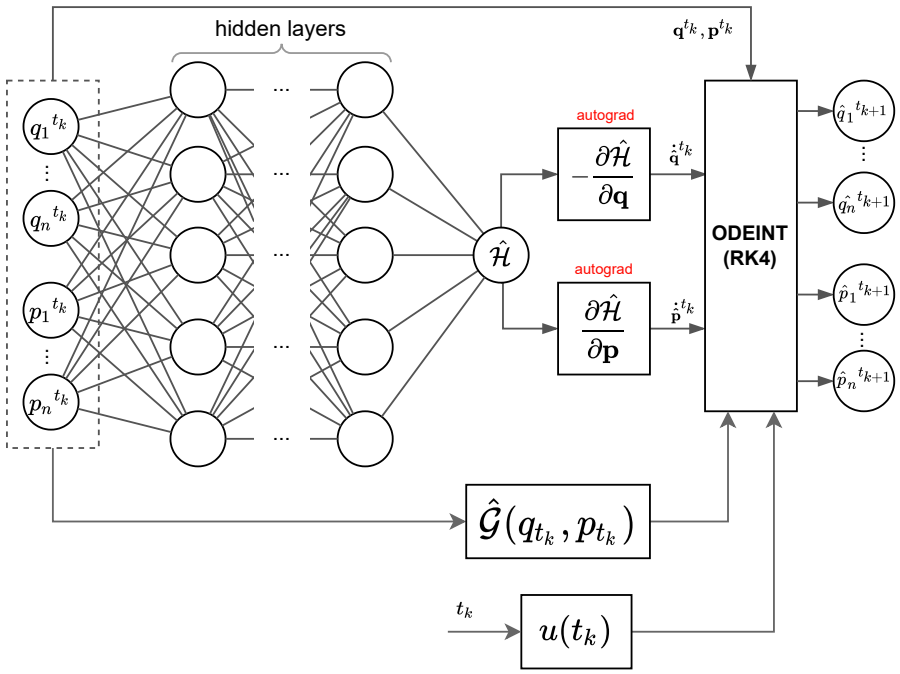
Learning Hamiltonian functions from data

Younes Moussaif - Semester project



Start : 21.02.2022
End : 15.06.2022
Professor : Giancarlo Ferrari Trecate
Assistants : Clara Galimberti and Muhammad Zakwan

Contents

List of Figures

1 Introduction

Recent advances in physics-informed neural networks have been successful at using neural networks to capture the dynamics of physical systems such as the simple and double pendulum[1][2][3][4]. These neural networks are endowed with physics-based inductive biases by incorporating Hamiltonian dynamics in their architecture. These architectures leverage neural ordinary differential equations [5] for this purpose. The goal of this project is to design physics-informed architectures based on the port-Hamiltonian framework to model dynamical systems. The simple pendulum and the Furuta pendulum were selected for this purpose.

The report is structured as follows: The Background and related work section presents the port-Hamiltonian framework and architectures existing in the current literature. The methods section details the architectures used in this project, and how they are trained. The Experiments, results and discussion section clarifies how the experiments on the proposed architectures were carried out and their results.

2 Background and related work

This section presents the port-Hamiltonian formalism along with related work done in the field of introducing physics-based inductive biases in deep learning and system identification.

2.1 Background

Hamiltonian dynamics describe a physical system using the coordinate system $(\mathbf{q}, \mathbf{p})$, where $\mathbf{q} = (q_1, q_2, \ldots, q_n)$ are the generalized coordinates and $\mathbf{p} = (p_1, p_2, \ldots, p_n)$ are the generalized momenta. The coordinates $\mathbf{p}$ are related to $\mathbf{q}$ by the following formula, where $\mathcal{L}(q, \dot{q})$ is the Lagrangian :

$$\boldsymbol{p} = \frac{\partial \mathcal{L}(q, \dot{q})}{\partial \dot{q}}. \tag{1}$$

Using the Lagrangian, the Hamiltonian can be derived as :

$$\mathcal{H}(\mathbf{p}, \mathbf{q}) = \sum_{i=1}^{N} p_i \dot{q}_i - \mathcal{L}(\mathbf{q}, \dot{\mathbf{q}}) \tag{2}$$

