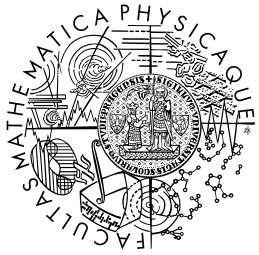


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jan Benda

Machine learning for recognition of simple physical systems

Mathematical Institute of Charles University

Supervisor of the bachelor thesis: doc. RNDr. Michal Pavelka, Ph.D.
Study programme: Mathematical Modelling
Study branch: MMOP

Prague 2024

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor, doc. Michal Pavelka, for his patient guidance and advice.

Title: Machine learning for recognition of simple physical systems

Author: Jan Benda

Institute: Mathematical Institute of Charles University

Supervisor: doc. RNDr. Michal Pavelka, Ph.D., Mathematical Institute of Charles University

Abstract: The rise of machine learning, particularly through the use of neural networks, has begun to change how we solve problems, including understanding simple physical systems. This thesis focuses on the Direct Poisson Neural Network (DPNN), a network that uses the structure of Hamilton's equations of motion to learn from data. This method allows us to extract the Hamiltonian and Poisson bivector from the data, helping to identify the type of physical systems. We explore how DPNN works with noisy data and when data is limited, checking its ability to make predictions in challenging conditions. Moreover, we have implemented Energy Ehrenfest regularisation to the model, which helps it recognise and simulate dissipative systems better.

Keywords: Machine Learning Hamiltonian Systems System Recognition Neural Networks

Contents

List of Abbreviations

DPNN Direct Poisson Neural Network

GD Gradient Descent

SGD Stochastic Gradient Descent

MSE Means Square Error

MLE Maximum Likelihood Estimator

WJ DPNN without Jacobi's identity regularisation

SJ DPNN with soft Jacobi's identity regularisation

IJ DPNN with implict Jacobi's identity

FE Forward Euler Method

CN Cranck-Nicolson scheme

IMR Implicit Midpoint Rule

RK4 Runge-Kutta of fourth order

Introduction

With the rise of large language models such as GPT4, deep learning has been growing in popularity in recent years. While achieving previously unimaginable results in high-dimensional problems like image and text generation, translation, or speech synthesis, it is worth exploring the potential of such models on problems, which we already have solutions for, in our case recognition and simulation of Hamiltonian systems from measured or calculated data.

One broad category of neural networks, that achieves this, is dubbed Physics-Informed Neural Networks, or PINNs for short. The core idea behind PINNs is to use plain Neural Networks, impose some physics-based constraints (e.g. conservation laws) on them, and train them on measured or simulated data. Some PINNs go even further and reflect the problem's underlying structure in their architecture. If the network is designed to reflect the structure of Hamilton's equations of motion, we can make it learn the Hamiltonian and recognise the underlying physical system based on it.

That is essentially what Direct Poisson Neural Network (DPNN) developed by Martin Šípka [2023] does. From the provided data it learns the Hamiltonian and Poisson bivector, from which we can identify the type of the system, its degeneracy, and more.

In this thesis, we explained the inner workings of DPNN and reproduced the results made by its authors. Then we explore its behaviour in settings, where there is either some amount of noise present in the dataset or when there is not enough data and the network needs to extrapolate. Finally, we have implemented Energy Ehrenfest regularisation described in Pavelka et al. [2019], which made it possible to recognise the presence of dissipation in the system.

1. Introduction to Deep Learning

This chapter covers basic parts and working principles of neural networks, which are key in deep learning. We will look into how these networks are built, trained and regulated to extract and process the information from provided data.

1.1 Neural networks basics

The core idea behind machine learning is fairly simple. Suppose have some data, let's say an image, and we would like to perform an operation on this data, e.g. count the number of cats or identify the person in the picture. This abstract mapping is called the ground truth. Since we can never have all the data in the world, all we can hope for is to get a solid approximation of this ground truth function. This is where machine learning models step in.

In this thesis we will be focusing only on one discipline of machine learning called deep learning or neural networks. The building blocks of a neural network are units called neurons or nodes. In their simplest form, each neuron takes in N input values and produces one output value using the following formula

$$x^{\text{out}} = \sigma\left(\sum_{i=1}^{N} w_i x_i^{\text{in}} + b\right) = \sigma\left(\mathbf{w}^{\mathsf{T}}\mathbf{x}^{\text{in}} + b\right), \tag{1.1}$$

where $\mathbf{w}$ and b are some unspecified parameters called weights and bias and σ a non-polynomial function called activation function. Concrete values $\mathbf{w}$ and b are sought after in a process called model training and are specific for each neuron, while activation functions are given by the network architecture and do not change during the training process.

In the simplest design of neural networks, neurons are organised into layers of fixed width denoted D - each forwarding its output as an input to the next layer. The first and the last layers are called the input and output layers respectively and the rest is called hidden layers. The number of hidden layers is usually referred to as the depth of the neural network and is $L + 1$[1]. Mathematically, we will refer to the neural network as a function Φ which can be written as a composition

$$\Phi = A_L \circ \boldsymbol{\sigma} \circ A_{L-1} \circ \cdots \circ A_2 \circ \boldsymbol{\sigma} \circ A_1\,, \tag{1.2}$$

where A_i as an affine transformation on $\mathbb{R}^D$ [2]. Algebraically it was defined elementwise in formula (1.1) as an argument of σ. Bold $\boldsymbol{\sigma}$ is just an elementwise generalisation of σ

$$\boldsymbol{\sigma}(\mathbf{x})_i = \sigma(\mathbf{x}_i). \tag{1.3}$$

This is a very powerful scheme according to the universal approximation theorem discussed in e.g. Hornik et al. [1989], Cybenko [1989], **?**. It, loosely speaking, states, that any function $f : \mathbb{R}^m \to \mathbb{R}^n$ can be approximated arbitrarily well on $K \subset \mathbb{R}^m$ by a sufficiently wide neural network of depth 2. In other words, if we happen to

[1] L is the number of affine transformations between layers.

[2] Input and output layers have typically lower dimensionality, which needs to be accounted for by making $A_1 : \mathbb{R}^{D_{\text{in}}} \to \mathbb{R}^D$ and $A_{L+1} : \mathbb{R}^D \to \mathbb{R}^{D_{\text{out}}}$.

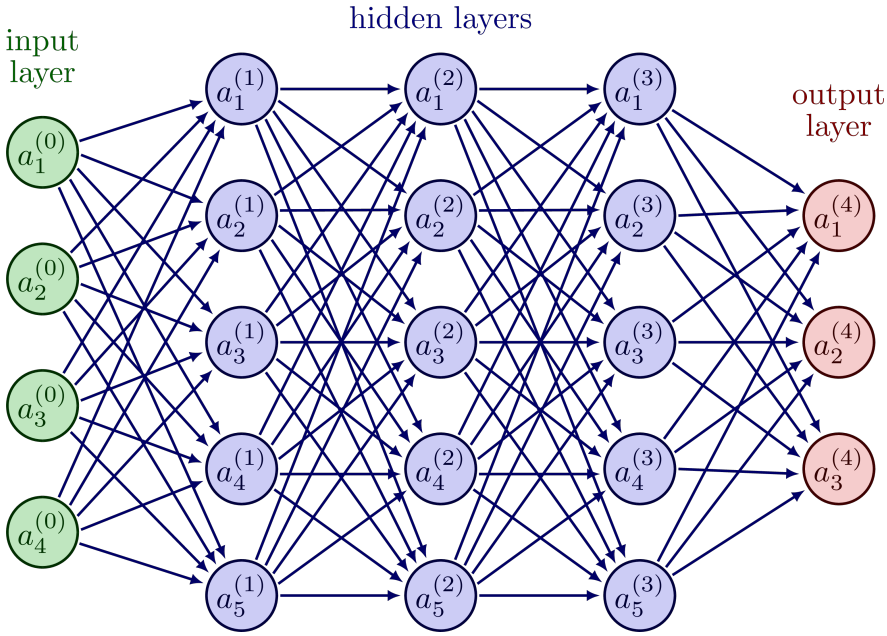


Figure 1.1: A neural network with a maximal width of 5 and a depth of 4. Each arrow represents one weight parameter of the neuron it is pointing at.

find the right values for weights and biases of each neuron, we can approximate the ground truth function we were looking for, thus solving the problem at hand with deep learning.

1.2 Training the model

To train the model we first need to get a dataset $\mathcal{D}$ to train on. That is a set of ordered pairs of features $\mathbf{x}_i$ and targets $\mathbf{t_i}$. Features will serve as an input to the neural network whose output will be compared to the targets.

1.2.1 Training and Test set

One of the easiest ways to measure the performance of the model is to calculate the mean square error between its predictions and the target values.

$$\mathcal{L}_{\text{MSE}}(\boldsymbol{\theta}) = \frac{1}{2|\mathcal{D}|} \sum_{(\mathbf{x},\mathbf{t})\in\mathcal{D}} \|\Phi(\boldsymbol{\theta}, \mathbf{x}) - \mathbf{t}\|^2 , \tag{1.4}$$

where $\boldsymbol{\theta}$ represents the parameters of the model described in the previous section. The measure of error is commonly called a loss function and is denoted by $\mathcal{L}$ and plays a crucial role in the process of training as we will see further in this chapter.

But this is not entirely correct. The model may learn to recognise the individual data points from the data set it has been trained on, which typically results in poor performance on data, which it yet has not seen. In this situation, $\mathcal{L}_{\text{MSE}}$ would be useless as a performance metric. For this reason, we have to split our dataset $\mathcal{D}$ into a training set $\mathcal{D}_{\text{train}}$ to be used during training and test set $\mathcal{D}_{\text{test}}$ to measure the generalisation error. This leads to a trade-off between performance and our ability to quantify it. The

relative size of the test set $\mathcal{D}_{\text{test}}$ to the whole dataset $\mathcal{D}$ is usually taken from the range of $20\% - 40\%$.

1.2.2 Gradient Descent

The goal of the training process is to tweak parameters $\boldsymbol{\theta}$ to minimise the loss function. The simplest algorithm which achieves this, is called gradient descent (GD for short) and works in the following way.

