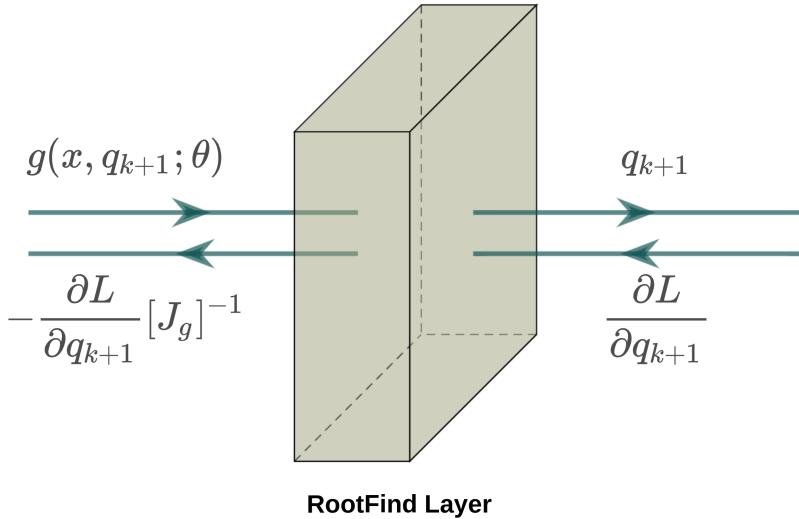


Figure 5.4: A general RootFind Layer. Instead of defining the output of the layer as the activation of layer, a root find layer takes as input a parameterized implicit function such as the DEL equations, and outputs the root of that implicit function. During back-propagation the Jacobian of the DEL equations, J_g , with respect to the root computed in the forward pass should be added to the computational graph.

5.2.3 Parameter Optimization

Assuming observations of the form of $(q_{k-1}, q_k, u_{k-1}, u_k, u_{k+1})$, and regression target q_{k+1} , the predictions are made by applying the root finding procedure,

$$\hat{q}_{k+1} = \text{RootFind}_{q_{k+1}}(g(x, q_{k+1}; \theta)). \quad (5.27)$$

Optimization consists in adjusting the free parameters θ by minimizing the mean squared error between the prediction and target:

$$\underset{\theta^*}{\text{minimize}} \quad \frac{1}{N} \sum_{i=1}^N \|\hat{q}_{k+1,i} - q_{k+1,i}\|_2^2 \quad (5.28)$$

Note that the addition of the implicit layer introduces two new learning hyperparameters defining the stopping criterion (algorithm 3), the tolerance, T and the maximum number of iterations $maxiter$. At inference the tolerance can be relaxed or, alternatively, the number of maximum iterations can be reduced.

Figure 5.5 shows the conceptual architecture of E2E-SyMos.

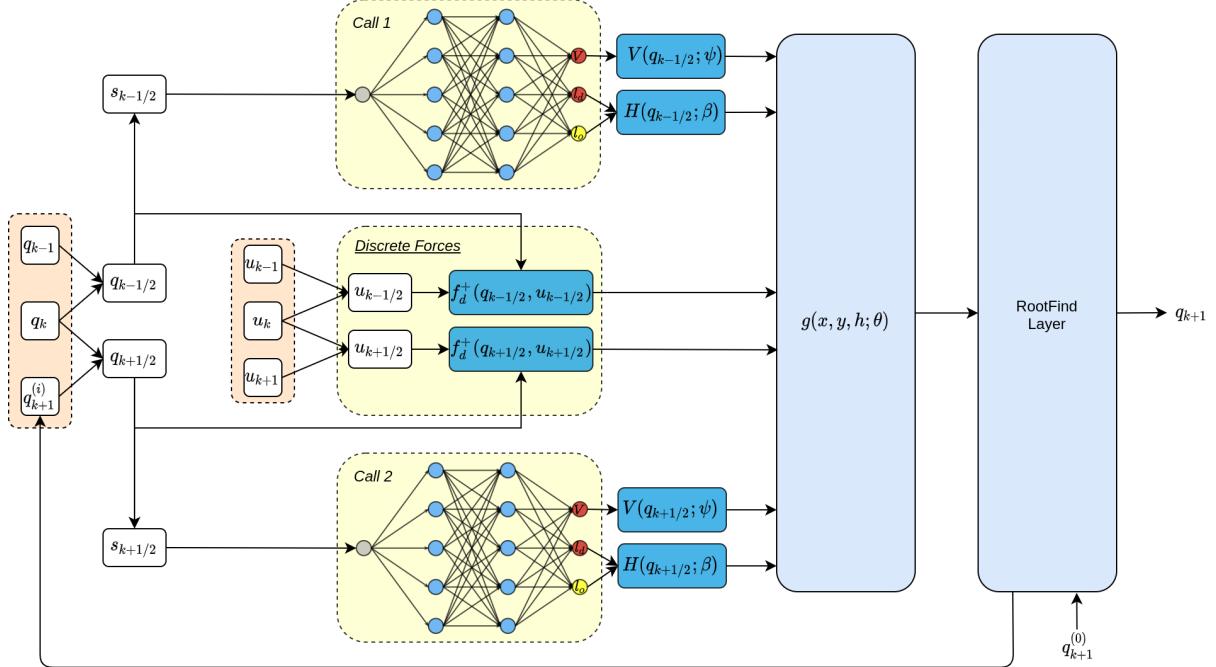


Figure 5.5: End-to-End Symplectic Momentum Neural Network with the embedding proposed. Given the input and output state, we form the collocation points (for $\alpha = 0.5$) and apply to them the embedding $s_{k\pm 1/2}$. Given the embeddings, we perform two calls to the neural network to get the inertia and potential energy correspondent to the two adjacent time-steps. These terms, alongside the discrete forces form then the Discrete Euler-Lagrange equations, $g(x, y, h; \theta)$ as a function of the learning parameters. The DEL are then used by the implicit layer defined by the root finding procedure. Given an initial estimate $q_{k+1}^{(0)}$ the root finding algorithm iterates over $g(x, y, h; \theta)$ to obtain \hat{q}_{k+1} .

Chapter 6

Results

With the models and baselines explained in the earlier chapters, we are now able to validate the models developed. We test them on three simulated robotic systems. Specifically the pendulum, the acrobot and the cartpole. One refers to annex B for the model's dynamics that govern these systems. We divide the results in three tasks, one for each robotic system. Results are shown in sections 6.4, 6.5 and 6.6.

6.1 Dataset Generation

The initial state is composed by the configurations and velocities $x_0 = [q, \dot{q}]$. We randomly generated initial conditions with a uniform distribution. The initial conditions are combined with a constant control input for each trajectory in a uniform distribution in the range of $u \in [-2, 2]$. The actuation is chosen to be constant to control the force discretization error within the SyMo models. Note that the left and right discrete forces (equation 5.19) depend on three adjacent time-steps and are approximated by the midpoint rule, inducing errors to the learning.

Based on the uniform sampled initial states, the ground truth trajectories are simulated by SciPy's *solve_ivp* adaptive solver¹ with method RK45 using Open AI Gym² [69]. Since OpenAI Gym favours other numerical integrators, such as the Euler integrator, and learning from inaccurate data is harder, we modify the environment to accommodate the RK45. In order to show that the methods that incorporate priors can learn from limited amount of data we vary the size of the training set by doubling from 8 to 128 the number of training trajectories. Since the training was stable for the pendulum, with larger time steps, then for this system, each trajectory, defined by each initial state condition, is integrated for 32 time steps with a time span of $h = 0.1$. For the cartpole and acrobot we use a smaller time step of $h = 0.05$.

¹https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html

²<https://gym.openai.com/>

6.2 Methods and Metrics

Table 6.1 shows an overview of the methods discussed in this thesis.

| Method | Section | Summary |
|----------|---------------|--|
| NODE | Section 4.3.3 | Geometric Neural Ordinary Differential Equation. |
| L-NODE | Section 4.3.4 | Combination of Neural Ordinary Differential Equations with the physical prior of the Euler-Lagrange equations. |
| SyMo | Section 5.1 | Learning based on the discrete Euler-Lagrange equations. Inference consists in solving the roots of the learned DEL. |
| E2E-SyMo | Section 5.2 | Development of an implicit layer that accommodates the root finding procedure necessary for solving the DEL equations. |

Table 6.1: Overview of the methods discussed.

For the NODE and L-NODE we choose the “RK4” and the midpoint³ as numerical integrators for training and making predictions. We use the labels NODE-RK4 and NODE-Midpoint to describe the models with black-box parameterizations of the system’s dynamics. Similarly, we denote by L-NODE-RK4 and L-NODE-Midpoint to describe the models that incorporate the Euler-Lagrangian equations within the learning framework, with the third word telling us the integrator used for training and inference.

To have a fair comparison with SyMos and because learning the L-NODEs and NODEs involves velocities, we set the train and test error as the Mean Squared Error (MSE) between the estimated configurations and the ground truth. The test loss is based on 128 test trajectories, where each is simulated for 32 time-steps. To evaluate the performance of each model in terms of long term prediction, we construct the metric of integration error per trajectory by using 16 random unseen initial state conditions without control actuation and integrating them for 500 time-steps, where the integration error corresponds to the difference between the ground truth and the predictions throughout the 500 time-steps. We logged the inertial loss for the models that make usage of physical priors through the inertia matrix, where the loss is defined by the MSE between the predicted inertia matrix and the ground truth. We also logged the energy loss per trajectory for the 16 trajectories. The reason for using only the unforced trajectories is that a constant nonzero control might cause the velocity to keep increasing or decreasing over time leading to large values in the velocities but also to study the conservation of energy in the models.

6.3 Training Details

For the NODEs a single neural network is used to parameterize the dynamics of the dynamical systems, as explained in section 4.3.3, where the activations are set to be the hyperbolic tangent for the hidden level and linear activation for the output layer. For the L-NODEs, SyMo and E2E-SyMo a single neural network topology is used, which outputs the elements of the lower triangular matrix \hat{L}_β and the potential energy \hat{V}_ψ . The diagonal elements of \hat{L}_β are enforced to be positive by using the Softplus

³<https://github.com/rtqichen/torchdiffeq>

activation and adding a small constant as described in section 4.1.2. The other activations are set to be linear except at the hidden level where we use the twice differentiable hyperbolic tangent. The neural network architecture consists of a two-hidden-layer MLP with 128 hidden neurons in each layer for the pendulum and 256 for the cartpole and acrobot. We initialize the weights with Xavier Uniform distribution [70] and we set the biases to zero. We set the size of mini-batches to be four times the number of initial state conditions. Implemented in PyTorch [71], we train our models using Adam optimizer [42] with 2000 epochs and with initial learning rate of 0.0001 for the pendulum, and 0.001 for the cartpole and acrobot. We use the *ReduceLROnPlateau* scheduler⁴ with patience 50 and factor 0.7. For the E2E-SyMo, for training and testing, we use a root finding tolerance of $1e - 5$ and a maximum number of 10 iterations. For integration we relax the tolerance to $1e - 4$. For the SyMos, after training, we perform the root finding operation to the learned DEL equations with the same hyperparameters.

6.4 Task 1 - Pendulum

6.4.1 Generalization Capabilities

In this subsection, we show the train, test and integration error as well as the predicted inertial error and energy for the integrated trajectories for the pendulum system. Models and parameter values are given in annex B.1.