On the other hand, port-Hamiltonian dynamics are a generalization of Hamiltonian dynamics that incorporate dissipation and input to the dynamical system [3]. For this report, the following formulation of the port-Hamiltonian framework is used :

$$\begin{bmatrix} \dot{\mathbf{q}}(t) \\ \dot{\mathbf{p}}(t) \end{bmatrix} = \left(\begin{bmatrix} \mathbf{0} & \mathbf{I} \\ -\mathbf{I} & \mathbf{0} \end{bmatrix} - \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{D} \end{bmatrix} \right) \begin{bmatrix} \frac{\partial \mathcal{H}(q,p)}{\partial \mathbf{q}} \\ \frac{\partial \mathcal{H}(q,p)}{\partial \mathbf{p}} \end{bmatrix} + \mathcal{G}(\mathbf{q}, \mathbf{p})\ \mathbf{u}(\mathbf{t}), \tag{3}$$

where $\mathbf{D} \in \mathbb{R}^{n \times n}$ is a diagonal matrix that models dissipation. The matrix containing $\mathbf{D}$ is sparse since dissipation is assumed to only affect the generalized momentum. The vector

$\mathcal{G}(\mathbf{q}, \mathbf{p}) \in \mathbb{R}^{2n}$ and scalar $\mathbf{u}(\mathbf{t})$ model the effect of external inputs such as torque. The dot in $\dot{\mathbf{q}}$ represents the derivative with respect to time.

Using numerical integration, we can solve the equation 3 to obtain the states $(\mathbf{q}^{t_{k+1}}, \mathbf{p}^{t_{k+1}})$ if the states at the previous time-step $(\mathbf{q}^{t_k}, \mathbf{p}^{t_k})$ are known:

$$\begin{pmatrix} \mathbf{q}^{t_{k+1}} \\ \mathbf{p}^{t_{k+1}} \end{pmatrix} = \begin{pmatrix} \mathbf{q}^{t_k} \\ \mathbf{p}^{t_k} \end{pmatrix} + \int_{t_k}^{t_{k+1}} \left(\frac{\partial \mathcal{H}(q,p)}{\partial \mathbf{q}}, \ (\mathbf{D} - \mathbf{I}) \frac{\partial \mathcal{H}(q,p)}{\partial \mathbf{p}} \right)^T + \mathcal{G}(\mathbf{q}, \mathbf{p}) \ \mathbf{u}(\mathbf{t}) \ \ dt. \tag{4}$$

Note that when dissipation and input are set to zero in equation 3, the classical Hamiltonian equations are recovered:

$$\begin{bmatrix} \dot{\mathbf{q}}(t) \\ \dot{\mathbf{p}}(t) \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathcal{H}(q,p)}{\partial \mathbf{q}} \\ -\frac{\partial \mathcal{H}(q,p)}{\partial \mathbf{p}} \end{bmatrix}, \tag{5}$$

where the term on the right is called the symplectic gradient.

2.2 Problem Statement

The first objective of this project is to approximate the Hamiltonian function $\mathcal{H}(\mathbf{p}, \mathbf{q})$, using a set of N trajectories $\mathcal{D}_1 = \{[(\mathbf{q}, \mathbf{p})^{t_0, t_1, .., t_K}]_1, [(\mathbf{q}, \mathbf{p})^{t_0, t_1, .., t_K}]_2, ..., [(\mathbf{q}, \mathbf{p})^{t_0, t_1, .., t_K}]_N\}$ sampled at K time-steps from a dynamical system. In this project the simple pendulum and the Furuta pendulum are the studied dynamical systems that will be described in Section 4.1.1 and 4.2.1.