1. We initialise $\boldsymbol{\theta}_0$ with uniformly randomly generated numbers
2. We iteratively calculate better approximations of the minimising parameters using the formula
$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \lambda \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{MSE}}(\boldsymbol{\theta}_i)\,, \tag{1.5}$$
for a given number of iterations.

The variable λ is called learning rate and is used to control the speed of learning and is typically chosen to be in range $\left[10^{-2}, 10^{-4}\right]$.

$\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{MSE}}(\boldsymbol{\theta}_i)$ is computed in a following way:

$$[\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{MSE}}(\boldsymbol{\theta}_i)]_l \approx \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(\mathbf{x},\mathbf{t})\in\mathcal{D}_{\text{train}}} (\Phi(\boldsymbol{\theta}_i, \mathbf{x}) - \mathbf{t})^{\mathsf{T}} \frac{\partial \Phi}{\partial \boldsymbol{\theta}_l}(\boldsymbol{\theta}_i, \mathbf{x})\,. \tag{1.6}$$

Training of large neural networks with millions of weights is commonly done over datasets with hundreds of thousands of samples. This yields a calculation of approximately 10^{11} gradients to produce one iteration of GD. Since datasets this big are bound to contain a lot of redundancy, we do not need to use the whole $\mathcal{D}_{\text{train}}$ to have a good approximation of the gradient.

A version of GD, called mini-batch GD or stochastic GD (SGD for short)[3], splits $\mathcal{D}$ into disjoint sets of equal size called mini-batches $\mathcal{B}_k$. Each step in 1.5 will in this case be taken using the approximation of the gradient

$$[\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{MSE}}(\boldsymbol{\theta}_i)]_l \approx \frac{1}{|\mathcal{B}_k|} \sum_{(\mathbf{x},\mathbf{t})\in\mathcal{B}_k} (\Phi(\boldsymbol{\theta}_i, \mathbf{x}) - \mathbf{t})^{\mathsf{T}} \frac{\partial \Phi}{\partial \boldsymbol{\theta}_l}(\boldsymbol{\theta}_i, \mathbf{x})\,. \tag{1.7}$$

After we use up the whole dataset, we create new mini-batches from it and repeat the process. One iteration of this process is called an epoch.

In most cases, SGD converges faster than GD since it makes multiple steps per epoch which corresponds to only one step of GD. Also by considering only a part of $\mathcal{D}_{\text{train}}$, the optimizer is less likely to get stuck in a local minimum, further enhancing its robustness.

Other heuristics can be leveraged to further improve the capabilities of the optimiser. For example, ADAM[4], which is one of the most used optimiser, employs besides mini-batching also a learning momentum or automatic adjustment of the learning rate.

Now that we know how to train a neural network, we will look at how to moderate the training process using various regularisations.

[3]Technically SGD is a different algorithm - mini-batch GD with batch size 1, but in practice mini-batch is usually referred to as SGD

[4]Proposed in Kingma and Ba [2014].

1.3 Model capacity and regularisation

Choosing the right model size is a complicated task. A model too small for the task at hand will produce large generalisation errors no matter for how long has it been trained.

On the other hand, oversized models will, loosely speaking, learn to recognise data from train set based on ever-present noise. This again leads to poor performance in the testing phase.

The predictive capabilities of a model are referred to as the capacity of a model. In the first case, the model is said to have insufficient capacity which leads to under-fitting. In the second case, the capacity is said to be excessive and the model has over-fitted.

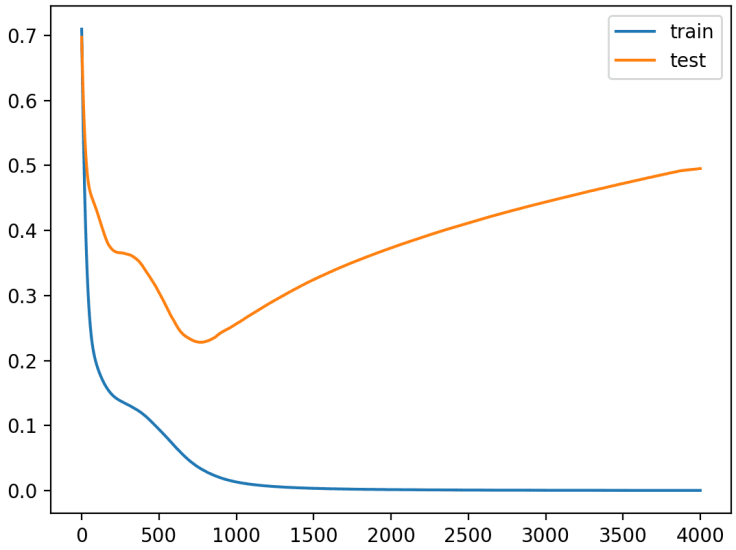


Figure 1.2: The dependence of train and test loss on the number of epochs. Over-parametrized model is learning at first, but then it starts to over-fit as the test loss starts growing. Source: PhD [2019]

To control the capacity of the model there is a handful of regularisation techniques at our disposal.

Early-Stopping

Early stopping is the easiest way to regularise the model. Test error is monitored during training and if it does not decrease in some number of consecutive epochs usually called patience, training is interrupted.

This prevents model over-fitting but may produce sub-optimal results since determining the right time to stop is sort of an arbitrary task usually with some heuristics involved.

Dropout

When dropout is in use, a fraction p of neurons in the layer is randomly chosen and has their output value artificially set to zero during the training phase. This essentially

turns them off and forces the neural not to rely on values of only a few neurons. Then, during the inference, the output of each layer needs to be scaled by a factor of p since all neurons are active now.

Networks trained in this way tend to be better at generalisation but at a cost of greater training time.

L2 and L1 regularisation

Over-trained networks tend to be sensitive to small changes in the input, which is a consequence of large values of weight parameters. To control their size we modify the loss function by adding the L2 or L1 norm of the parameters with the aim of "punishing" large parameter values.

$$\mathcal{L}_{\mathrm{L2}} = \frac{1}{2}\mathcal{L} + \lambda \sum_i \theta_i^2 \,, \qquad \mathcal{L}_{\mathrm{L1}} = \mathcal{L} + \lambda \sum_i |\theta_i| \,, \tag{1.8}$$

where θ_i are the weights and biases and λ serves to control the strength of the regularisation.

While L2 and L1 look similar, they differ in the behaviour they produce. Additional gradient term produced by L2 regularisation for a given parameter is proportional to its size. This makes it useful against outliers hidden in parameters.

L1, on the other hand, produces a gradient term of constant size same for every parameter. Regularisation therefore does not weaken as values of parameters decrease and can push them to zero. This makes L1 useful in situations when compressing the model by cutting the connections is desirable.

Also, both L2 and L1 are not considered safe regularisation, since they both introduce a bias towards 0 in the network output, especially for large values of λ.

In a way similar to L2 regularisation, it is possible to enforce almost any condition on Φ. This can be helpful, especially in situations when we know that the function to be approximated by the network possesses some property. We can let the network learn this property from the provided dataset or "help" it learn the property by incorporating it into the loss function.

1.4 Coping with noise

Most of this section is based on the lecture given by RNDr. Milan Straka, PhD., namely Straka [2023].

Every dataset contains some noise level, in the form of picture misclassification, measurement errors, etc. If we know something about its nature, it is possible to enhance the robustness of the network at hand against this type of noise, by choosing the right loss function.

Let us suppose that our dataset is affected by additive Gaussian noise. i.e

$$\tilde{t}(x) \sim \mathcal{N}\left(t(x), \sigma^2\right). \tag{1.9}$$

where $t(x)$ represents the ground truth function. We will treat the neural network as a maximum likelihood estimator (MLE) of the mean of the Gaussian. We assume σ^2 to

be constant and all elements to be i.i.d. MLE then satisfies

$$\theta_{\text{MLE}} = \arg\max_{\theta} \prod_{(x_i,t_i)\in\mathcal{D}} \frac{1}{\sqrt{2\pi\sigma^2}} \mathrm{e}^{\frac{-(t_i-\Phi(x_i,\theta))^2}{2\sigma^2}}, \tag{1.10}$$

$$= \arg\min_{\theta} -\ln\left(\prod_{(x_i,t_i)\in\mathcal{D}} \frac{1}{\sqrt{2\pi\sigma^2}} \mathrm{e}^{\frac{-(t_i-\Phi(x_i,\theta))^2}{2\sigma^2}} \right), \tag{1.11}$$

$$= \arg\min_{\theta} -\sum_{(x_i,t_i)\in\mathcal{D}} \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}} \mathrm{e}^{\frac{-(t_i-\Phi(x_i,\theta))^2}{2\sigma^2}} \right), \tag{1.12}$$

$$= \arg\min_{\theta} -\sum_{(x_i,t_i)\in\mathcal{D}} \left(\ln\left(\frac{1}{\sqrt{2\pi\sigma^2}} \right) - \frac{(t_i-\Phi(x_i,\theta))^2}{2\sigma^2} \right), \tag{1.13}$$

$$= \arg\min_{\theta} \frac{1}{2|\mathcal{D}|} \sum_{(x_i,t_i)\in\mathcal{D}} (t_i - \Phi(x_i,\theta))^2 . \tag{1.14}$$

MLE is a consistent estimator. This in our setting means that if we choose a bigger subset of $\mathcal{D}$ to calculate θ'_{MLE} using 1.14, then the network Φ parametrized with θ'_{MLE} will make more accurate predictions on $\mathcal{D}$.

Furthermore, it has also been shown by Rao and Cramér that out of all consistent estimators, MLE produces the smallest means square error. This justifies the choice of mean square error as a loss function in situations where $\mathcal{D}$ contains Gaussian noise.

It is also regarded as a good practice to choose the appropriate activation function for the last layer, such that the formula for calculating the gradients always takes the form described in 1.6. This for Gaussian noise happens to be the identity transformation.

2. Hamiltonian Systems

If we come across a physical system whose behaviour we would like to predict, our tool of choice would most likely be Newton's equations of motion for their simplicity and intuitiveness. They may work well with simple systems like a harmonic oscillator or an object in free fall, but for more complex systems, like a double pendulum, the mathematics becomes rather cumbersome.

To cope with this problem we may employ different approaches like Lagrange's or Hamilton's equations of motion, which use more advanced mathematics, but make the calculations more straightforward and most importantly offer richer theoretical insight into the problem at hand.