Figure 6.1 shows the variation in train error, test error and integrator error with changes in the number of initial state conditions in the training set. We can see that the incorporation of prior knowledge generally yields better train and test losses, specially for a small number of trajectories. For instance, the NODE-RK4 requires more trajectories to achieve similar results with the SyMos and L-NODE-RK4. This means that the priors provide the models with a capability of controlling its variance. However, in terms of integration, it fails to keep up with those same models. One can see that the inclusion of priors

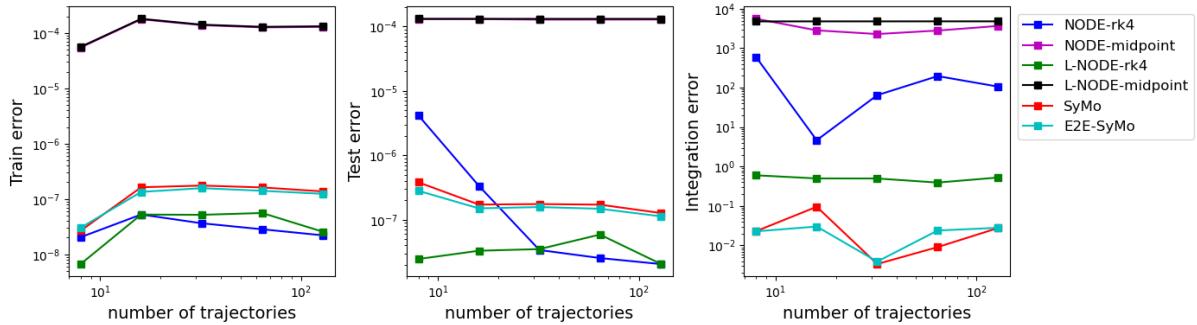


Figure 6.1: Train, test and integration loss for the pendulum.

works as regularizers to the learning. Note that the train and test errors are practically of the same order of magnitude, except for the models where there are no prior, i.e., the NODE-RK4, where the test loss is two orders of magnitude superior than the train loss for a low number of training trajectories which indicates overfitting.

⁴https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html

We can also see that the second-order accurate SyMos (SyMo and E2E-SyMo), despite having higher test and train losses, show better long term behaviour (integration loss) than the fourth-order accurate L-NODE-RK4. This is mainly due to the conservation of geometric properties characteristic from variational integrators, in opposition with the methods coming from continuous mechanics combined with traditional integrators that are not geometric.

The second-order L-NODE-Midpoint and NODE-Midpoint fail to give decent results for long term simulation (integration loss), despite the acceptable train and test losses. This could be explained by the fact that the errors tend to accumulate between time-steps leading to unacceptable long-term integration error. It is also possible to see that the inclusion of the root finding algorithm to the prior does not make a big difference for the pendulum system. The end-to-end SyMo presents only a light advantage on the train and test error.

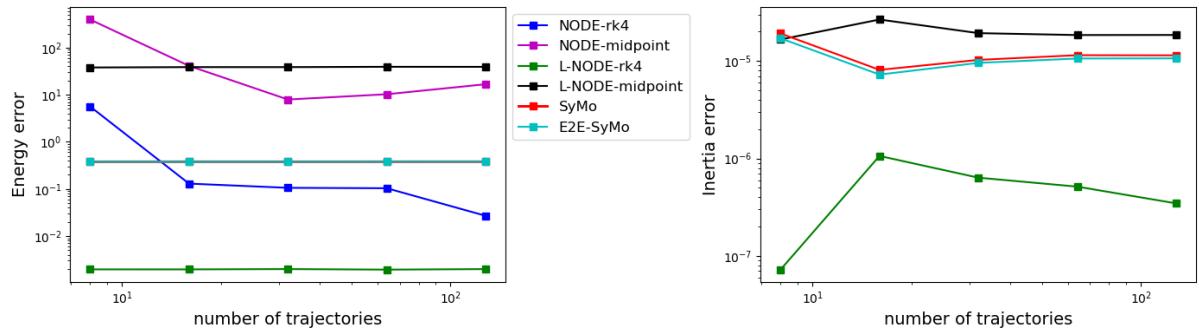


Figure 6.2: Energy and Inertia MSE for the pendulum.

Figure 6.2 shows the energy and inertial error for the integrated trajectories. Both are calculated through the data points provided by the integration trajectories. The L-NODE-RK4 is the method with best inertia and energy error. Still, SyMo and E2E-SyMo overcome the second order accurate counterparts NODE-Midpoint and L-NODE-Midpoint in terms of energy error.

Table 6.2 shows the losses for the models trained in a moderate data regime with 32 training trajectories. SyMo and E2E-SyMo have lowest integration error by two orders of magnitude when compared with the L-NODE-RK4.

| Model | Integrator | Train | Test | Integration | Energy | Inertia |
|----------|------------|-----------|-----------|-----------------------|----------------------|-----------|
| NODE | MP | $1.38e-4$ | $1.28e-4$ | $2.29e3 \pm 2.79e3$ | 7.92 ± 12.28 | N.A. |
| | RK4 | $3.64e-8$ | $3.39e-8$ | $64.55 \pm 1.23e2$ | $0.11 \pm 8.72e-2$ | N.A. |
| L-NODE | MP | $1.41e-4$ | $1.31e-4$ | $4.8e3 \pm 4.92e3$ | 38.6 ± 90.7 | $1.93e-5$ |
| | RK4 | $5.17e-8$ | $3.51e-8$ | 0.498 ± 2.3 | $2.0e-3 \pm 3.82e-3$ | $6.36e-7$ |
| SyMo | MP | $1.75e-7$ | $1.75e-7$ | $3.38e-3 \pm 5.04e-2$ | 0.378 ± 0.379 | $1.03e-5$ |
| E2E-SyMo | MP | $1.57e-7$ | $1.58e-7$ | $3.9e-3 \pm 4.98e-2$ | 0.378 ± 0.379 | $9.57e-6$ |

Table 6.2: Losses for the moderate data regime (32 training trajectories) measured in terms of mean squared error. The L-NODE-RK4 beats all the other models in every aspect except in the integration error, where SyMo and E2E-SyMo have lower mean squared error. MP stands for Midpoint.

6.4.2 Long Term Integration

In order to study better the long-term behaviour of the models, we pick the models trained with 32 trajectories and starting with a random initial condition we simulate them without actuation for 4000 time-steps. Figure 6.3 shows the trajectories for the simulated time-span. SyMo and E2E-SyMo are able to follow the ground truth with high level of accuracy. We can see that all models except SyMo and E2E-SyMo do not preserve the symplectic form as the simulation drifts away from the ground truth. This shows the influence of the geometric integrator used in SyMo and E2E-SyMo.

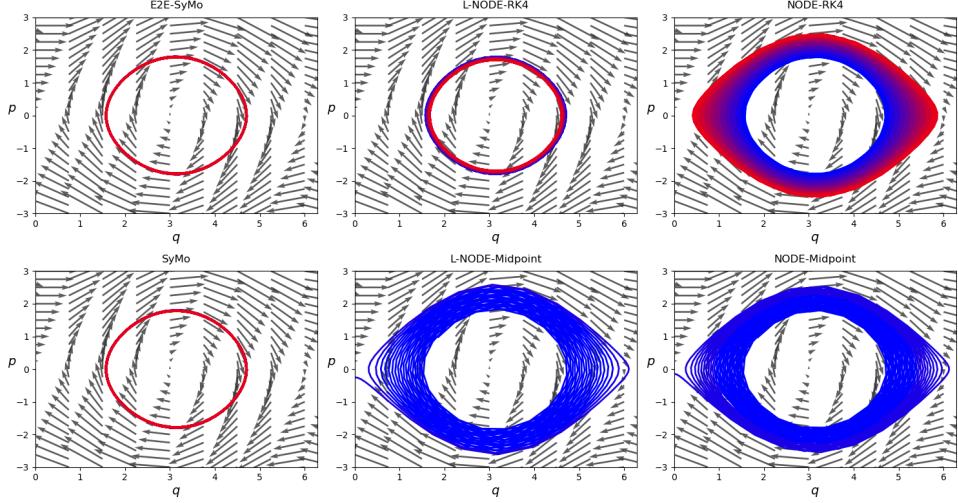


Figure 6.3: Phase Spaces for the Pendulum system. While the SyMos preserve the symplectic form we observe that the baseline model's dynamics moderately drift away from the circular trajectory (ground truth).

Simultaneously with the preservation of the symplectic form SyMo and E2E-SyMo bound the true energy (figure 6.4) and present a lower mean squared error during the first 50 seconds of simulation than the NODEs and L-NODE-Midpoint. The L-NODE-RK4 is able to conserve the energy during the first 50 seconds. Note that the non conservation of the symplectic form results in addition or dissipation of energy as well as a drifting in the trajectories from the ground truth.

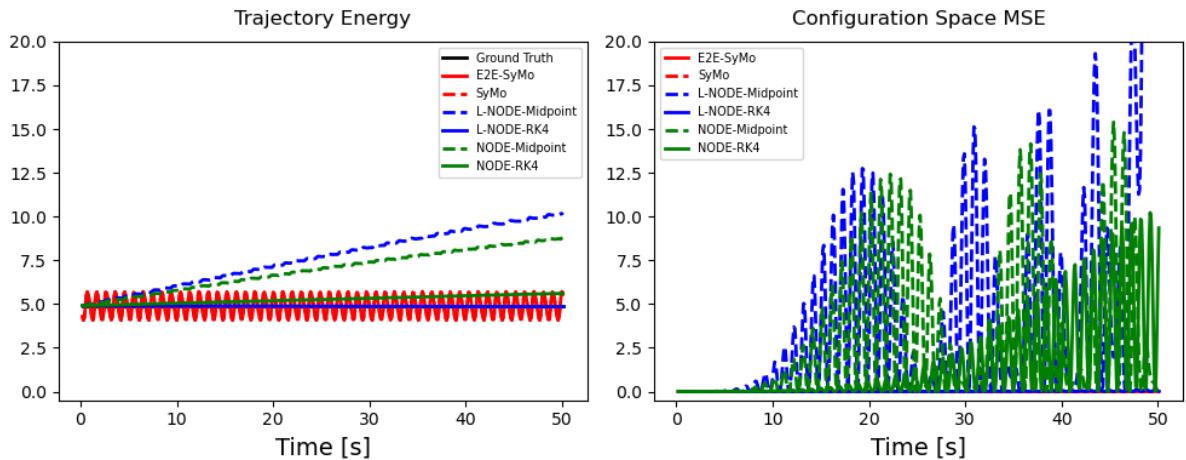


Figure 6.4: Energy and configuration space MSE during the first 50 seconds of the simulation.

6.4.3 Learned Quantities

Once we are modeling the system's energy through the inertia and potential energy for the SyMo, E2E-SyMo and L-NODEs, we show in figure 6.5 the learned quantities by the models with priors. All the models are able to learn approximately the exact inertia of the pendulum as well the kinetic energy. The potential energy learned differs from the ground truth with a constant. This is acceptable as the potential energy depends on the referential chosen. The L-NODE-Midpoint tends to drift from the ground truth with time showing once more worse behaviour when compared to its counterparts, second order accurate SyMo and E2E-SyMo.