In practice, we often do not have access to the generalized momenta. Therefore, the second objective of the project is to use a set of trajectories containing generalized velocities :

$$\mathcal{D}_2 = \{[(\mathbf{q}, \dot{\mathbf{q}})^{t_0, t_1, .., t_K}]_1, [(\mathbf{q}, \dot{\mathbf{q}})^{t_0, t_1, .., t_K}]_2, ..., [(\mathbf{q}, \dot{\mathbf{q}})^{t_0, t_1, .., t_K}]_N\}.$$

The third objective of this project is to explore learning the Hamiltonian function $\mathcal{H}(\mathbf{p}, \mathbf{q})$, and the input function $\mathcal{G}(\mathbf{q}, \mathbf{p})$ from a set of trajectories containing the time dependent inputs $\mathbf{u}(t)$:

$$\mathcal{D}_3 = \{[(\mathbf{q}, \mathbf{p}, \mathbf{u})^{t_0, t_1, .., t_K}]_1, [(\mathbf{q}, \mathbf{p}, \mathbf{u})^{t_0, t_1, .., t_K}]_2, ..., [(\mathbf{q}, \mathbf{p}, \mathbf{u})^{t_0, t_1, .., t_K}]_N\}.$$

The final objective of this project is to approximate $\mathcal{H}(\mathbf{p}, \mathbf{q})$ using a set that additionally contains $\mathcal{G}(\mathbf{q}, \mathbf{p})$ at every point along with the corresponding input $\mathbf{u}(t)$:

$$\mathcal{D}_4 = \{[(\mathbf{q}, \mathbf{p}, \mathcal{G}, \mathbf{u})^{t_0, t_1, .., t_K}]_1, [(\mathbf{q}, \mathbf{p}, \mathcal{G}, \mathbf{u})^{t_0, t_1, .., t_K}]_2, ..., [(\mathbf{q}, \mathbf{p}, \mathcal{G}, \mathbf{u})^{t_0, t_1, .., t_K}]_N\}.$$

2.3 Related work

In [5], the neural ODE framework is introduced, where a neural network can be used to approximate the function $f(x)$ in a differential equation $\frac{d\mathbf{x}(t)}{dt} = f(\mathbf{x}(t))$. This has been used in the context of dynamical systems by authors in [1]. They present Hamiltonian neural networks (HNNs) which use classical Hamiltonian dynamics. In this method, a neural

network approximates the Hamiltonian function and the symplectic gradients are obtained using automatic differentiation [6]. An illustration of such a model is given in Figure 1.

In [2] and [3] the Symplectic ODE Net (SymODEN), and dissipative SymODEN architectures are proposed. They are similar to the HNN architecture but use port-Hamiltonian dynamics to model dissipation and control. In [2] and [3] a port-Hamiltonian formulation, where the control input u is constant and where $\mathcal{G}$ can depend on the states $(\mathbf{q}, \mathbf{p})$, is used. They also impose a structure on the learned Hamiltonian function that uses two neural networks $\mathbf{M}_{\theta_1}$ and V_{θ_2} :

$$\mathcal{H}(\mathbf{q}, \mathbf{p}) = \frac{1}{2}\mathbf{p}^T \mathbf{M}_{\theta_1}^{-1}(\mathbf{q})\mathbf{p} + V_{\theta_2}(\mathbf{q}). \tag{6}$$

In [7], an autoencoder architecture, dissipative SymODEN and other models are discussed in the context of the simple and Furuta pendulum. The autoencoder model from [7] extends the HNN architecture to use trajectories in the form $\mathbf{q}, \dot{\mathbf{q}}$. We continue this work by implementing the autoencoder on the Furuta pendulum and by proposing other architectures.

In [4], a time dependent input is introduced by replacing the $\mathcal{G}(\mathbf{q}, \mathbf{p})\mathbf{u}(t)$ term in the port-Hamiltonian equations with $\mathbf{F}(t)$. This formulation is close to the one presented in Equation (3).

In [8], non-linear dynamics such as the ones of a double pendulum are modelled using a neural ODE. The results in this paper seemed promising but their training approach is different from the previous papers. Furthermore, their code still has not been made public at the time this report was written.

Finally, the authors in [9] present a multi-level strategy to stabilize training for deep neural networks. The proposed method begins with a model that has a small number of parameters. During training, new model parameters are introduced and initialized using linear interpolation. For example, one can start training with a residual network consisting of two residual blocks. Eventually, during the training procedure, a new residual block is introduced between the two previous ones. The parameters of the new residual block are initialized with the average of its two neighbouring residual blocks. These methods are discussed more in depth in Section 3.4.6. Although the examples presented in [9] were classification problems, the presented methods could be extended to Hamiltonian neural networks.