2.1 Hamilton's equations

Let us start with the principle of the least action.

$$\delta S = \delta \int_{t_0}^{t_1} L(\dot{\mathbf{q}}(t), \mathbf{q}(t), t)\mathrm{d}t = 0\,. \tag{2.1}$$

Hamiltonian H is defined as the Legendre transform of the Lagrangian

$$H = p_i \dot{q}^i - L\,, \tag{2.2}$$

where $p_i = \frac{\partial L}{\partial \dot{q}_i}$ By substituting one formula into the other we get

$$\begin{aligned}
0 &= \delta \int_{t_0}^{t_1} p_i \dot{q}^i - H\mathrm{d}t\,, && (2.3)\\
&= \int_{t_0}^{t_1} \delta(p_i \dot{q}^i) - \delta H\mathrm{d}t\,, && (2.4)\\
&= \int_{t_0}^{t_1} \dot{q}^i \delta p_i + p_i \delta \dot{q}^i - \frac{\partial H}{\partial p_i}\delta p_i - \frac{\partial H}{\partial q^i}\delta q^i \mathrm{d}t\,, && (2.5)\\
&= \int_{t_0}^{t_1} \dot{q}^i \delta p_i - \dot{p}_i \delta q^i - \frac{\partial H}{\partial p_i}\delta p_i - \frac{\partial H}{\partial q^i}\delta q^i \mathrm{d}t\,, && (2.6)\\
&= \int_{t_0}^{t_1} \left(\dot{q}^i - \frac{\partial H}{\partial p_i}\right)\delta p_i - \left(\dot{p}_i + \frac{\partial H}{\partial q^i}\right)\delta q^i \mathrm{d}t\,, && (2.7)
\end{aligned}$$

where we used the integration by parts and the fact that $\delta q_i(t_1) = \delta q_i(t_2) = 0$ in the third step, yields Hamilton's equations

$$\dot{q}_i = \frac{\partial H}{\partial p_i}\,, \qquad \dot{p}_i = -\frac{\partial H}{\partial q_i}\,, \tag{2.8}$$

which can be written in a more compact form using $\mathbf{z} = (q_1, ..., q_N, p_1, ..., p_N) \in \mathbb{R}^{2N}$.

$$\dot{\mathbf{z}} = L_C \boldsymbol{\nabla} H \tag{2.9}$$

where $\boldsymbol{\nabla} H = \left(\frac{\partial H}{\partial q_1}, ..., \frac{\partial f}{\partial q_N}, \frac{\partial f}{\partial p_1}, ..., \frac{\partial f}{\partial p_N}\right)$ and $L_C \in \mathbb{R}^{2N\times 2N}$ is a matrix representing canonical Poisson bivector

$$L_C = \begin{pmatrix} 0 & I_{N\times N} \\ -I_{N\times N} & 0 \end{pmatrix}. \tag{2.10}$$

2.2 Poisson Brackets

Suppose that we want to calculate the time derivative of function f.

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \sum_i \left(\frac{\partial f}{\partial q_i}\dot{q}_i + \frac{\partial f}{\partial p_i}\dot{p}_i \right) + \frac{\partial f}{\partial t}\,, \tag{2.11}$$

$$= \sum_i \left(\frac{\partial f}{\partial q_i}\frac{\partial H}{\partial p_i} - \frac{\partial f}{\partial p_i}\frac{\partial H}{\partial q_i} \right) + \frac{\partial f}{\partial t}\,. \tag{2.12}$$

By denoting

$$\{f, g\}_{\mathrm{C}} = \sum_i \left(\frac{\partial f}{\partial q_i}\frac{\partial g}{\partial p_i} - \frac{\partial f}{\partial p_i}\frac{\partial g}{\partial q_i} \right), \tag{2.13}$$

supposing $\frac{\partial f}{\partial t} = 0$, we can shortly write

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \{f, H\}_{\mathrm{C}}\,. \tag{2.14}$$

The operation $\{\circ, \circ\}_{\mathrm{C}}$ is called a canonical Poisson bracket and as we have just shown it naturally arises when we try to describe the rate of change of arbitrary variables.

2.2.1 Properties of Poisson brackets

There are four key properties of Poisson brackets. Let $\alpha, \beta \in \mathbb{R}$ and f, g, h differentiable functions on the phase space. The Poisson bracket needs to satisfy

1. Skew-symmetry
$$\{f, g\}_{\mathrm{C}} = -\{g, f\}_{\mathrm{C}} \tag{2.15}$$

2. Bilinearity
$$\{f + g, h + i\}_{\mathrm{C}} = \{f, h\}_{\mathrm{C}} + \{f, i\}_{\mathrm{C}} + \{g, h\}_{\mathrm{C}} + \{g, i\}_{\mathrm{C}}\,,\ \{\alpha f, \beta g\}_{\mathrm{C}} = \alpha\beta\{f, g\}_{\mathrm{C}} \tag{2.16}$$

3. Leibnitz's rule
$$\{fg, h\}_{\mathrm{C}} = f\{g, h\}_{\mathrm{C}} + \{f, h\}_{\mathrm{C}}g \tag{2.17}$$

4. Jacobi's identity
$$\{f, \{g, h\}_{\mathrm{C}}\}_{\mathrm{C}} + \{h, \{f, g\}_{\mathrm{C}}\}_{\mathrm{C}} + \{g, \{h, f\}_{\mathrm{C}}\}_{\mathrm{C}} = 0 \tag{2.18}$$

These properties lead to a few interesting implications. For example, combining 2.14 and 2.15 yields

$$\frac{\mathrm{d}H}{\mathrm{d}t} = \{H, H\}_{\mathrm{C}} = -\{H, H\}_{\mathrm{C}} = 0\,.$$

In other words, the skew-symmetry of the Poisson bracket is sufficient to produce the dynamics in which the total energy is conserved.

Another consequence of these properties allows us to construct new integrals of motion from ones, we already know. Let f, g be the integrals of motion. Then

1. $\alpha f + \beta g$,

2. fg,

3. $\{f, g\}_{\mathrm{C}}$

are also integrals of motion. For (1) and (2) can this be directly proven from 2.16 and 2.17 respectively. To prove (3) we will use 2.18 in the following way.

$$\begin{aligned}
\frac{\mathrm{d}\{f, g\}_{\mathrm{C}}}{\mathrm{d}t} &= \{\{f, g\}_{\mathrm{C}}, H\}_{\mathrm{C}}\,, && (2.19)\\
&= -\{H, \{f, g\}_{\mathrm{C}}\}_{\mathrm{C}}\,, && (2.20)\\
&= \{g, \{H, f\}_{\mathrm{C}}\}_{\mathrm{C}} + \{f, \{g, H\}_{\mathrm{C}}\}_{\mathrm{C}}\,, && (2.21)\\
&= \{g, 0\}_{\mathrm{C}} + \{f, 0\}_{\mathrm{C}}\,, && (2.22)\\
&= 0\,. && (2.23)
\end{aligned}$$

2.3 Non-canonical Poisson brackets

So far we have been considering only canonical Poisson brackets. As the naming suggests, there exist other, non-canonical bracket operations capable of producing various dynamical systems. Let us, to properly motivate the general Poisson bracket, take a closer look at the canonical Poisson bracket. The Poisson bracket can be reformulated using the canonical Poisson bivector L_C introduced in the last section.

$$\{f, g\}_{\mathrm{C}} = \boldsymbol{\nabla} f^{\mathsf{T}} L_C \boldsymbol{\nabla} g\,. \tag{2.24}$$

It is worthy of mention that this is a direct consequence of a famous relation between canonical coordinates

$$\left\{q_i, q_j\right\}_{\mathrm{C}} = 0\,, \quad \left\{q_i, p_j\right\}_{\mathrm{C}} = \delta_{ij}\,, \quad \left\{p_i, p_j\right\}_{\mathrm{C}} = 0\,. \tag{2.25}$$

Now, to go from a canonical Poisson bracket to a generalised one we replace L_C with arbitrary skew-symmetric $L(\mathbf{z}) \in \mathbb{R}^{N\times N}$

$$\{f, g\} = \boldsymbol{\nabla} f^{\mathsf{T}} L(\mathbf{z}) \boldsymbol{\nabla} g\,, \tag{2.26}$$

and Hamilton's equations become

$$\dot{\mathbf{z}} = \{\mathbf{z}, H\} = L(\mathbf{z}) \boldsymbol{\nabla} H\,. \tag{2.27}$$

Note that at this point we have left the world of generalised coordinates and cotangent bundles completely. In this new setting not only can L depend on $\mathbf{x}$, we can even have $\mathbf{x} \in \mathbb{R}^N$ for odd N, which was with canonical Hamiltonian systems impossible. The only condition we are enforcing is that the Poisson bracket generated by $L(x)$ satisfies properties outlined in 2.15,2.16,2.17 and 2.18.

It is evident that 2.16 and 2.17 are satisfied - they are consequences of the linearity of derivatives and product rule. 2.15 also holds if $L(x)$ is skew-symmetric

$$\{f, g\} = \boldsymbol{\nabla} f^{\mathsf{T}} L(x) \boldsymbol{\nabla} g = \left(\boldsymbol{\nabla} f^{\mathsf{T}} L(x) \boldsymbol{\nabla} g\right)^{\mathsf{T}} = \boldsymbol{\nabla} g^{\mathsf{T}} L(x)^{\mathsf{T}} \boldsymbol{\nabla} f = -\boldsymbol{\nabla} g^{\mathsf{T}} L(x) \boldsymbol{\nabla} f = -\{g, f\}\,.$$

Only enforcing Jacobi's identity (2.18) is not trivial.