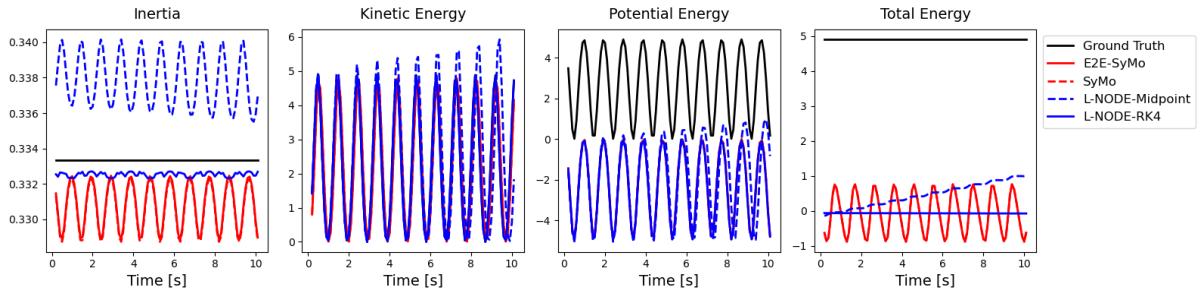


Figure 6.5: Learned quantities for the pendulum system for the test trajectory during the first 10 seconds of simulation.

6.4.4 Prediction with Forcing

In order to simulate real robotic systems, we simulate the models with a sinusoidal actuation

$$u = A \sin(2\pi t/T), \quad (6.1)$$

with $A=1$ and $T=10s$, during 20 seconds with a time-step of $h = 0.1$. We evaluate the modeling capabilities of all models, in figure 6.6, we show the trajectory due to the actuation and respective on-the-fly mean squared error. The L-NODE-midpoint and NODE-Midpoint are not able to follow the trajectory, showing a drift and high MSE with time. Both the SyMo, E2E-SyMo and L-NODE-RK4 are able to follow the trajectory and present a lower MSE.

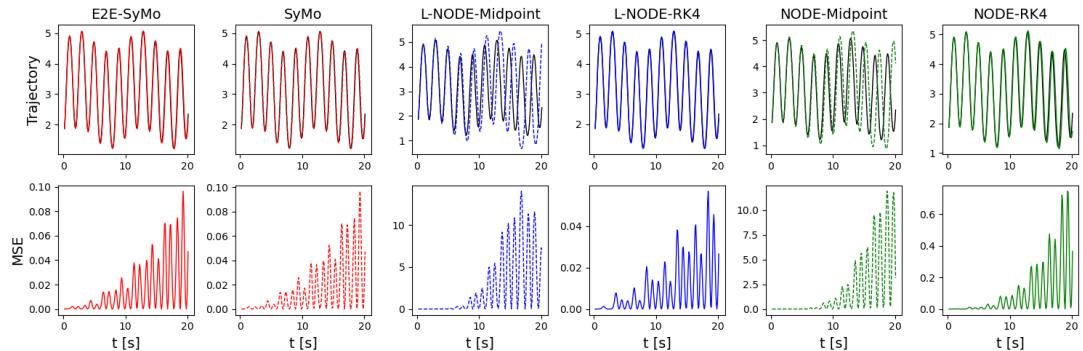


Figure 6.6: Trajectory and MSE for the test forced trajectory .

Figure 6.7 shows the system energy and inertia during the simulation time-span. The L-NODE-Midpoint and NODE-Midpoint are not able to follow the system's energy change through time, even though, similar to the others, they are able to follow the inertia accurately. Similar with the unforced test trajectory, the L-NODE-RK4 follows the true mass more accurately than the others. This is due to the fourth-order integrator used as a prior when compared to the second-order accurate counterparts.

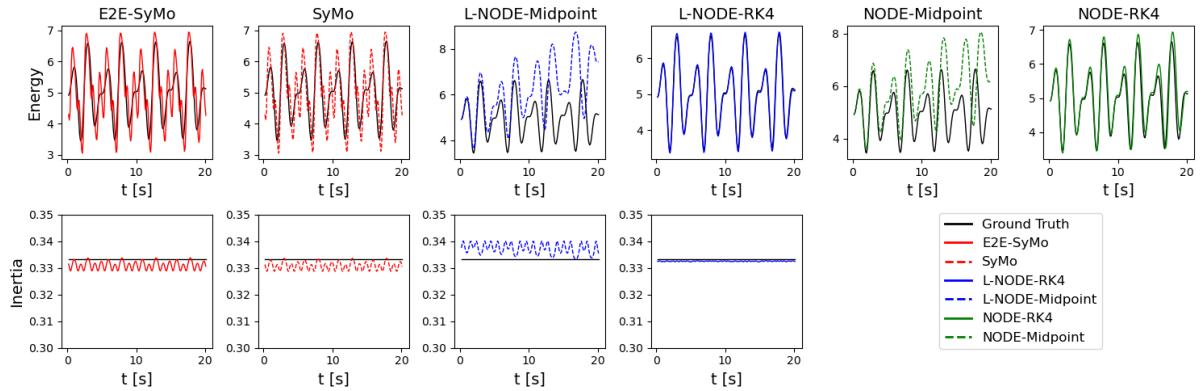


Figure 6.7: Energy and Inertia for the forced test trajectory

6.5 Task 2 - Acrobot

6.5.1 Generalization Capabilities

Figure 6.8 shows the overfitting of NODEs. Note that these models beat the remaining in terms of train loss but fail to achieve the same in terms of test loss, having a test loss orders of magnitude greater than the train loss. SyMo and E2E-SyMo have the best test and integration loss specially for a small number of training trajectories. E2E-SyMo presents the best test loss which emphasizes the effect of the implicit layer with extra prior knowledge. The integration errors are significant because the acrobot when unforced is a chaotic system. This means that a small change in the input will induce a significant change in the output. Yet, E2E-SyMo shows the smallest integration error for a small number of training trajectories.

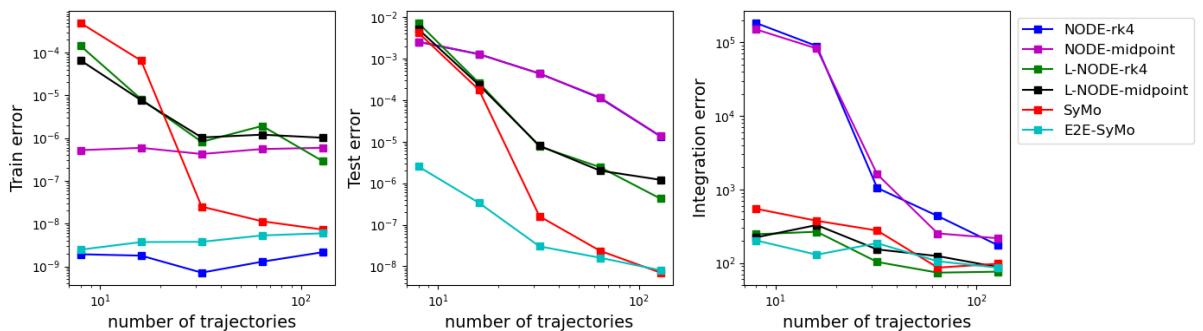


Figure 6.8: Train, Test and Integration error for the acrobot.

Figure 6.9 shows the energy and inertial error for the acrobot. All the models except the L-NODEs

fail to capture the true energy. This is due to the chaotic nature of the acrobot, making simulation harder. For a small number of training trajectories, E2E-SyMo show the best inertia modeling error.

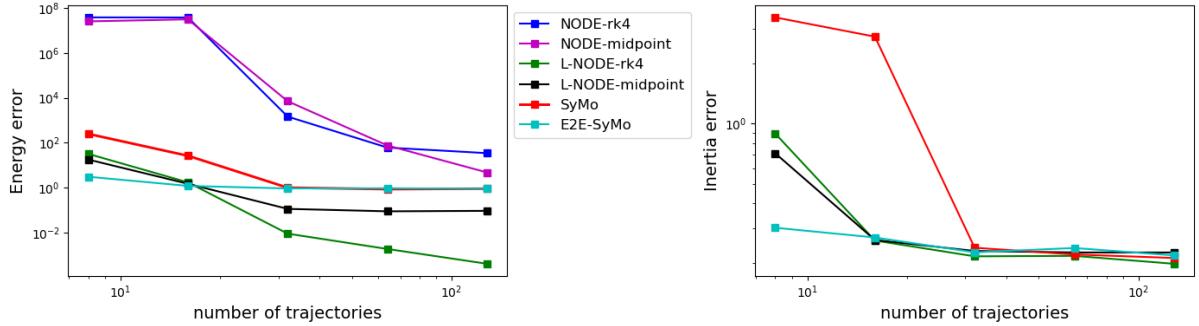


Figure 6.9: Energy and Inertial error for the acrobot.

Table 6.3 shows the losses for the models trained in a moderate data regime with 32 training trajectories. E2E-SyMo have lowest test loss by one order of magnitude when compared with SyMo. Unexpectedly, L-NODE-RK4 present the lowest integration, energy and inertia loss. One would expect lowest integration error for the SyMo and E2E-SyMo as the geometric priors used in those models provide the models with the capability of preserving important geometric structures about the continuous models. However, that does not happen for the chaotic unforced acrobot. Note that the methods without physical prior (NODEs) are subject to overfitting, showing train loss orders of magnitude smaller than the test loss.

| Model | Integrator | Train | Test | Integration | Energy | Inertia |
|----------|------------|------------------------------|-----------------------------|---------------------------------------|---|-------------|
| NODE | MP | $4.25e-7$ | $4.36e-4$ | $1.60e3 \pm 2.51e3$ | $7.04e3 \pm 2.63e4$ | N.A. |
| | RK4 | $7.14e-10$ | $4.39e-4$ | $1.05e3 \pm 1.54e3$ | $1.46e3 \pm 5.03e3$ | N.A. |
| L-NODE | MP | $1.04e-6$ | $8.0e-6$ | $1.53e2 \pm 2.55e2$ | 0.11 ± 0.128 | 0.23 |
| | RK4 | $8.16e-7$ | $7.7e-6$ | $1.03e2 \pm 1.06e2$ | $9.03e-3 \pm 1.16e-2$ | 0.22 |
| SyMo | MP | $2.50e-8$ | $1.6e-7$ | $2.76e2 \pm 6.83e2$ | 1.00 ± 0.75 | 0.24 |
| E2E-SyMo | MP | $3.75e-9$ | $3.03e-8$ | $1.85e2 \pm 3.07e2$ | 0.93 ± 0.708 | 0.227 |

Table 6.3: Losses for the moderate data regime (32 training trajectories) measured in terms of mean squared error. The L-NODE-RK4 beats all the other models in every aspect except in the train and test loss, where SyMo and E2E-SyMo have lower test loss. MP stands for Midpoint.

6.5.2 Test Trajectory

In Figure 6.10 we simulate the models for 500 time-steps without actuation. The figure shows the resulting trajectories. NODEs are not able to follow the ground truth accurately, showing a drift with time, while the others match the ground truth.