While there are other numerical integration packages that implement neural ODEs such as `torchdyn`, ODEint from the `torchdiffeq` package will be used for that purpose in this project.

2.4 Contribution

The main contribution presented in this report is first implementing the Autoencoder based HNN architecture presented in [7] for the Furuta pendulum. Then, using the port-Hamiltonian formulation, a method to learn a state dependent input function $\mathcal{G}(\mathbf{q}, \mathbf{p})$ is presented and evaluated. Finally, multi-level strategies that introduce new parameters and ways to initialise

them during the training procedure are presented and used to try to improve the results. Unfortunately, some of these strategies did not lead to considerable improvements. This could be due to the chaotic nature of the Furuta pendulum, which could make learning its true underlying dynamics a challenging task for a neural network.

3 Methods

This section details the main architectures used in this project along with the challenges that were encountered when training them and the strategies used to overcome those challenges.

The Autoencoder-HNN architecture uses trajectories that have the state space representation $(\mathbf{q}, \dot{\mathbf{q}})$. Both the Simple-HNN and Input-HNN architectures use the state space representation $(\mathbf{q}, \mathbf{p})$, since the objective of these architectures is to evaluate learning procedures for a model that implements input.

3.1 Simple-HNN

The Simple Hamiltonian Neural network (Simple-HNN) architecture is the same as the one proposed in [1], where the Hamiltonian function is approximated by an MLP and has no structure imposed to it contrary to the SymODEN architecture[3]. As illustrated in Figure 1, the MLP in the Simple-HNN architecture takes as input the vectors $(\mathbf{q}, \mathbf{p})$ at time t_k and outputs $\hat{\mathcal{H}}$, which is an approximation of $\mathcal{H}$. Then automatic differentiation is used to calculate the partial derivatives with respect to $\mathbf{q}$ and $\mathbf{p}$. Finally ODEint numerically solves the resulting ODE to predict the states at the next time-step : $(\hat{\mathbf{q}}^{t_{k+1}}, \hat{\mathbf{p}}^{t_{k+1}})$.

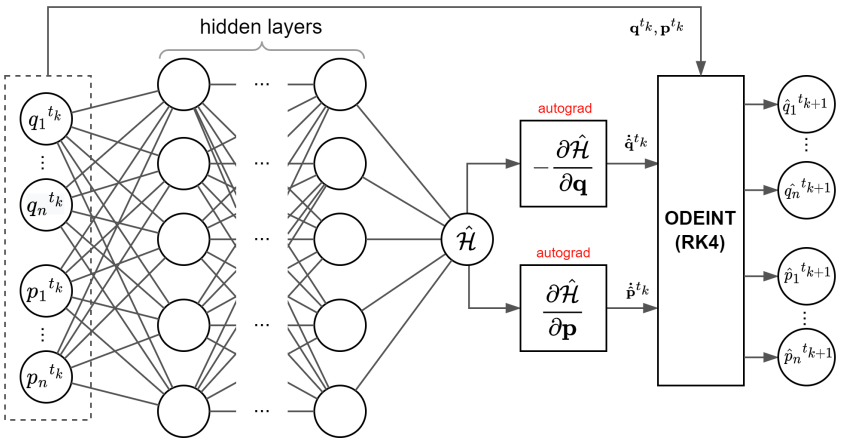


Figure 1: Simple-HNN architecture

The prediction of the state at the next time-step is performed sequentially until a trajectory $(\mathbf{q}, \mathbf{p})^{t_0, t_1, \ldots, t_K}$ is obtained as illustrated in Figure 2.