2.3.1 Jacobiator

To arrive at a condition on $L(x)$ equivalent to 2.18 we will plug 2.26 into 2.18. To make derivation more clear let us focus just on the first bracket.

$$\{f,\{g,h\}\} = \mathrm{d}f_k\, L^{kl}\mathrm{d}^2 g_{li} L^{ij}\,\mathrm{d}h_j + \mathrm{d}f_k\, L^{kl}\mathrm{d}g_i \frac{\partial L^{ij}}{\partial z_l}\,\mathrm{d}h_j + \mathrm{d}f_k\, L^{kl}\mathrm{d}g_i L^{ij}\mathrm{d}^2 h_{jl}\,, \tag{2.28}$$

$$= \mathrm{d}f_k\, L^{kl}\mathrm{d}^2 g_{li} L^{ij}\,\mathrm{d}h_j + \mathrm{d}f_k\, L^{kl}\mathrm{d}g_i \frac{\partial L^{ij}}{\partial z_l}\,\mathrm{d}h_j - \mathrm{d}f_k\, L^{kl}\mathrm{d}^2 h_{lj} L^{ji}\mathrm{d}g_i\,, \tag{2.29}$$

$$= F_g(f,h) + \mathrm{d}f_k\, L^{kl}\mathrm{d}g_i \frac{\partial L^{ij}}{\partial z_l}\,\mathrm{d}h_j - F_h(f,g)\,, \tag{2.30}$$

where $F_g(f,h) = \mathrm{d}f_k\, L^{kl}\mathrm{d}^2 g_{li} L^{ij}\,\mathrm{d}h_j$. Note that $F_g(f,h) = F_g(h,f)$, since L is skew-symmetric and Hess matrix $\mathrm{d}^2 g_{ij}$ is symmetric. By plugging 2.30 into 2.18 the terms with F_f, F_g, F_h cancel out and we will be left with

$$\left(L^{kl}\frac{\partial L^{ij}}{\partial z_l} + L^{jl}\frac{\partial L^{ki}}{\partial z_l} + L^{il}\frac{\partial L^{jk}}{\partial z_l}\right)\mathrm{d}f_k\,\mathrm{d}g_i\,\mathrm{d}h_j = 0\,. \tag{2.31}$$

This has to hold for any choice of f, g, h and thus we arrive at

$$J_{ijk} = L^{kl}\frac{\partial L^{ij}}{\partial z_l} + L^{jl}\frac{\partial L^{ki}}{\partial z_l} + L^{il}\frac{\partial L^{jk}}{\partial z_l} = 0\,, \tag{2.32}$$

where J_{ijk} is called the Jacobiator.

Note that for one-dimensional systems, where $L : \mathbb{R}^2 \to \mathbb{R}^{2\times 2}$, is Jacobi's identity always satisfied, since

$$J_{111} = J_{222} = 0 \tag{2.33}$$

$$J_{211} = J_{121} = J_{112} = L^{12}\frac{\partial L^{21}}{\partial z_2} + L^{12}\frac{\partial L^{12}}{\partial z_2} = L^{12}\frac{\partial L^{21}}{\partial z_2} - L^{12}\frac{\partial L^{21}}{\partial z_2} = 0 \tag{2.34}$$

$$J_{122} = J_{212} = J_{221} = L^{21}\frac{\partial L^{21}}{\partial z_1} + L^{21}\frac{\partial L^{12}}{\partial z_1} = L^{21}\frac{\partial L^{21}}{\partial z_1} - L^{21}\frac{\partial L^{21}}{\partial z_1} = 0\,, \tag{2.35}$$

since $L^{11} = L^{22} = 0$ due to skew-symmetry of L.

2.4 3D Hamiltonian systems

Let us start with an observation.

$$\begin{pmatrix} 0 & -J_1 & J_2 \\ J_1 & 0 & -J_3 \\ -J_2 & J_3 & 0 \end{pmatrix}\cdot v = \begin{pmatrix} J_1 \\ J_2 \\ J_3 \end{pmatrix} \times v\,. \tag{2.36}$$

If $\mathbf{z} \in \mathbb{R}^3$ we have an alternative formulation of Hamiltonian dynamics. Since skew-symmetry of L is guaranteed we need only to take care of Jacobi's identity. In Martin Šípka [2023] it was rewritten as

$$\mathbf{J}\cdot(\boldsymbol{\nabla}\times\mathbf{J}) = 0\,, \tag{2.37}$$

The solution of this equation takes the form of

$$\mathbf{J} = \phi \boldsymbol{\nabla} C \tag{2.38}$$

for $\phi, C \in C^1(\mathbb{R}^3)$. If we construct L in this manner using C and ϕ, it will automatically fulfil Jacobi's identity. C also has the following interesting property. $\forall f \in C^1(\mathbb{R}^3)$

$$\{f, C\} = \boldsymbol{\nabla} f L \boldsymbol{\nabla} C = \boldsymbol{\nabla} f \cdot \phi \boldsymbol{\nabla} C \times \boldsymbol{\nabla} C = 0\,. \tag{2.39}$$

this makes it a so-called Casimir function. Their existence is a result of the degeneracy of the system, which, since skew-symmetric $L \in \mathbb{R}^{3\times 3}$ is always singular, is to be expected. They are also an integral of motion as a result of 2.39 by choosing $f = H$.

In the following chapter, we will combine the information from the last two chapters. This will finally bring us to the main topic of this thesis - Direct Poisson Neural Networks.

3. Direct Poisson Neural Network

Direct Poisson Neural Network was developed by Martin Šípka [2023][1] with a goal to learn the Poisson structure and Hamiltonian from provided data, which can be either simulated or a result of numerical computation. In this chapter, we will explain the inner workings of DPNN and show some of its results.

3.1 The Architecture and workflow

As mentioned in the first chapter, every neural network is trying to approximate some complicated function. In our case, DPNN is trying to approximate $\dot{\mathbf{z}} \approx \Phi(\mathbf{z})$.

It comes in three flavours.

- Without Jacobi (WJ)
- Soft Jacobi (SJ)
- Implicit Jacobi (IJ)

Without Jacobi and Soft Jacobi

The first two, use the same architecture. They learn the Hamiltonian and the upper triangle of L. They differ in the loss function used during the training phase.

$$\mathcal{L}_{\mathrm{WJ}}(\theta) = \sum_{\mathbf{z}_n \in \mathcal{D}} \left\| \frac{\mathbf{z}_{n+1} - \mathbf{z}_n}{dt} - \frac{\Phi_\theta(\mathbf{z}_n) + \Phi_\theta(\mathbf{z}_{n+1})}{2} \right\|^2, \tag{3.1}$$

$$\mathcal{L}_{\mathrm{SJ}}(\theta) = \mathcal{L}_{\mathrm{WJ}}(\theta) + \sum_{\mathbf{z}_n \in \mathcal{D}} \sum_{ijk} \left| J_{ijk}(\mathbf{z}_n) \right|^2. \tag{3.2}$$

While both WJ and SJ have to learn Jacobi's identity from data, the additional regularisation in SJ makes it adapt it faster. The loss function for WJ by design resembles the Cranck-Nicolson scheme, but any numerical scheme could be used in this place - more on that in the next chapter.

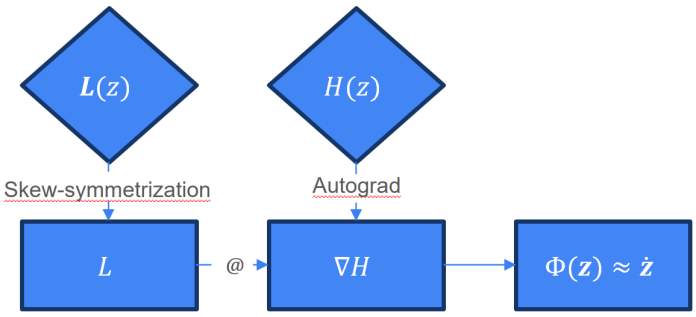


Figure 3.1: The architecture of WJ and SJ networks. The diamonds represent the actual neural networks, which get trained. The rest symbolises the computation process.

[1] Original authors' code we based our thesis on, is at Pavelka [2023].

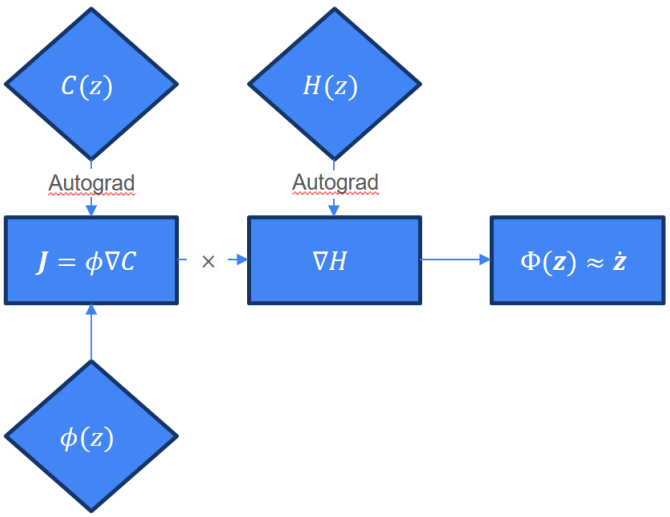


Figure 3.2: The architecture of IJ network. The diamonds represent the actual neural networks, which get trained. The rest symbolises the computation process.

Implicit Jacobi

IJ on the other hand is instead of the upper triangle of L learning C and ϕ as mentioned in section 2.4. Using the procedure described in this section we can produce L which implicitly fulfills Jacobi's identity, hence its name. This however makes it usable only on three-dimensional systems.

Loss function for IJ is the same as for WJ, since there is no need to enforce Jacobi's identity anymore.

Functions H, L, C, ϕ are all represented as a neural network with a default width of 64 nodes and two fully connected layers with softplus activation function.

3.1.1 The workflow

First, the data to train on needs to be generated, since we do not have any measured real-world data. They are generated in the simulation phase before the training. The initial conditions are randomly sampled from a ball (by default 100 of them is taken) from the phase space, whose radius is two times the Euclidean norm of the provided initial conditions. Each trajectory is then numerically simulated using Runge-Kutta of 4th order[2] and stored into the dataset. This dataset needs to be sufficiently dense in the region of phase space, on which we want to make simulations using DPNN. We will show later what happens if the simulated solution escapes the training region of phase space.

These data are then split into training and test sets, the former of which is used for training. All three flavours of DPNN are subsequently trained on the training set. The whole dataset is used in each epoch. The number of epochs required to properly train the network depends on the sample size. The bigger the sample the fewer epochs are needed. Each network has a width of 64 neurons and 2 hidden layers by default

After training, the networks are used to simulate trajectories based on the different randomly sampled initial conditions. DPNN provides an approximation of $\dot{\mathbf{z}}$ with

[2]The original authors used forward Euler or Implicit midpoint rule, which are of order 1 and 2 respectively. To evaluate the performance of DPNN we implemented a method of higher order.

which, using the numerical method of choice[3], the next step in each trajectory is calculated. Apart from network-generated trajectories, grand truth trajectories are simulated using a solver to later serve as a comparison to DPNN's results in error tally.