Figure 6.11 shows the energy and mean squared error with time for the simulated test trajectory. NODEs rapidly drift away from the true energy while SyMo, E2E-SyMo and the L-NODEs bound the true energy. This shows the effect of the physical priors in terms of conservation of properties. The NODEs,

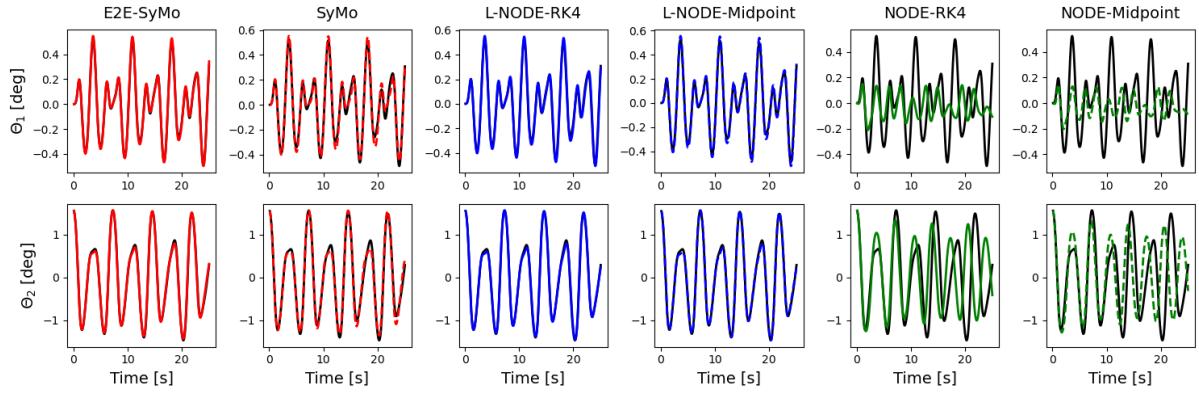


Figure 6.10: Trajectories for the test trajectory for the acrobot.

that make no usage of the physics as a prior, are not able to conserve the energy, while the remaining models, dependent on physical priors, bound the true energy.

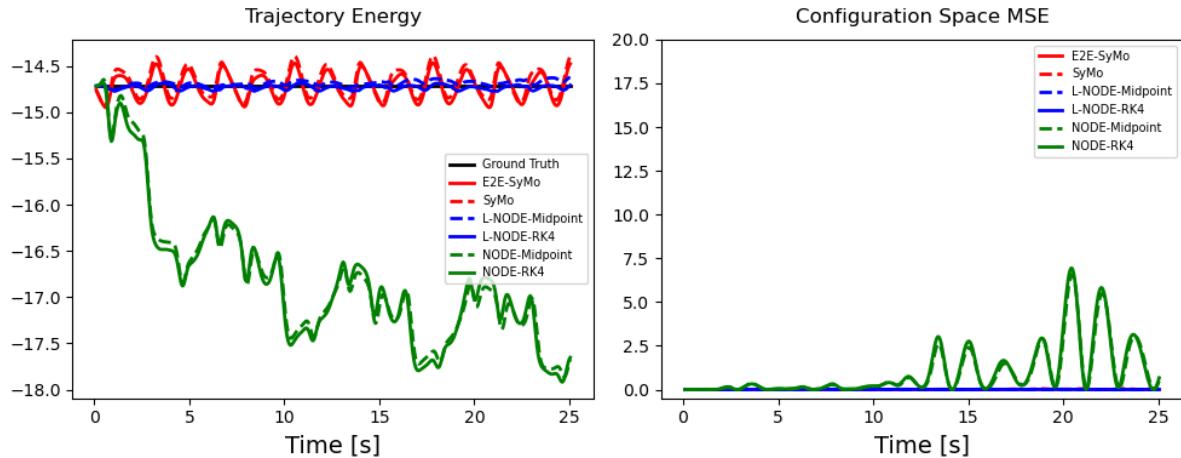


Figure 6.11: Energy and MSE for the test trajectory for the acrobot.

Figure 6.12 shows the learned quantities for the acrobot. All models are able to learn the true kinetic energy. The potential energy, once more, differs by a constant and so does the total energy.

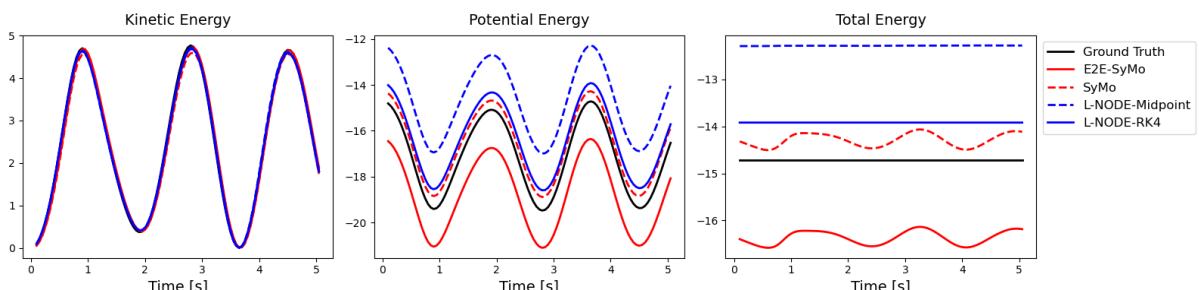


Figure 6.12: Learned quantities for the acrobot.

6.6 Task 3 - Cartpole

6.6.1 Generalization Capabilities

Figure 6.13 shows the train, test and integration loss for the cartpole. SyMo and E2E-SyMo show the best test and integration loss. This emphasizes the incorporation of geometric integration within the learning framework. Even though the integrators are less accurate by two orders of magnitude than the integrator used in NODE-RK4 and L-NODE-RK4, still E2E-SyMo and SyMo perform better in terms of long term integration. Surprisingly, NODE-RK4 and NODE-Midpoint perform better than L-NODE-RK4 and L-NODE-Midpoint in the test loss but fail to keep the same consistency in the integration loss. One can also see that the models without priors (NODEs) have a very low train loss but achieve test loss orders of magnitude greater. This shows how easily models without a prior are subject to overfitting.

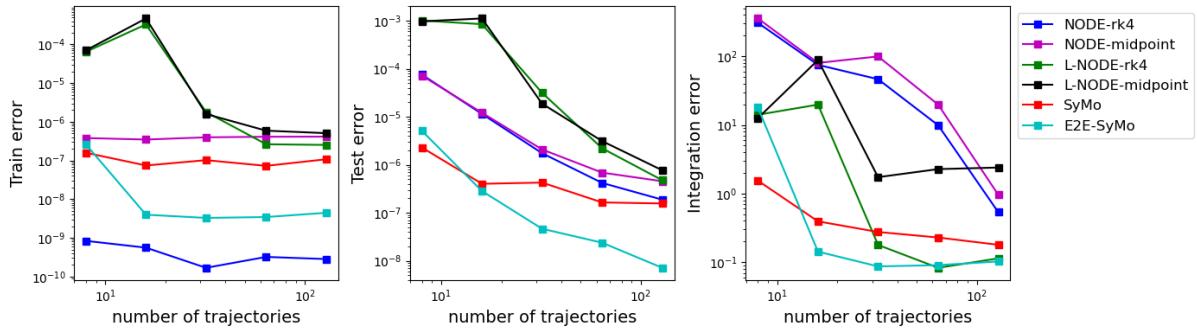


Figure 6.13: Train, test and integration loss for the cartpole.

In figure 6.14, E2E-SyMo not only show a good long term behaviour in terms of preservation of energy but also outperform the other in terms of modeling the inertia matrix. Showing that these models are capable of inferring the true inertial parameters with an high degree of accuracy. NODE-Midpoint and NODE-RK4 fail to preserve the energy specially for a small number of trajectories. This is normal as the absence of priors do not provide the models with the capability of preserving important physical properties, such as energy.

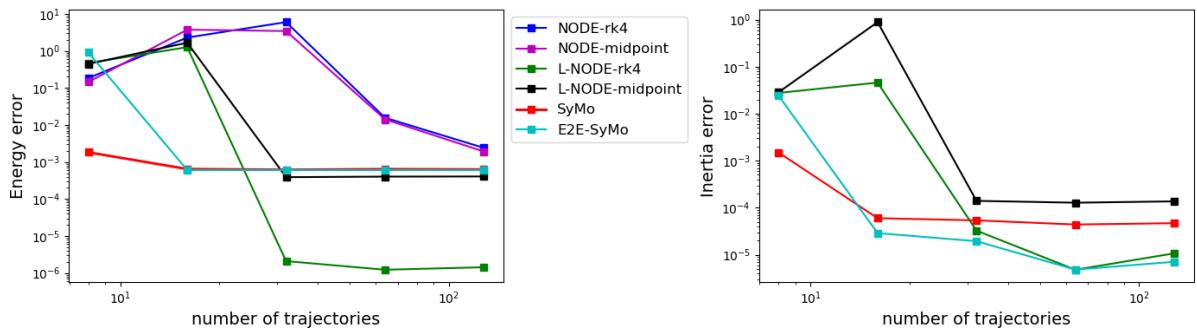


Figure 6.14: Energy and Inertial Error for the cartpole.

Table 6.4 shows the losses for the models trained in a moderate data regime with 32 training trajectories. E2E-SyMo outperform the remainder in terms of test, integration and inertial loss. It is also

possible to see, once more, the overfitting of NODEs once the train loss is orders of magnitude smaller than the test loss.

| Model | Integrator | Train | Test | Integration | Energy | Inertia |
|--------------|-------------------|------------------------------|-----------------------------|---|--|-----------------------------|
| NODE | MP | $3.97e-7$ | $2.09e-6$ | 99.5 ± 372.2 | 3.45 ± 12.2 | N.A. |
| | RK4 | $1.68e-10$ | $1.74e-6$ | $46.4 \pm 1.58e2$ | 6.0 ± 21.2 | N.A. |
| L-NODE | MP | $1.59e-6$ | $1.88e-5$ | 1.73 ± 4.76 | $3.89e-4 \pm 2.57e-4$ | $1.4e-4$ |
| | RK4 | $1.79e-6$ | $3.16e-5$ | 0.179 ± 0.174 | $2.1e-6 \pm 2.99e-6$ | $3.26e-5$ |
| SyMo | MP | $1.02e-7$ | $4.25e-7$ | $2.76e-1 \pm 5.05e-1$ | $6.11e-4 \pm 6.09e-4$ | $5.38e-5$ |
| E2E-SyMo | MP | $3.28e-9$ | $4.64e-8$ | $8.69e-2 \pm 1.31e-1$ | $6.05e-4 \pm 5.84e-4$ | $1.94e-5$ |

Table 6.4: Losses for the moderate data regime (32 training trajectories) measured in terms of mean squared error. E2E-SyMo show the better test, integration and inertial loss. MP stands for Midpoint.

6.6.2 Test Trajectory

We simulate the models for a test trajectory without actuation during 500 time-steps. Figure 6.15 shows the resulting trajectories. Only E2E-SyMo is capable to follow the ground truth with high accuracy, for the translational coordinate. For the rotational coordinate of the cartpole, all methods are able to follow the ground truth.

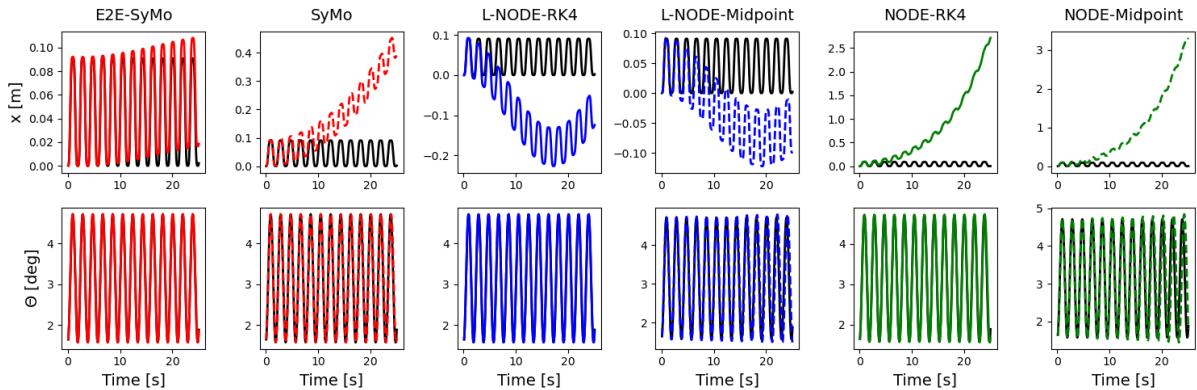


Figure 6.15: Trajectories for the cartpole. The E2E-SyMo are capable of following precisely the ground truth.

Figure 6.16 shows the energy and configuration space MSE for the test trajectory. NODEs and L-NODE-Midpoint fail to preserve the energy with time. Those same models also present a high mean squared error with time. E2E-SyMo and SyMo bound the true energy. Once more, the models without a physical prior (NODEs) fail to bound the true energy, showing a drift from the true energy with time. This emphasizes the importance of the physical priors in terms of conservation of properties, such as energy.

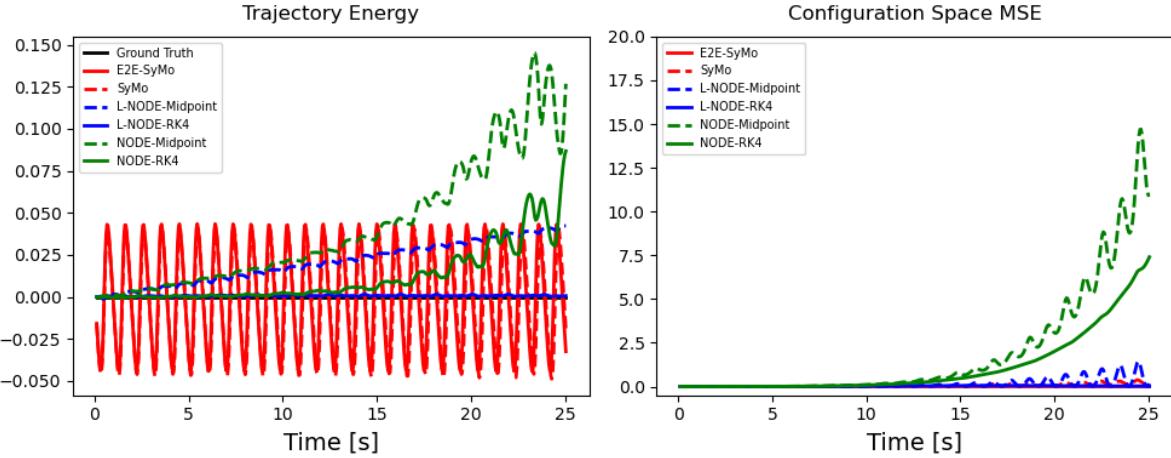


Figure 6.16: Energy and MSE for the test trajectory cartpole.

Figure 6.17 shows the learned quantities for the cartpole during the first 5 seconds of simulation. All models are capable to model the kinetic energy accurately. The potential energies differ only by a constant which is normal as the potential energy depends on the referential. The total energy also differs from a constant as it is the sum of the kinetic with potential energy. E2E-SyMo and SyMo show the expected energy bounding behaviour characteristic from variational integrators.

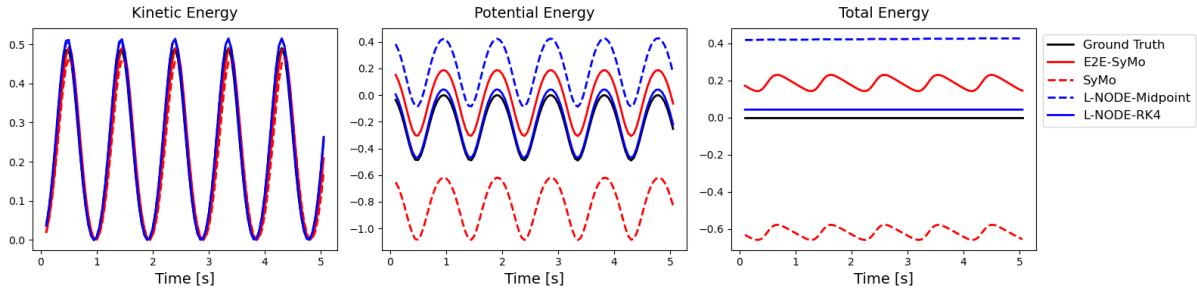


Figure 6.17: Learned Quantities for the cartpole.

6.6.3 Prediction with Forcing

We simulate the models with a sinusoidal actuation with $T = 2s$ and $A = 1$, figure 6.18 shows the resulting trajectories, mean squared error and energy. NODEs and L-NODE-Midpoint rapidly drift away from the ground truth, present a higher on-the-fly mean squared error and are not able to follow the energy change. SyMo shows the best mean squared error, outperforming E2E-SyMo and L-NODE-RK4.

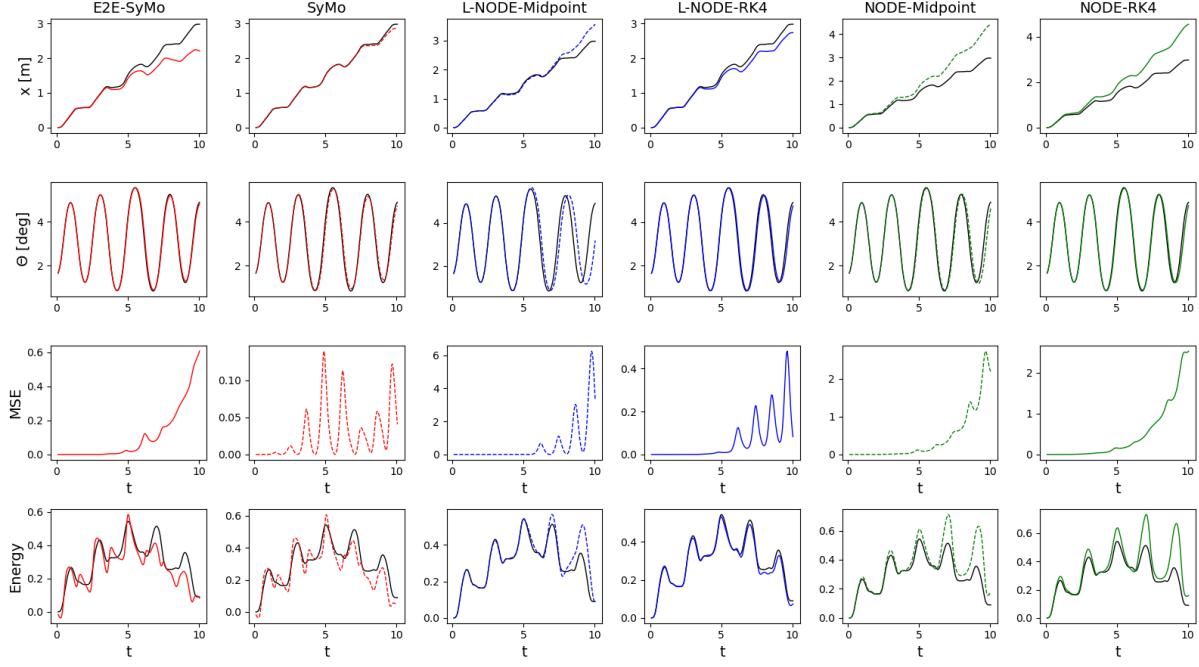


Figure 6.18: Forced trajectories for the cartpole.

6.6.4 Training with noise

We consider our models in the presence of measurement noise. Specifically, the model is given noisy position and velocity measurements as training data from which they need to learn the dynamics. We constructed training trajectories of 32 trajectories with 32 data points each and added Gaussian noise with standard deviations, σ , of $[0, 0.0001, 0.0005, 0.001, 0.005, 0.01]$ in the positions and velocities to every data point. We train the models with the same setup from the models trained without noise.

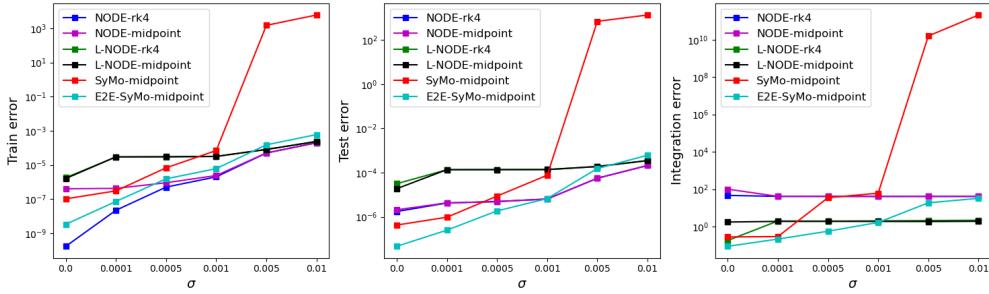


Figure 6.19: Train, Test and Integration Loss for the noisy training data.

Figure 6.19 shows the losses for the noisy datasets. E2E-SyMo outperforms the remaining models in terms of test and integration loss for regimes with low noise. SyMo fail to learn the dynamics for high levels of noise, having high train, test and integration losses. Note that SyMo is matching the implicit discrete Euler-Lagrange equations while the others are matching either the poses or the poses and velocities. This behaviour does not occur with E2E-SyMo as we are matching the poses with the ground truth. This shows the benefit of using an implicit layer with extra prior knowledge for noisy datasets. For

high levels of noise E2E-SyMo are outperformed by the NODEs and L-NODEs which shows that these models are less robust to high levels of noise.

Chapter 7

Conclusions

In this work we introduced Symplectic Momentum Neural Networks (SyMo) as models that combine traditional deep learning techniques with a discrete formulation of mechanics, leading to geometric integration, that preserves physical invariants about the original equations of motion. We extend this combination to account for variational integration by means of creating an implicit layer that solves a root finding algorithm, leading to End-To-End Symplectic Momentum Neural Networks (E2E-SyMo). These models allow to learn Lagrangian dynamics only with the poses as training data, avoiding potential noisy velocity measurements as opposed to the models arriving from continuous mechanics that require not only the poses but also the velocities.

On the continuous side of mechanics we use neural networks to parameterize the dynamics in a black-box way, learning from discrete data and using Neural ODEs with traditional integrators such as the Runge Kutta 4th-order and the midpoint. We extend this to incorporate the continuous Lagrangian mechanics typical from mechanical systems within the learning framework by using the resulting Euler-Lagrange equations as a model prior, with the same integrators.

We compare the short and long-term behaviour of the models arising from continuous mechanics, with and without prior knowledge about the underlying equations of motion, with the ones coming from discrete mechanics. On one side, we have continuous mechanics with models with an implicit layer that solves an ODE between two adjacent time steps. On the other side, we have discrete mechanics where E2E-SyMo make use of an implicit layer that solves a root finding problem. We test our models in three simulated robotic systems, the pendulum, the acrobot and the cartpole.

Results show that the priors have desirable effects in the long-term behaviour. Specifically, the models with geometric integrators as priors show better long term behaviour, achieving lower integration losses. E2E-SyMo and SyMo (second order accurate) even outperform the fourth-order accurate L-NODE-RK4, for the pendulum and cartpole, in terms of integration loss which shows the importance of preservation of geometric properties while discretizing. For the cartpole, we can see the effect of the implicit layer by adding a prior on the jacobian of the discrete Euler-Lagrange equations since E2E-SyMo outperform all the other models in terms of test, integration and inertial error. SyMo and E2E-SyMo present the energy bounding behaviour as expected from the discrete formulation of mechanics while

NODEs drift away rapidly from the true energy. Results also show that the usage of priors provide the models with the capability of bounding the true energy and training with less data.