3.2 Demonstration

In this section, we will demonstrate DPNN's capabilities to learn and simulate. All systems in this thesis were implemented by the author, except for Free rigid body, which was implemented by Martin Šípka [2023].

Coupled harmonic oscillator

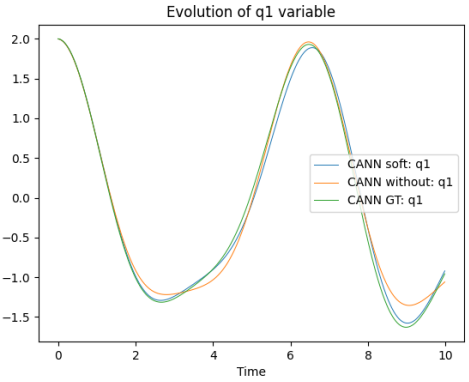
As the first system, we chose a coupled harmonic oscillator with the following Hamiltonian.

$$H = \frac{p_1^2}{2} + \frac{p_2^2}{2} + \frac{q_1^2}{4} + q_2^2 + \frac{(q_1 - q_2)^2}{2}. \tag{3.3}$$

As we have shown in Chapter 2, in one-dimensional systems every skew-symmetric L satisfies Jacobi's identity, which makes soft Jacobi regularisation pointless. Hence the two-particle system.

To train and simulate canonical systems we implemented two additional classes `Canonical` and `GeneralNeural` in `RigidBody.py`. In theory, by changing the `H` parameter, `Canonical` can numerically solve any canonical system. `GeneralNeural` on the other hand serves to produce "neural solutions" of the system. It common numerical methods to simulate trajectories from trained Hamiltonian and Poisson bivector networks. Results from these two classes will be indicated by the 'CANN' label in the figures. Our version of code can be found at Our GitHub repository, commit a802297[4].

After a short experimentation period, a network of width 32 and 4 hidden layers was used. With 5 epochs of training, we obtained results depicted in graphs below



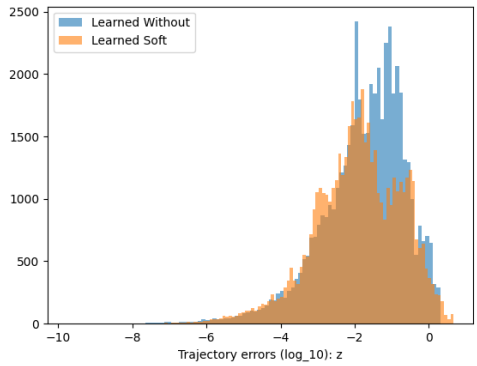


Figure 3.3: Plot of generated trajectories.Figure 3.4: Tally plot of log10 of errors obThe initial conditions were $\mathbf{z} = (2, 0, 0, 0)$ tained from sampling after training.

[3] So far forward Euler which is also a default choice, Crank-Nicholson scheme and implicit midpoint rule have been implemented.

[4] Benda [2024]

From 3.6, we can observe that the trajectory produced by SJ follows the grand truth (GT in the graph) trajectory more closely, hinting that SJ indeed learns faster, at least as far as the number of epochs is concerned. This is further illustrated in the tally plot of error sizes in 3.7.

But as far as training time is concerned the benefit of the Jacobiator regularisation is questionable because these five epochs SJ took approximately 50% longer than WJ. The reason is that the calculation of Jacobiator regularisation includes calculations of the gradient of L using Pytorch's autograd, which is computationally intensive.

Let us now focus on the evolution of learned Hamiltonians along simulated trajectories. We can see that they do not coincide. This is not wrong since

$$\dot{\mathbf{z}} = L \cdot \nabla H\,, \tag{3.4}$$

$$= \frac{L}{C_1} \cdot \nabla(C_1 H + C_2)\,. \tag{3.5}$$

$$= \tilde{L} \cdot \nabla \tilde{H}\,. \tag{3.6}$$

An affine transformation of Hamiltonian may therefore produce the same dynamics, as long as it was compensated by appropriately scaling L.

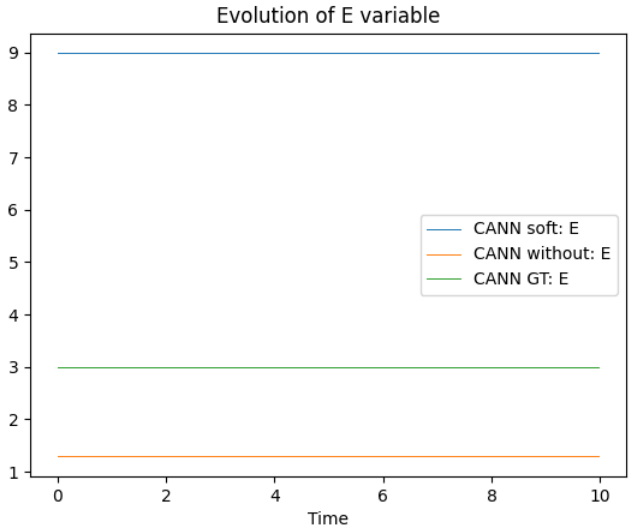


Figure 3.5: Hamiltonian evolution along trajectories

If we were to plot the train and test loss we may conclude that the model could have continued learning, further decreasing the overall error to the point where the trajectories in 3.6 would be indistinguishable. The network learned the dynamics of a two-particle harmonic oscillator fairly fast in comparison with other systems as we will see later.

Free rigid body rotation

As a second demo, we chose an already implemented system describing the rotation of a free rigid body. The purpose of this demo is to demonstrate the usefulness of IJ, which can unfortunately be used only on 3-dimensional non-canonical systems. The dynamics of the free rigid body are governed by the following equation of motion.

$$\dot{\mathbf{m}} = \begin{pmatrix} 0 & -m_3 & m_2 \\ m_3 & 0 & -m_1 \\ -m_2 & m_1 & 0 \end{pmatrix} \cdot \nabla H\,, \qquad H = \frac{m_1^2}{2I_1} + \frac{m_2^2}{2I_2} + \frac{m_3^2}{2I_3}\,. \tag{3.7}$$

Again, networks of width 32 with 4 hidden layers were used. This time, 50 epochs were used, because non-linearity of 3.7 complicates the learning process and also sparser sampling was used.

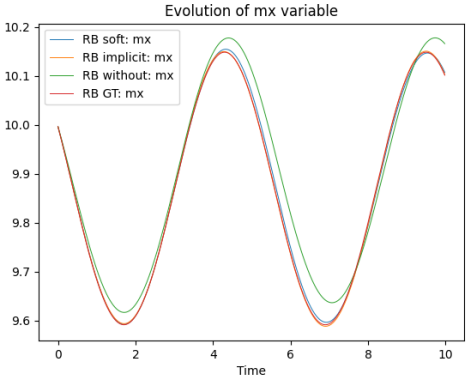


Figure 3.6: Plot of generated trajectories. The initial conditions were $\mathbf{m} = (10, 3, 4)$.

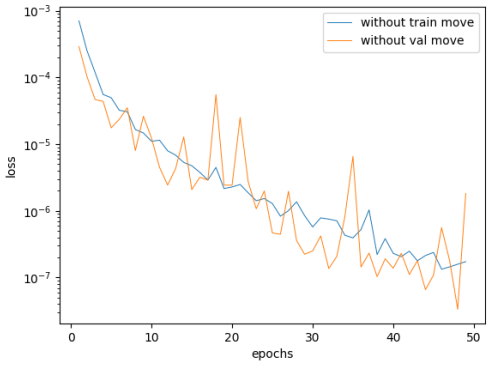


Figure 3.7: Log-plot of the evolution of train and test loss of WJ. IJ and SJ converged even quicker in the beginning.

We can observe that SJ and IJ trajectories follow the grand truth trajectory more closely than WJ. This is even better illustrated in the tally plot 3.8. It is apparent from the tally, that IJ performs even better than both SJ and WJ. It also took the least time to train - only 10 minutes, while SJ and WJ took 30 minutes and 15 minutes respectively. We think that the short training time of IJ is a consequence of its simpler structure.

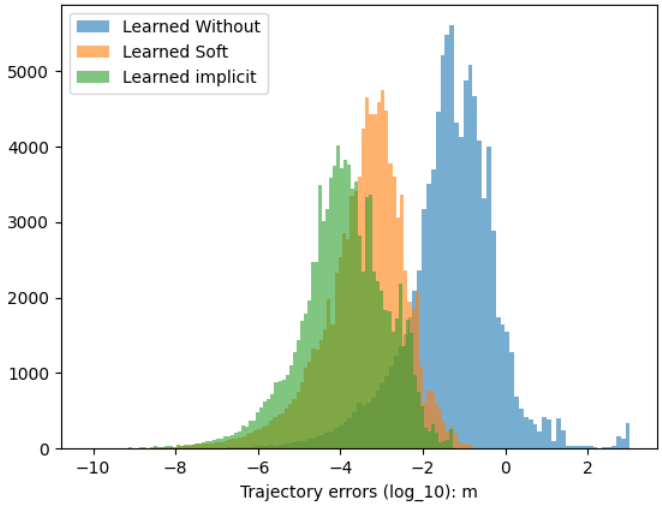


Figure 3.8: Tally plot of log10 of errors obtained from sampling after training. IJ tends to produce the smallest errors, followed by SJ and finally WJ.

Besides synthesising new data from those already measured, it is also possible to extract information about the underlying Hamiltonian and Poisson bivector, which may be useful for further analysis of the system. If we take a look at the plot of a slice of Hamiltonian along the m_z axis (Figure3.9) we can see that it looks like a surface of a paraboloid, which is concise with 3.7.