7.1 Future Work

The work we have carried out opens up interesting possibilities of future work, both in terms of theoretical developments and more efficient implementations.

Inclusion of dissipation terms. Real life physical systems often lose energy in a structured way. For instance frictional losses in robotic systems. One can add to the developed approaches dissipation that can be included in the learning process or exploited with it.

Using Quasi-Newton methods. While the Newton's method used for E2E-SyMo typically experiences very fast convergence, it is also expensive to have to recompute the jacobian $\nabla g(x_n)$ at each iteration of the method. For instance, the complexity of the jacobian operation is $\mathcal{O}(n^3)$ which when coupled with traditional root-finders, such as the Newton's method, that require the inverses of the Jacobian, this adds an approximately $\mathcal{O}(n^3)$ complexity for matrix inversion [72]. Quasi-Newton methods such as the Broyden that approximate the inverse of the jacobian can be used with $\mathcal{O}(n)$ complexity. Note that the RootFind layer is independent of the method chosen to solve the roots.

Lie Group Variational Integrator. Another research direction would be to make use of the implicit differentiation definitions and explore them with a possible combination of the exceptionally efficient Lie Group Variational Integrator [73] with deep learning.

Bibliography

- [1] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60:84 – 90, 2012.
- [2] T. Young, D. Hazarika, S. Poria, and E. Cambria. Recent trends in deep learning based natural language processing. *IEEE Computational Intelligence Magazine*, 13(3):55–75, 2018. doi: 10.1109/mci.2018.2840738.
- [3] O. Sigaud, C. Salan, and V. Padois. On-line regression algorithms for learning mechanical models of robots: A survey. *Robotics and Autonomous Systems*, 59(12):1115–1129, 2011. ISSN 09218890. doi: 10.1016/j.robot.2011.07.006.
- [4] J. K. Gupta, K. Menda, Z. Manchester, and M. J. Kochenderfer. A general framework for structured learning of mechanical systems. *arXiv preprint arXiv:1902.08705*, 2019.
- [5] M. Lutter, C. Ritter, and J. Peters. Deep Lagrangian networks: Using physics as model prior for deep learning. *7th International Conference on Learning Representations, ICLR 2019*, pages 1–17, 2019.
- [6] J. K. Gupta, K. Menda, Z. Manchester, and M. J. Kochenderfer. Structured Mechanical Models for Robot Learning and Control. *2nd Annual Conference on Learning for Dynamics and Control*, 2020. URL <http://arxiv.org/abs/2004.10301>.
- [7] S. Greydanus, M. Dzamba, and J. Yosinski. Hamiltonian neural networks. *Advances in Neural Information Processing Systems*, 32:1–16, 2019. ISSN 10495258.
- [8] Z. Chen, J. Zhang, M. Arjovsky, and L. Bottou. Symplectic recurrent neural networks. *International Conference on Learning Representations*, 2020. URL <http://arxiv.org/abs/1909.13334>.
- [9] J. Willard, X. Jia, S. Xu, M. S. Steinbach, and V. Kumar. Integrating physics-based modeling with machine learning: A survey. *arXiv preprint arXiv:2003.04919*, 2020.
- [10] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations. *arXiv preprint arXiv:1711.10566*, 2017.
- [11] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differen-

- tial equations. *Journal of Computational Physics*, 378:686–707, 2019. ISSN 10902716. URL <https://doi.org/10.1016/j.jcp.2018.10.045>.
- [12] M. Raissi. Deep hidden physics models: Deep learning of nonlinear partial differential equations. *arXiv preprint arXiv:1801.06637*, 2018.
- [13] M. Lutter and J. Peters. Differential equations as a model prior for deep learning and its applications in robotics. In *ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations*, 2020. URL https://openreview.net/forum?id=_uPd3skTsj.
- [14] M. Cranmer, S. Greydanus, S. Hoyer, P. Battaglia, D. Spergel, and S. Ho. Lagrangian Neural Networks. *arXiv preprint arXiv:2003.04630*, 2020.
- [15] Y. D. Zhong, B. Dey, and A. Chakraborty. Symplectic ODE-Net: Learning Hamiltonian Dynamics with Control. *International Conference on Learning Representations*, 2020. URL <http://arxiv.org/abs/1909.12077>.
- [16] A. Sanchez-Gonzalez, V. Bapst, K. Cranmer, and P. Battaglia. Hamiltonian Graph Networks with ODE Integrators. *arXiv preprint arXiv:1909.12790*, 2019.
- [17] E. Hairer, C. Lubich, and G. Wanner. *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations; 2nd ed.* Springer, Dordrecht, 2006. doi: 10.1007/3-540-30666-8.
- [18] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud. Neural ordinary differential equations. *Advances in Neural Information Processing Systems*, 2018. URL <http://arxiv.org/abs/1806.07366>.
- [19] J. E. Marsden and M. West. Discrete mechanics and variational integrators. *Acta Numerica*, 10: 357–514, 2001. doi: 10.1017/s096249290100006x.
- [20] S. Saemundsson, A. Terenin, K. Hofmann, and M. P. Deisenroth. Variational Integrator Networks for Physically Structured Embeddings. *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2020. URL <http://arxiv.org/abs/1910.09349>.
- [21] S. Ober-Blöbaum, O. Junge, and J. E. Marsden. Discrete mechanics and optimal control: An analysis. *ESAIM - Control, Optimisation and Calculus of Variations*, 17(2):322–352, 2011. ISSN 12928119. doi: 10.1051/cocv/2010012.
- [22] Z. Bing, C. Meschede, F. Röhrbein, K. Huang, and A. C. Knoll. A survey of robotics control based on learning-inspired spiking neural networks. *Frontiers in Neurorobotics*, 12, 2018. doi: 10.3389/fnbot.2018.00035.
- [23] D. Nguyen-Tuong and J. Peters. *Model learning for robot control: A survey*, volume 12. Cognitive Processing, 2011. doi: 10.1007/s10339-011-0404-1.

- [24] J. Kocijan, R. Murray-Smith, C. E. Rasmussen, and A. Girard. Gaussian process model based predictive control. *Proceedings of the American Control Conference*, 3:2214–2219, 2004. doi: 10.23919/acc.2004.1383790.
- [25] L. Hewing, A. Liniger, and M. N. Zeilinger. Cautious NMPC with Gaussian Process Dynamics for Autonomous Miniature Race Cars. *2018 European Control Conference, ECC 2018*, pages 1341–1348, 2018. doi: 10.23919/ECC.2018.8550162.
- [26] L. Hewing, J. Kabzan, and M. N. Zeilinger. Cautious Model Predictive Control Using Gaussian Process Regression. *IEEE Transactions on Control Systems Technology*, pages 1–12, 2019. ISSN 15580865. doi: 10.1109/TCST.2019.2949757.
- [27] A. Carron, E. Arcari, M. Wermelinger, L. Hewing, M. Hutter, and M. N. Zeilinger. Data-Driven Model Predictive Control for Trajectory Tracking With a Robotic Arm. *IEEE Robotics and Automation Letters*, 4(4):3758–3765, 2019. ISSN 2377-3766. doi: 10.1109/lra.2019.2929987.
- [28] C. J. Ostafew, A. P. Schoellig, T. D. Barfoot, and J. Collier. Learning-based Nonlinear Model Predictive Control to Improve Vision-based Mobile Robot Path Tracking. *Journal of Field Robotics*, 33(1): 133–152, 2016. ISSN 15564967. doi: 10.1002/rob.21587.
- [29] D. Nguyen-Tuong and J. Peters. Local Gaussian process regression for real-time model-based robot control. *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS*, pages 380–385, 2008. doi: 10.1109/IROS.2008.4650850.
- [30] I. Lenz, R. Knepper, and A. Saxena. DeepMPC: Learning deep latent features for model predictive control. *Robotics: Science and Systems*, 11, 2015. ISSN 2330765X. doi: 10.15607/RSS.2015.XI.012.
- [31] M. T. Gillespie, C. M. Best, E. C. Townsend, D. Wingate, and M. D. Killpack. Learning non-linear dynamic models of soft robots for model predictive control with neural networks. In *2018 IEEE International Conference on Soft Robotics (RoboSoft)*, pages 39–45, 2018. doi: 10.1109/ROBOSOFT.2018.8404894.
- [32] S. Santos, M. Ekal, and R. Ventura. Symplectic momentum neural networks - using discrete variational mechanics as a prior in deep learning. *arXiv preprint arXiv:2201.08281*, 2022.
- [33] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [34] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.
- [35] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989. doi: 10.1007/bf02551274.

- [36] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993. doi: 10.1016/s0893-6080(05)80131-5.
- [37] D. Yarotsky. Universal approximations of invariant maps by neural networks. *arxiv preprint arXiv:1804.10306*, 2018.
- [38] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. doi: 10.1016/0893-6080(89)90020-8.
- [39] K. Hornik, M. Stinchcombe, and H. White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3(5):551–560, 1990. doi: 10.1016/0893-6080(90)90005-6.
- [40] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research* 18, 2018. URL <https://arxiv.org/abs/1502.05767>.
- [41] D. Masters and C. Luschi. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*, 2018.
- [42] D. P. Kingma and J. L. Ba. Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pages 1–15, 2015.
- [43] L. E. Ghaoui, F. Gu, B. Travacca, and A. Askari. Implicit deep learning. *arXiv preprint arXiv:1908.06315*, 2019.
- [44] B. Amos and J. Z. Kolter. OptNet: Differentiable optimization as a layer in neural networks. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 136–145. PMLR, 2017.
- [45] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and Z. Kolter. Differentiable convex optimization layers. In *Advances in Neural Information Processing Systems*, 2019.
- [46] S. Bai, J. Z. Kolter, and V. Koltun. Deep equilibrium models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.
- [47] S. Bai, V. Koltun, and J. Z. Kolter. Multiscale deep equilibrium models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
- [48] R. M. Murray, Z. Li, and S. S. Sastry. A mathematical introduction to robotic manipulation. CRC Press, 1994.
- [49] C. Runge. Ueber die numerische auflosung von von differentialgleichungen. *Mathematische Annalen*, 46(2):167–178, 1895. doi: 10.1007/bf01446807.

- [50] E. Hairer, S. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I Nonstiff problems*. Springer, 1987.
- [51] M. West. *Variational Integrators*. PhD thesis, California Institute of Technology, Pasadena, California, June 2003. URL <https://thesis.library.caltech.edu/2492/>.
- [52] G. Zhong and J. E. Marsden. Lie-poisson hamilton-jacobi theory and lie-poisson integrators. *Physics Letters A*, 133(3):134–139, 1988. doi: 10.1016/0375-9601(88)90773-6.
- [53] S. Xiong, Y. Tong, X. He, C. Yang, S. Yang, and B. Zhu. Nonseparable symplectic neural networks. *arXiv preprint arXiv:2010.12636*, 2020.
- [54] J. Lee. *Introduction to Topological Manifolds*. Graduate texts in mathematics. Springer, 2000. ISBN 9780387950266.
- [55] R. I. McLachlan and G. R. W. Quispel. Geometric integrators for odes. *Journal of Physics A: Mathematical and General*, 39(19):5251–5285, 2006. doi: 10.1088/0305-4470/39/19/s01.
- [56] R. C. Fetecau, J. E. Marsden, M. Ortiz, and M. West. Nonsmooth lagrangian mechanics and variational collision integrators. *SIAM Journal on Applied Dynamical Systems*, 2(3):381–416, 2003. doi: 10.1137/s1111111102406038.
- [57] Z. Manchester and S. Kuindersma. Variational contact-implicit trajectory optimization. *Springer Proceedings in Advanced Robotics Robotics Research*, page 985–1000, 2019. doi: 10.1007/978-3-030-28619-4_66.
- [58] T. D. M. Jarvis Schultz, Elliot Johnson. Trajectory Optimization in Discrete Mechanics. *Differential-Geometric Methods in Computational Multibody System Dynamics*. Springer International Publishing, 2015.
- [59] Z. Shareef and A. Trachtler. Optimal trajectory planning for robotic manipulators using Discrete Mechanics and Optimal Control. *2014 IEEE Conference on Control Applications, CCA 2014*, pages 240–245, 2014. doi: 10.1109/CCA.2014.6981358.
- [60] T. M. Caldwell, D. Coleman, and N. Correll. Optimal parameter identification for discrete mechanical systems with application to flexible object manipulation. *IEEE International Conference on Intelligent Robots and Systems*, 2014. ISSN 21530866. doi: 10.1109/IROS.2014.6942666.
- [61] M. Ekal and R. Ventura. A dual quaternion-based discrete variational approach for accurate and online inertial parameter estimation in free-flying robots. *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020.
- [62] T. Matsubara, A. Ishikawa, and T. Yaguchi. Deep energy-based modeling of discrete-time physics. In *Advances in Neural Information Processing Systems 33 (NeurIPS2020)*, 2020. URL <https://arxiv.org/abs/1905.08604>.

- [63] Z. Long, Y. Lu, X. Ma, and B. Dong. Pde-net: Learning pdes from data. In *International Conference on Machine Learning*, pages 3214–3222, 2018.
- [64] V. B. Lev Semenovich Pontryagin, EF Mishchenko and R. Gamkrelidze. *Mathematical Theory of Optimal Processes*. Wiley, New York, 1962.
- [65] E. Dupont, A. Doucet, and Y. W. Teh. Augmented neural odes. *arXiv preprint arXiv:1904.01681*, 2019.
- [66] S. Massaroli, M. Poli, J. Park, A. Yamashita, and H. Asama. Dissecting neural odes. In *Advances in Neural Information Processing Systems*, volume 33, pages 3952–3963. Curran Associates, Inc., 2020. URL <https://proceedings.neurips.cc/paper/2020/file/293835c2cc75b585649498ee74b395f5-Paper.pdf>.
- [67] Y. D. Zhong, B. Dey, and A. Chakraborty. Dissipative symoden: Encoding hamiltonian dynamics with dissipation and control into deep learning. *ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations*, 2020.
- [68] B. Amos, L. Xu, and J. Z. Kolter. Input convex neural networks. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 146–155. PMLR, 2017.
- [69] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.
- [70] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh and M. Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256. PMLR, 2010. URL <https://proceedings.mlr.press/v9/glorot10a.html>.
- [71] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.
- [72] J. Lee, C. K. Liu, F. C. Park, and S. S. Srinivasa. A linear-time variational integrator for multibody systems. In *International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2016.
- [73] T. Lee, M. Leok, and N. H. McClamroch. Lie group variational integrators for the full body problem in orbital mechanics. *Celestial Mechanics and Dynamical Astronomy*, 98(2):121–144, 2007. doi: 10.1007/s10569-007-9073-x.
- [74] S. Krantz and H. Parks. *The Implicit Function Theorem: History, Theory, and Applications*. Modern Birkhäuser classics. Birkhäuser, Boston, 2002.

- [75] Z. Kolter, D. Duvenaud, and M. Johnson. Deep implicit layers - neural odes, deep equilibrium models, and beyond. NeurIPS 2020 tutorial. URL <http://implicit-layers-tutorial.org/>.
- [76] R. Sutton and A. G. Barto. Reinforcement learning: An introduction. *Massachusetts, USA: MIT Press*, 1998.
- [77] R. V. Florian. Correct equations for the dynamics of the cart-pole system. Tech. rep., Center for Cognitive and Neural Studies (Coneural), Str. Saturn 24, 400504 Cluj-Napoca, Romania, 2007.

Appendix A

Implicit Differentiation in SyMos

A.1 Uniqueness and Existence of solutions in Symplectic-Momentum Neural Networks

Definition A.1.1 (Regular Lagrangians). A Lagrangian, $\mathcal{L} : \mathcal{T} \times \mathcal{Q} \rightarrow \mathbb{R}$ is said to be regular if the Hessian matrix $\frac{\partial^2 \mathcal{L}}{\partial \dot{q}_i \partial \dot{q}_j}$ is non-singular. In that case the Euler-Lagrange equations (3.25) have a unique solution.

Exact Discrete Lagrangian and Forcing [21]. Given a regular Lagrangian $\mathcal{L} : TQ \rightarrow \mathbb{R}$ and a Lagrangian control force $f_{\mathcal{L}} : TQ \times U \rightarrow T^*Q$, the *exact discrete Lagrangian* $L_d^E : Q \times Q \times \mathbb{R} \rightarrow \mathbb{R}$ and the exact discrete control forces $f_d^{E+}, f_d^{E-} : Q \times Q \times \mathcal{C}([0, h], U) \times \mathbb{R} \rightarrow T^*Q$ are given by

$$L_d^E = \int_0^h \mathcal{L}(q(t), \dot{q}(t)) dt, \quad (\text{A.1})$$

$$f_d^{E+}(q_0, q_1, u_0, h) = \int_0^h F_{\mathcal{L}}(q(t), \dot{q}(t), u(t)) \cdot \frac{\partial q(t)}{\partial q_1} dt, \quad (\text{A.2})$$

$$f_d^{E-}(q_0, q_1, u_0, h) = \int_0^h F_{\mathcal{L}}(q(t), \dot{q}(t), u(t)) \cdot \frac{\partial q(t)}{\partial q_0} dt, \quad (\text{A.3})$$

with $u_k \in \mathcal{C}([kh, (k+1)h], U)$ and $q : [0, h] \rightarrow Q$ is the solution of the forced Euler-Lagrange equations 3.7 with control function $u : [0, h] \rightarrow U$ for \mathcal{L} and F satisfying the boundary conditions $q(0) = q_0$ and $q(h) = q_1$. Based on the notion of exact discrete Lagrangian and forcing, the authors in [19] (for the unforced cased, (Lemma 1.6.2)) and [21] (Lemma 2.3) authors prove that for a regular Lagrangian and for sufficiently small h and close $q_0, q_1 \in Q$ then *exact discrete Lagrangians are automatically regular*.

Definition A.1.2 (Discrete Regular Lagrangians). A discrete Lagrangian, $\mathcal{L}_d : \mathcal{Q} \times \mathcal{Q} \rightarrow \mathbb{R}$ is said to be regular if the matrix $\frac{\partial \mathcal{L}_d}{\partial q_k \partial q_{k+1}}$ is non-singular [19]. In that case the unforced discrete Euler-Lagrange equations (3.43) have a unique solution.

Note that the Jacobian required for the root-finding process

$$J_g = \left[\frac{\partial \mathcal{L}_d(q_k, q_{k+1}; \theta)}{\partial q_{k+1} \partial q_k} + \frac{\partial f_d^-(q_k, q_{k+1}, u_k, u_{k+1})}{\partial q_{k+1}} \right], \quad (\text{A.4})$$

needs to be invertible. For the unforced cases or when the discrete forces do not depend on the configuration, existence and uniqueness is guaranteed, as long as the continuous Lagrangian describing the mechanical system is regular and for sufficiently small time-steps. For configuration dependent discrete forces the full matrix J_g should be regular.

A.2 Backward Pass for the Implicit RootFind Layer

The main core of implicit differentiation resides on the resulting properties of the Implicit Function Theorem A.2.1. In multivariable calculus, the implicit function theorem [74] is a principle that provides a sufficient condition for converting relations between variables to functions of a part of those variables. With this in mind, proving Theorem 5.2.1 is straightforward by making use of the chain rule.

Theorem A.2.1. *Implicit Function Theorem* [74, 75]. Let $g : \mathcal{R}^p \times \mathcal{R}^n \rightarrow \mathcal{R}^n$ and $x_0 \in \mathcal{R}^p$, $y_0 \in \mathcal{R}^n$ be such that

1. $g(x_0, y_0) = 0$, and
2. g is a C^1 function with non-singular Jacobian $\frac{\partial g(x_0, y_0)}{\partial y_0} \in \mathcal{R}^{n \times n}$.

Then there exist open sets $S_{x_0} \subset \mathcal{R}^p$ and $S_{y_0} \subset \mathcal{R}^n$ containing x_0 and y_0 , respectively, and a unique continuous function $f : S_{x_0} \rightarrow S_{y_0}$ such that

1. $y_0 = f(x_0)$.
2. $g(x, f(x)) = 0 \quad \forall x \in S_{x_0}$, and
3. f is differentiable on S_{x_0} .

Note that the theorem above gives a sufficient condition for defining the root of an implicit function as a function of the other variables, i.e., let q_{k+1}^* be the solution to the parameterized DEL equations (5.19), then we can write $q_{k+1}^*(x; \theta)$ as a function of the inputs $x = (q_{k-1}, q_k, u_{k-1}, u_k, u_{k+1})$ and parameters θ . The proof for the backward pass of the End-to-End Symplectic-Momentum neural networks follows next.

Theorem A.2.2. *Gradient of the RootFind solution.* Let $q_{k+1} \in \mathbb{R}^n$ be the solution to the physical constrained parameterized RootFind procedure based on the implicit DEL mapping $(q_{k-1}, q_k) \rightarrow (q_k, q_{k+1})$, defined by $g(q_{k+1}, x; \theta) \in \mathbb{R}^n$ (equation 5.19). The gradients of a scalar loss function $\mathbb{L}(q_{k+1}, x; \theta)$ with respect to the parameters θ are obtained by vector-Matrix products as follows:

$$\frac{\partial \mathbb{L}}{\partial \theta} = -\frac{\partial \mathbb{L}}{\partial q_{k+1}} \underbrace{\left[\frac{\partial \mathcal{L}(q_k, q_{k+1}; \theta)}{\partial q_{k+1} \partial q_k} + \frac{\partial f_d^-(q_k, q_{k+1}, u_k, u_{k+1})}{\partial q_{k+1}} \right]}_{J_g}^{-1} \frac{\partial g(q_{k+1}, x; \theta)}{\partial \theta} \quad (\text{A.5})$$

Proof. Under the conditions of theorem A.2.1, we can formulate the relation between the root q_{k+1}^* and the remaining variables as the graph of a function of the remaining variables, $q_{k+1}^* = f(x; \theta)$, that define

the implicit DEL equations. Let, the DEL implicit equations be defined by:

$$g(x, q_{k+1}, h; \theta) = \frac{\partial}{\partial q_k} [\mathcal{L}_d(q_{k-1}, q_k; \theta) + \mathcal{L}_d(q_k, q_{k+1}; \theta)] + f_d^-(q_k, q_{k+1}, u_k, u_{k+1}) + f_d^+(q_{k-1}, q_k, u_{k-1}, u_k). \quad (\text{A.6})$$

The theorem also tell us how to compute derivatives of $q_{k+1}(x; \theta)$ by implicit differentiation and based on that obtain the partial derivative of q_{k+1} w.r.t. θ .

$$\begin{aligned} g(x, q_{k+1}; \theta) &= 0 \\ \frac{\partial}{\partial \theta} [g(x, q_{k+1}; \theta)] &= 0 && \text{Differentiate both sides.} \\ \frac{\partial g(x, q_{k+1})}{\partial \theta} + \frac{\partial g(x, q_{k+1}; \theta)}{\partial q_{k+1}} \cdot \frac{\partial q_{k+1}}{\partial \theta} &= 0 && \text{Using the Chain Rule.} \\ \frac{\partial q_{k+1}}{\partial \theta} &= - \left[\frac{\partial g(x, q_{k+1}; \theta)}{\partial q_{k+1}} \right]^{-1} \frac{\partial g(x, q_{k+1})}{\partial \theta} && \text{Rearranging the equations.} \end{aligned}$$

Since the goal of back-propagation is to compute the gradient with respect to some scalar loss, L , and based on equation (A.6) our backward pass consists on:

$$\frac{\partial L}{\partial \theta} = - \frac{\partial L}{\partial q_{k+1}} \underbrace{\left[\frac{\partial \mathcal{L}_d(q_k, q_{k+1}; \theta)}{\partial q_{k+1} \partial q_k} + \frac{\partial f_d^-(q_k, q_{k+1}, u_k, u_{k+1})}{\partial q_{k+1}} \right]}_{J_g}^{-1} \frac{\partial g(x, q_{k+1}; \theta)}{\partial \theta},$$

under the condition that J_g is invertible (see annex A.1) and q_{k+1} is, indeed, a root of g . □

Appendix B

Models

B.1 Physical Pendulum

The physical pendulum is a one degree of freedom system consisting of a rigid body that rotates about a fixed point. The coordinate system (potential energy equals zero) and force diagram for the simple pendulum is shown in figure B.1.