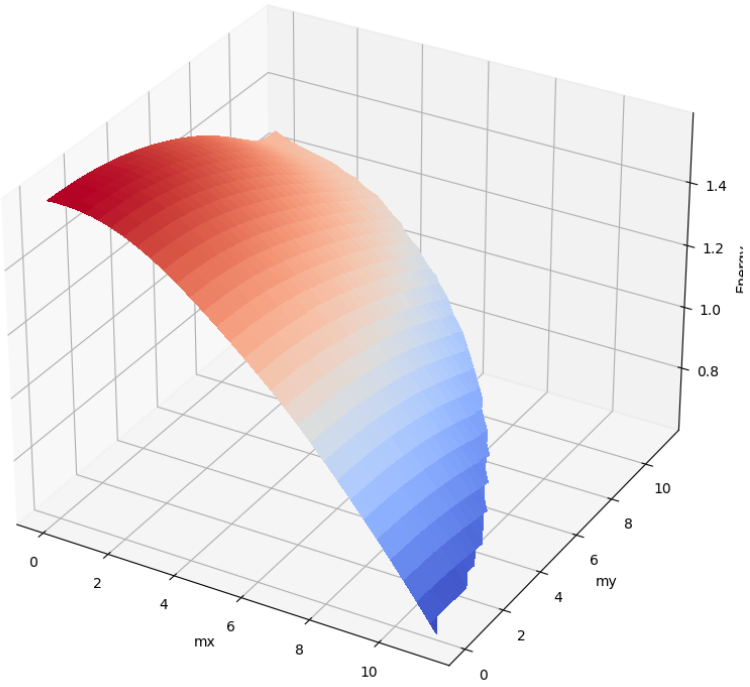


Figure 3.9: Plot of the Hamiltonian along the m_z axis.

Paraboloid is upside-down and shifted but that is again due to the degrees of freedom discussed in the demo with harmonic oscillator.

4. Results and improvements

In the last Chapter, we have introduced the Direct Poisson Neural Network, explained its three modes and their inner workings and demonstrated its generative capabilities on two physical systems. In this chapter we are going to experiment with it, by putting it in different settings, suggest some improvements here and there, and by the end show how to make it recognise systems with dissipation.

4.1 Robustness against noise

DPNN is intended to be eventually trained on datasets obtained by sets of measurement, which will inevitably contain some level of noise. This section is dedicated to the exploration of its effects on training and simulation.

During the dataset generation, we added Gaussian noise $\varepsilon \sim \mathcal{N}(0, \sigma^2)$ to each generalised coordinate just before we save it to the dataset[1]. To produce any meaningful results we should control the relative size of the noise to values of data points. For this reason, we decided to go with a one-dimensional harmonic oscillator as our system for this demo with the following Hamiltonian

$$H = \frac{p^2}{2} + \frac{q^2}{2}\,.$$

Trajectories in the phase space produced by this Hamiltonian are circular. Thus we have direct control over the size of $\mathbf{z}$ along the trajectory since $||\mathbf{z}|| = ||\mathbf{z}_0||$.

Observation

We run the simulation for five different values of σ. The training dataset consisted of 20 randomly sampled trajectories, each 1000 steps long. Network width and depth were the same as in demonstrations in chapter three.

As is to be expected, the higher the σ was the worse the predictions were. Instead of an error tally, we will use a median of errors calculated from simulated trajectories. Also, introducing noise to the dataset had a huge effect on when the network stopped learning. This can be observed at loss function "convergence". Eventually, it stops decreasing and starts fluctuating around a certain value as can be seen in 4.3.

$\sigma/\|\|\mathbf{z}_0\|\|$	0	0.02	0.05	0.10	0.20	0.50
WJ (med. error)	$1{,}0{\cdot}10^{-4}$	$2{,}4{\cdot}10^{-2}$	$3{,}3{\cdot}10^{-2}$	$3{,}4{\cdot}10^{-1}$	$2{,}1{\cdot}10^{-1}$	2,3
SJ (med. error)	$4{,}5{\cdot}10^{-4}$	$7{,}9{\cdot}10^{-2}$	$1{,}9{\cdot}10^{-2}$	$6{,}6{\cdot}10^{-2}$	$9{,}3{\cdot}10^{-1}$	$9{,}4{\cdot}10^{-1}$

Table 4.1: Median errors for different levels of noise.

Notice that the intrinsic model error[2] gets quickly dominated by error introduced by noise in the dataset, which may be orders of magnitude larger. This decreases the quality of simulated trajectories as can be seen in 4.2.

[1] In the next step, the values without the added perturbation are used

[2] An inference error a model makes, when trained on noise-free dataset

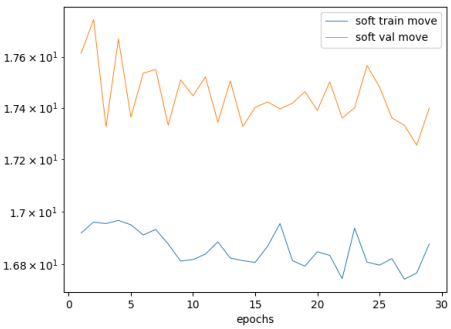


Figure 4.1: Stagnation of loss for SJ with $\sigma = 0,02$. Loss values remain relatively high in comparison with noise-free demos in Chapter 3.

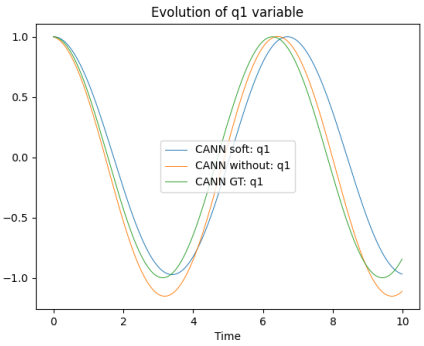


Figure 4.2: Position variable evolution. The training was done on a noised dataset with $\sigma = 0,02$. If there is no noise present, trajectories look identical.

The results were even less precise for the rigid body system with noise. With the same setting as in the demo in chapter 3, only with $\sigma = 0,05$, error medians jumped from values $\sim 10^{-3}$ to values close to one. In other words a perturbation of a relative size of $0,05/||\mathbf{z}_0|| \doteq 4 \cdot 10^{-3}$ produced a simulation error of relative size $0,9/||\mathbf{z}_0|| \doteq 8 \cdot 10^{-2}$

Compensating for the noise

Unfortunately, we were not able to significantly decrease the effect of the noise. DPNN already uses a MSE loss and identity as the last layer activation, which, as we discussed in the first chapter, should be the first thing to try.

Surprisingly, increasing the sample size had a negligible effect. Using a sample size 5 times larger with $\sigma = 0,02$ produced median errors $2,6 \cdot 10^{-2}$ and $4,4 \cdot 10^{-2}$ for WJ and SJ respectively, which are comparable with values in the corresponding column in the table 4.1.

Using dropout cut down the error median roughly by a factor of 10. In the setting of the harmonic oscillator, the best results were obtained for the dropout rate $p = 0,3$.

Lastly, feeding the average of neighbouring data points instead of the data points themselves into the network, would decrease the noise in the dataset. However, we were not able to implement and test this feature due to the complexity of the existing codebase.

Summary

DPNN is, as we have just shown, very susceptible to noise in the dataset. Even small fluctuations propagate and degrade the quality of simulated data. Finally, we have proposed some means of bolstering the network against the noise, enhancing its performance on real-world datasets.

4.2 Extrapolation

So far, we have simulated systems on regions of phase space, on which the neural network had plenty of data points to train on. As we move further from this region, which we will refer to as the training region in this section, the approximations of the Hamiltonian and Poisson bivector get worse and worse - until they become basically a random guess. Let us now explore this behaviour.

For this demo, we decided to use a simple pendulum system with Hamiltonian

$$H = \frac{p^2}{2} - \cos q\,.$$

Unlike Hamiltonians, which we have considered so far, this one is not convex and therefore some of its trajectories are not bounded. If $p_{\text{init}} > 2$, the pendulum will make a full rotation around its axis, always increasing in q variable Therefore it does not stay in any part of the phase space but rather wander off to infinity in the direction of q coordinate.

The baseline configuration for this section was $(p_{\text{init}}; q_{\text{init}}) = (2, 5; 0)$, 30 epochs of training, dataset size of 20 trajectories - each 500 steps long. Time step dt was taken to be as 0.01. Each trajectory in the dataset was therefore 5 time units long. Trajectories produced by the network were twice as long with the same time step.

To evaluate the error growth we plotted the mean square error between network prediction and the grand truth. The red vertical line marks the point when the simulation left the training region.

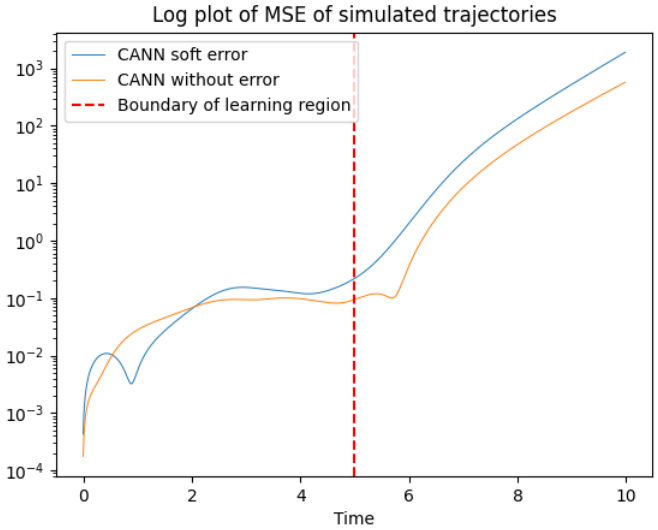


Figure 4.3: Log of MSE between the grand truth and simulated trajectories.

Within the training region, the errors for both SJ and WJ remain relatively small, upon crossing the boundary the error remains smaller than 1 for roughly one unit of time and then starts growing exponentially.

Improving the extrapolation

In order to improve the performance of the network outside of the training region we tried adding quadratic features to the Hamiltonian network.

Feature addition is a mapping, that creates new features, which are then used as a network input[3]. In our case, the quadratic feature map would be

$$(p, q) \longmapsto (p, q, p^2, q^2, pq)\,.$$

In the setting of neural network feature engineering is an uncommon practice, but in our case, it is justified since Hamiltonians tend to be asymptotically quadratic.