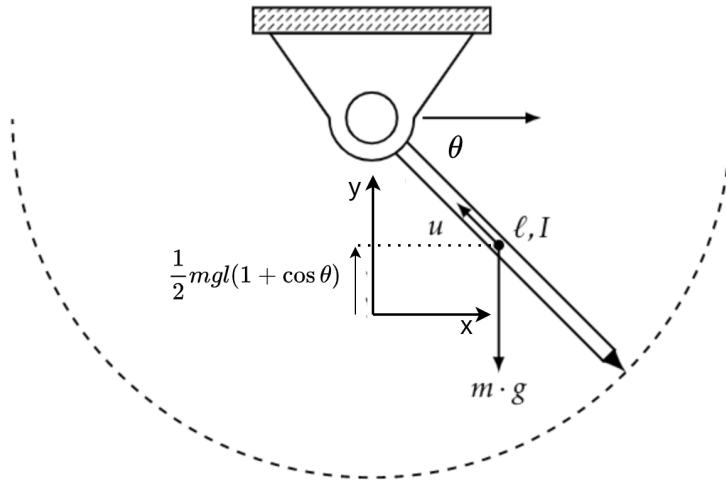


Figure B.1: Free-body force diagram on rod with a gravitational and an external force acting on the center of mass of the pendulum.

The energy is given by

$$T = \frac{1}{2}I_G\dot{\theta}^2 \quad (\text{B.1})$$

$$V = \frac{1}{2}mg\ell(1 + \cos\theta), \quad (\text{B.2})$$

where $I_g = \frac{1}{3}ml^2$ is the moment of inertia of a rod about the end point (origin). The equations of motion

are given by [69]:

$$\ddot{\theta} = \frac{3}{2l}mg \sin \theta + \frac{3u}{ml^2}. \quad (\text{B.3})$$

| Physical Pendulum Parameters | | |
|------------------------------|-------|---------|
| Variable | Value | Unit |
| g | 9.8 | m/s^2 |
| l | 1 | m |
| m | 1 | kg |

Table B.1: Set of parameters for the physical pendulum system used throughout this work. The values used are the standard ones in [69]

B.2 Acrobot

The acrobot is a two-link, underactuated robot analogous to a gymnast swinging on a high bar (B.2). The first joint (corresponding to the gymnast's hands on the bar) cannot exert torque, but the second joint (corresponding to the gymnast bending at the waist) can. The system has four continuous state variables: two joint positions and two joint velocities.

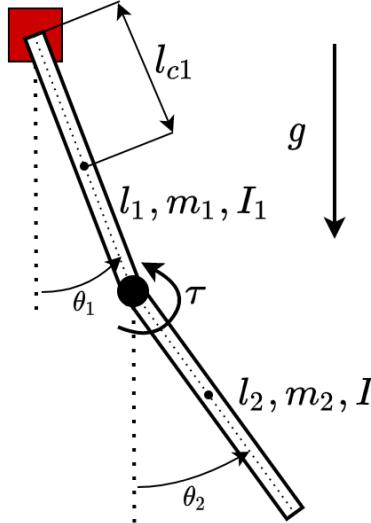


Figure B.2: A two-link underactuated robot - Acrobot. The angle θ is the counter-clockwise angle of both the pendulum system (zero is hanging straight down). The generalized coordinates are $q = [\theta_1, \theta_2]^T$. I_1 , I_2 and l_1 , l_2 are the moments of inertia and lengths of links respectively. l_{c1} and l_{c2} are the moments of inertia of links and g gravity. The respective values used can be found in table B.2. The acrobot is a chaotic system when unforced. The following derivation is coherent with the equations of motion of [76] and the OpenAI simulator [69]. We designate c_1 and c_2 as the cosines of θ_1 and θ_2 . C_{1+2} is the cosine of $\theta_1 + \theta_2$.

The Acrobot energy is given by

$$T = \frac{1}{2}\dot{q}^T H(q)\dot{q} \quad (\text{B.4})$$

$$V = -m_1 g l_{c1} c_1 - m_2 g (l_1 c_1 + l_{c2} c_{1+2}), \quad (\text{B.5})$$

with

$$H(q) = \begin{bmatrix} m_1 l_{c1}^2 + m_2(l_1^2 + l_{c2}^2 + 2l_1 l_{c2} \cos \theta_2) + I_1 + I_2 & m_2(l_{c2}^2 + l_1 l_{c2} \cos \theta_2) + I_2 \\ m_2(l_{c2}^2 + l_1 l_{c2} \cos \theta_2) + I_2 & m_2 l_{c2}^2 + I_2 \end{bmatrix},$$

The equations of motion for the acrobot are [76]:

$$\ddot{\theta}_1 = -d_1^{-1}(d_2 \ddot{\theta}_2 + \phi_1), \quad (\text{B.6})$$

$$\ddot{\theta}_2 = \left(m_2 l_{c2}^2 + I_2 - \frac{d_2^2}{d_1} \right)^{-1} \left(\tau + \frac{d_2}{d_1} \phi_1 - m_2 l_1 l_{c2} \dot{\theta}_1^2 \sin \theta_2 - \phi_2 \right), \quad (\text{B.7})$$

$$d_1 = m_1 l_{c1}^2 + m_2(l_1^2 + l_{c2}^2 + 2l_1 l_{c2} \cos \theta_2) + I_1 + I_2, \quad (\text{B.8})$$

$$d_2 = m_2(l_{c2}^2 + l_1 l_{c2} \cos \theta_2) + I_2, \quad (\text{B.9})$$

$$\phi_1 = -m_2 l_1 l_{c2} \dot{\theta}_2^2 \sin \theta_2 - 2m_2 l_1 l_{c2} \dot{\theta}_2 \dot{\theta}_1 \sin \theta_2 \quad (\text{B.10})$$

$$+ (m_1 l_{c1} + m_2 l_1) g \cos(\theta_1 - \pi/2) + \phi_2, \quad (\text{B.11})$$

$$\phi_2 = m_2 l_{c2} g \cos(\theta_1 + \theta_2 - \pi/2). \quad (\text{B.12})$$

| Acrobot Parameters | | |
|--------------------|-------|---------|
| Variable | Value | Unit |
| g | 9.81 | m/s^2 |
| l_1 | 1 | m |
| l_2 | 1 | m |
| l_{c1} | 0.5 | m |
| l_{c2} | 0.5 | m |
| m_1 | 1 | kg |
| m_2 | 1 | kg |
| I_1 | 1 | kgm^2 |
| I_2 | 1 | kgm^2 |

Table B.2: Set of parameters for the acrobot system used throughout this work. The values used are the same as [69].

B.3 Cartpole

The other model system that we will investigate in this work is the cartpole system. The cartpole consists of a pendulum on a cart that can move on the horizontal axis.

The equations of motion are given by [77]

$$\ddot{\theta} = \frac{g \sin \theta + \cos \theta \left(\frac{-F - m_p L \dot{\theta}^2 \sin \theta}{m_c + m_p} \right)}{L \left(\frac{4}{3} - \frac{m_p \cos^2 \theta}{m_c + m_p} \right)}, \quad (\text{B.13})$$

$$\ddot{x} = \frac{F + m_p L (\theta^2 \sin \theta - \dot{\theta} \cos \theta)}{m_c + m_p}, \quad (\text{B.14})$$

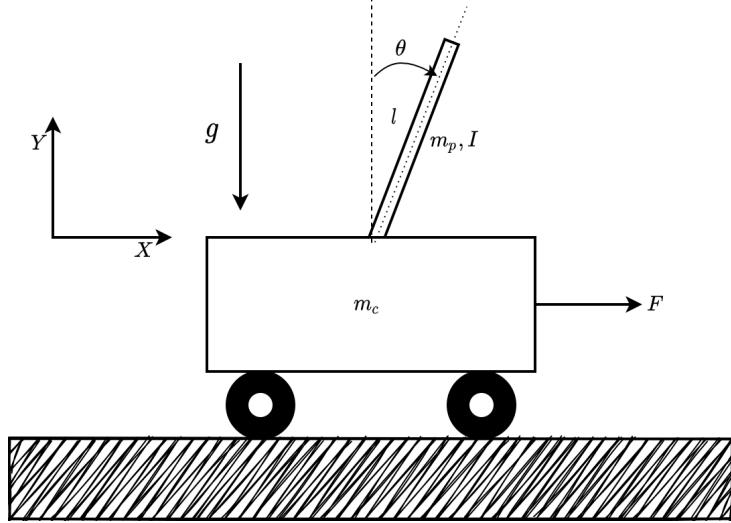


Figure B.3: Inverted Pendulum on a cart - Cartpole. The angle θ is the counter-clockwise angle of the pole (zero is hanging straight up). The generalized coordinates are $q = [x, \theta]^T$. I is the moment of inertia of the pole with $L = 0.5l$. The respective values used can be found in table B.3. The cartpole dynamics derived here are coherent with the ones used in OpenAI Gym toolkit for reinforcement learning [69] and with [77].

or in the standard manipulator form, using $F = u$:

$$H(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) = Bu, \quad (\text{B.15})$$

where

$$H(q) = \begin{bmatrix} m_c + m_p & m_p L \cos \theta \\ m_p L \cos \theta & \frac{4}{3} m_p L^2 \end{bmatrix}, \quad C(q, \dot{q}) = \begin{bmatrix} 0 & -m_p L \dot{\theta} \sin \theta \\ 0 & 0 \end{bmatrix},$$

$$G(q) = \begin{bmatrix} 0 \\ -m_p g L \sin \theta \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

| Cartpole Parameters | | |
|---------------------|-------|---------|
| Variable | Value | Unit |
| g | 9.8 | m/s^2 |
| m_p | 0.1 | kg |
| m_c | 1 | kg |
| L | 0.5 | m |

Table B.3: Set of parameters for the cartpole system used throughout this work. Coherent with [69].