Using quadratic features improved the performance on the training region by roughly an order of magnitude. But outside of it, it did not make any difference.

Letting the network train for more epochs had the same effect and only took longer.

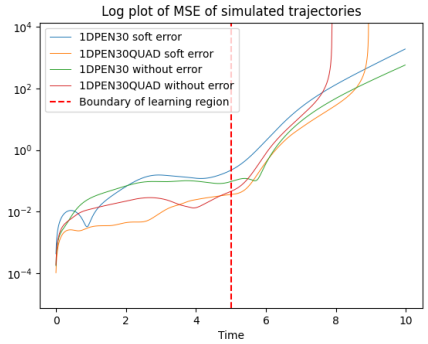


Figure 4.4: MSE log plot comparison for toggled quadratic features.

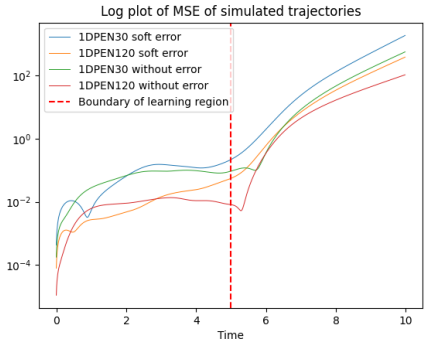


Figure 4.5: MSE log plot comparison for long training time (120 epochs).

Taking a larger sample size, on the other hand, did have a significant effect on the extrapolation. Doubling the training dataset size lowered the prediction error almost by a factor of 10^3. This, in combination with quadratic features, which reduced the error on the training region, produced the best result thus far in terms of extrapolation for DPNN, depicted in 4.6.

4.3 RK4 movement loss

There are in total 3 places in the run of the program, where a numerical scheme is used. First in dataset generation - here RK4 was used. Then in the loss function and lastly during simulation from learned networks. Here RK4 was also used. In this section, we will examine the effects of basing the loss function on different numerical schemes, namely IMR and RK4, and how this choice will influence the learning process.

CN-based loss function was already introduced in chapter 3 as 3.1. IMR-based loss, which was also already implemented by Martin Šípka [2023] takes the form

$$\mathcal{L}_{\mathrm{WJ,IMR}}(\theta) = \sum_{\mathbf{z}_n \in \mathcal{D}} \left\| \frac{\mathbf{z}_{n+1} - \mathbf{z}_n}{dt} - \Phi_\theta\left(\frac{\mathbf{z}_n + \mathbf{z}_{n+1}}{2}\right) \right\|^2 . \tag{4.1}$$

RK4 loss was implemented by ourselves as

$$\mathcal{L}_{\mathrm{WJ,RK4}}(\theta) = \sum_{\mathbf{z}_n \in \mathcal{D}} \left\| \frac{\mathbf{z}_{n+1} - \mathbf{z}_n}{dt} - \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} \right\|^2 , \tag{4.2}$$

[3]Input layer must be appropriately adjusted.

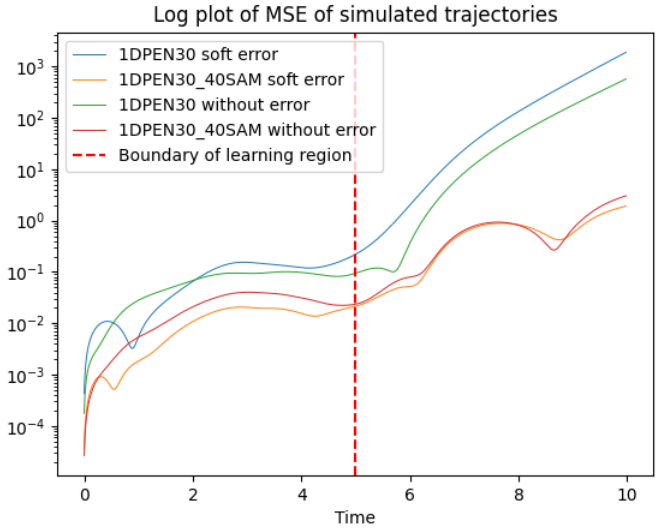


Figure 4.6: MSE log plot comparison between the base case and network trained on larger sample size with quadratic features toggled.

where

$$k_1 = \Phi_\theta(\mathbf{z}_n) \qquad k_2 = \Phi_\theta\Big(\mathbf{z}_n + \mathrm{dt} \cdot \tfrac{k_1}{2}\Big)$$
$$k_3 = \Phi_\theta\Big(\mathbf{z}_n + \mathrm{dt} \cdot \tfrac{k_2}{2}\Big) \quad k_4 = \Phi_\theta(\mathbf{z}_n + \mathrm{dt} \cdot k_3)\,.$$

Both CN and IMR are schemes of second order. Also, IMR is a symplectic integrator. This means that it preserves the phase space volume when used in simulations. Furthermore, while total energy is not conserved, its value tends to fluctuate around its true value as the trajectory evolves. These are the properties guaranteed when it is used to simulate symplectic systems, such as Hamiltonian systems. But there is no reason for it to have similar desired properties when it is used as a loss function. (As for example in Hairer et al. [2006])

On the other hand, we think that using fourth-order Runge-Kutta-based loss may have some. One iteration requires evaluating Φ_θ at four different locations, which introduces three more points into the calculation of the loss. We think that should effectively quadruple the dataset size.

We believe that the order of method, the loss was based on, does not affect the speed of convergence of the loss. However, we think that it influences the minimal achievable loss, which, in the best case scenario, will align with the local error of the scheme.

Observations

We used a double pendulum system with $l_1 = l_2 = m_1 = m_2 = g = 1$ to test our hypotheses. The following Hamiltonian was used.

$$H = \frac{1}{2}\frac{p_1^2 + 2p_2^2 - 2p_1p_2\cos(q_1 - q_2)}{1 + \sin^2(q_1 - q_2)} - 2\cos(q_1) - \cos(q_2) \tag{4.3}$$

The double pendulum system is chaotic. This means that any arbitrarily small perturbation of initial conditions or along the trajectory will eventually cause a divergence

from the exact solution. To numerically solve this system, more precise numerical schemes should be used. This makes it a good candidate to test method precision on.

All the runs were made with the inital condition $(q_1, q_2, p_1, p_2) = (1.56, -1.56, 0, 0)$ - the pendulum started in an almost upright position. The dataset size was chosen to be 50 trajectories, 500 steps each. Network width and depth were the same as in demonstrations in chapter three and were trained for 60 epochs.

Training DPNN on a double pendulum system took much longer than on systems considered so far. Also, the performance of SJ was far better than that of WJ, which even after 60 epochs of training quite often produced diverging trajectories. Therefore comparisons were made only on SJ.

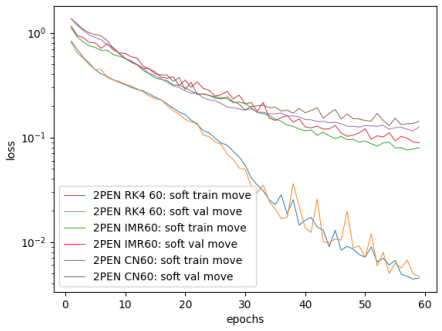


Figure 4.7: Comparison of the loss convergence between CN, IMR and RK4 based SJ.

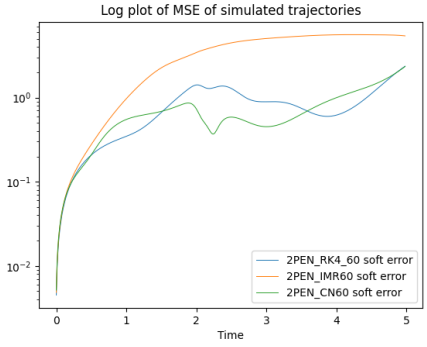


Figure 4.8: Comparison of trajectory divergence between CN, IMR and RK4 based SJ.

As we can see in figure 4.9, while CN and IMR-based losses seem to start to stagnate, RK4 loss keeps decreasing, indicating, that it could have been trained longer. This might be a consequence of both the method order limitation and the artificial sample size multiplication described earlier. Trajectories generated by IMR-based SJ tended to diverge faster than those generated by RK4 and CN-based SJ. As can be seen in 4.10.

Training the network with CN and IMR-based losses on datasets four times as big improved the convergence of losses and decreased the divergence from the grand truth solution, to the point, where they become comparable with RK4-based SJ[4]. While this does not prove nor disprove our hypothesis, it hints at its plausibility.

Summary

We introduced RK4-based loss and compared it with CN and IMR-based losses. It seems that using RK4-based loss has the same effect as using CN or IMR-based losses with quadruple sample size, and only takes approximately 50 % more time per epoch to compute. One way to test this hypothesis would be to implement higher-order Runge-Kutta methods, like the DOPRI method, which is of fifth order and uses 7 different evaluations of the right-hand side function.

[4] Atl least at the beginning of the training process

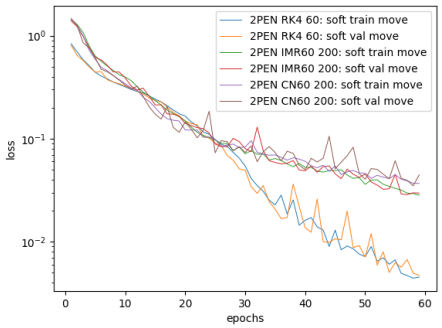


Figure 4.9: Comparison of the loss convergence between CN, IMR and RK4-based SJ. CN and IMR-based SJ used quadruple dataset size.

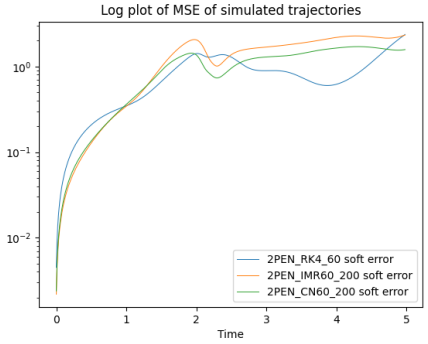


Figure 4.10: Comparison of trajectory divergence between CN, IMR and RK4 based SJ. CN and IMR-based SJ used quadruple dataset size.

4.4 Ehrenfest dissipation

Every real-world physical system exhibits some sort of dissipative behaviour, be it external or internal friction, radiation, etc. Each dissipative mechanism works uniquely and as far as we are aware, there is no singular description, that would describe them all. in this demo, we have decided to use Energetic Ehrenfest Regularisation described in Pavelka et al. [2019] and implement it within the rigid body system framework. The regularisation is given by

$$\dot{\mathbf{m}} = \mathbf{L}\nabla H + \frac{\tau}{2}\mathbf{L}\mathbf{H}_H\mathbf{L}\nabla H\,, \tag{4.4}$$

where τ is a parameter of regularisation dubbed relaxation time and $\mathbf{H}_H$ is Hessian matrix of the Hamiltonian.

It can be shown easily that

$$\dot{H} = \nabla H^\mathsf{T}\dot{\mathbf{m}} = \nabla H^\mathsf{T}\mathbf{L}\nabla H + \frac{\tau}{2}\nabla H^\mathsf{T}\mathbf{L}\mathbf{H}_H\mathbf{L}\nabla H = \{H,H\} - \frac{\tau}{2}\dot{\mathbf{m}}^\mathsf{T}\mathbf{H}_H\dot{\mathbf{m}} < 0\,. \tag{4.5}$$

This only holds if H is a convex function, which for free rigid body rotation it is. The regularisation is therefore dissipative.

Moreover, we can show that the Energetic Ehrenfest regularisation preserves the Casimir functions of the original dynamics.

$$\frac{\mathrm{d}C}{\mathrm{d}t} = \nabla C\cdot\dot{\mathbf{z}} = \nabla C\left(\mathbf{L}\nabla H + \frac{\tau}{2}\mathbf{L}\mathbf{H}_H\mathbf{L}\nabla H\right) = 0\,, \tag{4.6}$$

since by definition of Casimir function $L\nabla C = 0$.

Since for the rigid body system the Casimir function is $C = \left\|\mathbf{m}^2\right\|$,[5] we should see that no matter with which initial conditions the system starts, it will end up rotation around the axis with the highest moment of inertia.

Implementation and training

The regularised version of Hamilton's equations was implemented in a similar fashion as the non-regularised version in Chapter 3, with the addition of a new trainable

[5] Which can be easily verified by plugging it into the formula above

parameter τ.

The Hessian of H was computed using Pytorch's autograd, which also supports the calculation of Hessians. It is a very expensive operation that makes the evaluation of Φ take much longer. For this reason, we chose to use IMR-based loss, since it does only one Φ evaluation per sample.

All arguments had to be chosen carefully. We wanted to capture the whole dissipation process in the simulation while making sure that the network gets trained sufficiently well in a reasonable amount of time. This was achieved with $\mathbf{m} = (10, 3, 4)$, $\mathbf{I} = \text{diag}(1, 2, 16)$ and $\tau = 0.01$. The architecture of H, L, C, ϕ networks had to be changed to have only 2 hidden layers and a width of 64 neurons. The number of epochs was 50. To have a good sample coverage, we chose sample size = 500 but set the length of sample trajectories to be 100 steps. The simulation trajectory was 1000 steps long, dt was 0.02 in both cases.

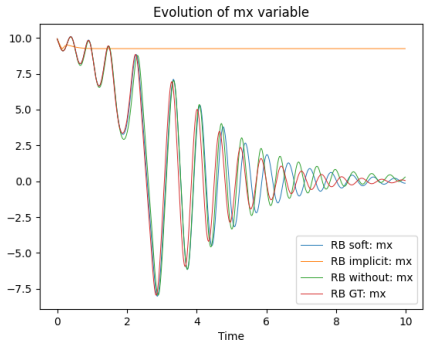


Figure 4.11: Solutions produced by trained models. Only m_x is plotted.

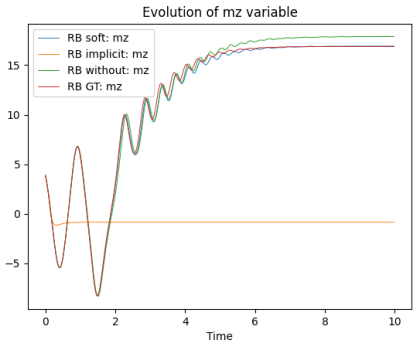


Figure 4.12: Solutions produced by trained models. Only m_z is plotted.

After 6 hours of training, we obtained following results. As it was said in previous section, the rigid body eventually started rotating around the axis with the highest moment of inertia as can be seen in 4.11 and 4.12. According to 4.13, the best results were obtained by SJ, than by WJ and lastly by IJ, apparently has not learnt the dynamics at all.

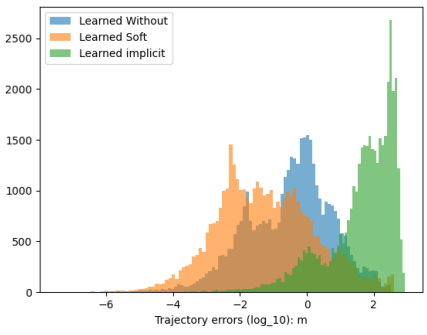


Figure 4.13: Tally plot of errors for RB with dissipation

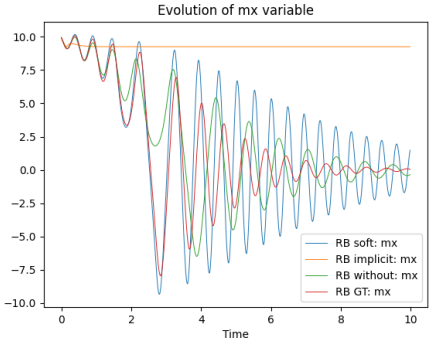


Figure 4.14: The plot of m_x produced by trained model without the dissipation regularisation.

The learnt values of τ were $-8.6 \cdot 10^{-3}$, $7.2 \cdot 10^{-3}$ and $-1.1 \cdot 10^{-2}$ for WJ, SJ and IJ respectively. While at least WJ and SJ have reproduced the dynamics correctly, they

both failed in learning the τ parameter. The reason for this is, that H and L have somehow learnt to mimic the dissipative behaviour. By training H and L networks with the regularisation enabled, disabling it for the simulation, we produce trajectories, which look as if there was dissipation involved. But this cannot be, because in all cases L is skew-symmetric and, as we have discussed in Chapter 2, this implies that the energy is conserved. A similar phenomenon has been described in Martin Šípka [2023], where the authors were training plain DPNN on data capturing similar dissipative behaviour.

As to why IJ had such a trouble learning the dissipative dynamics, we do not have any explanation. We suspect that it may have something to do with the stability of the learning process. Some choices of arguments lead to a divergence of the loss values or a stagnation of validation error, suggesting that the network is no longer learning. Changing the architecture of ϕ and C networks, using L2 regularisation on L and H or tweaking the relative learning rate for τ parameter may stabilise the learning process, but due to the approaching deadline of submission of this thesis we were not able to test these options.

Conclusion

We obtained mixed results from implementing the Energy Ehrenfest regularisation. While SJ and WJ could successfully reproduce the behaviour of the system, they were not able to learn the value of the regularisation parameter τ. Introducing the regularisation also lead to distortions in learnt Hamiltonian an Poisson bivector. IJ was not able to learn any of the dynamics at all, which we suspect is due to the instability of learning process. To stabilise it we suggested some options.

Conclusion

In conclusion, this thesis has demonstrated the potential of neural networks for recognising physical systems.

We were able to train the Direct Poisson Neural Network on both canonical and non-canonical Hamiltonian systems and retrieve some information about the governing equations of dynamics, namely the Hamiltonian and Poisson bivector. Then, based on some initial conditions, we used this network to generate new data, which approximately followed the same dynamics as the data, which it was trained on.

In the last chapter we saw, that the noise drastically worsens the predictive capabilities and training dynamics of DPNN. We were able to partially mitigate this problem, by introducing dropout layers into the architecture. Then we discussed, how the predictions get exponentially worse, when a simulated trajectory left the training region of phase space. Increasing the dataset size along with the use of quadratic features for Hamiltonian network decreased the prediction error. Thirdly, we explored how different choices of the loss function influence the training process. RK4 based loss has trained faster than IMR based and Crank-Nicolson based losses. Lastly we implemented the Energy Ehrenfest dissipation for a free rigid body system. While the networks were able to decently reproduce the dynamics, they were not able to learn the parameter τ correctly.

Bibliography

Jan Benda. direct-poisson-neural-networks. https://github.com/JanBenda123/direct-Poisson-neural-networks, 2024. [Software]. Available at https://github.com/JanBenda123/direct-Poisson-neural-networks. Commit a802297.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989.

Ernst Hairer, Christian Lubich, and Gerhard Wanner. *Geometric Numerical Integration*, volume 31. Springer, 2006. ISBN 978-3-540-30663-4. doi: 10.1007/3-540-30666-8. URL https://link.springer.com/book/10.1007/3-540-30666-8.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. *arXiv: Optimization and Control*, 2014.

Oğul Esen Miroslav Grmela Martin Šípka, Michal Pavelka. Direct poisson neural networks: learning non-symplectic mechanical systems. *Journal of Physics A: Mathematical and Theoretical*, 56(49):1–25, 2023. URL https://arxiv.org/pdf/2305.05540.

Michal Pavelka, Václav Klika, and Miroslav Grmela. Ehrenfest regularization of hamiltonian systems. *Physica D: Nonlinear Phenomena*, 399:193–210, 2019. ISSN 0167-2789. doi: https://doi.org/10.1016/j.physd.2019.06.006. URL https://www.sciencedirect.com/science/article/pii/S0167278918305232.

Šípka Pavelka. direct-poisson-neural-networks. https://github.com/enaipi/direct-Poisson-neural-networks, 2023. [Software]. Available at https://github.com/enaipi/direct-Poisson-neural-networks. Commit 72f1117.

Jason Brownlee PhD. Line plots of loss on train and test datasets while training showing an overfit model, 2019. URL https://machinelearningmastery.com/how-to-stop-training-deep-neural-networks-at-the-right-time-using-early-st

Milan Straka. Machine learning for greenhorns [lecture slides]. https://ufal.mff.cuni.cz/~straka/courses/npfl129/2223/slides/?03#1, 2023.

List of Figures

List of Tables