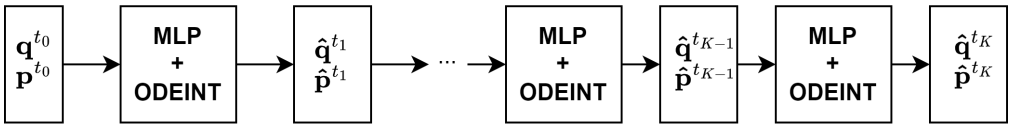


Figure 2: Simple-HNN architecture predicting a state trajectory $(\mathbf{q}, \mathbf{p})^{t_0, t_1, \ldots, t_K}$

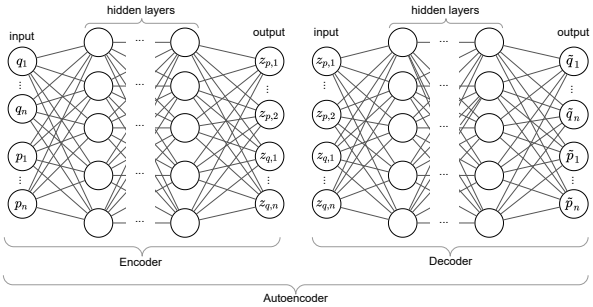
The predicted trajectory is then evaluated using a loss function similar to the one in the SymODEN architecture:

$$L = \frac{1}{NK} \sum_{k=1}^{T} \left\| \mathbf{w}_1^T \left(\hat{\mathbf{p}}^{t_k} - \mathbf{p}^{t_k} \right) \right\|_2^2 + \left\| \mathbf{w}_2^T \left(\hat{\mathbf{q}}^{t_k} - \mathbf{q}^{t_k} \right) \right\|_2^2 , \tag{7}$$

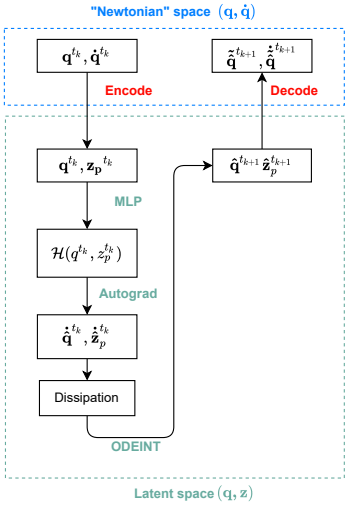
where N is the number of training trajectories, T the number of time-steps that are used during training, $\mathbf{w_1}$ and $\mathbf{w_2}$ are vectors that multiply each generalized coordinate and momentum so that their contribution in the loss function can be modulated. K is the number of time-steps in one trajectory, and N the number of trajectories in a batch. For simplicity, we train all of the models on a single batch of 100 trajectories. The loss function is then used to update the MLP's parameters, using backpropogation, that is implemented in the `torchdiffeq`[10] library that is based on `PyTorch`.

Note that the hat in $\hat{\mathcal{H}}$ indicates that it is an approximation.

3.2 Autoencoder-HNN



(a) Representation of the autoencoder



(b) One prediction step of the autoencoder model

Figure 3: Representation of the Autoencoder-HNN model and one prediction step using an initial state in the form $\mathbf{q}, \dot{\mathbf{q}}$

The Autoencoder-HNN architecture presented in [7] uses an autoencoder to transform the state representation $(\mathbf{q}, \dot{\mathbf{q}})$ into a latent-space $(\mathbf{q}, \mathbf{z_p})$. The autoencoder can also transform the coordinates in the latent space back to the $(\mathbf{q}, \dot{\mathbf{q}})$ representation, which will be addressed as "Newtonian space". This mapping is illustrated in Figure 3a. The idea behind this architecture is that training the MLP and autoencoder simultaneously could cause the encoder in the autoencoder to learn a non-linear mapping that transforms $\dot{\mathbf{q}}$ into $\mathbf{p}$, such that $\mathbf{z_p} \approx \mathbf{p}$. Since this is not enforced during training, $\mathbf{z_p}$ will not necessarily be close to $\mathbf{p}$.

The procedure used by the Autoencoder-HNN model to predict trajectories is summarized in Figure 3b. In practice, the initial state $(\mathbf{q}^{t_0}, \dot{\mathbf{q}}^{t_0})$ is encoded into $(\mathbf{q}^{t_0}, \mathbf{z_p}^{t_0})$. A model using the Simple-HNN architecture then performs predictions in the latent space $(\mathbf{q}, \mathbf{z_p})$, and outputs the trajectory $(\hat{\mathbf{q}}, \hat{\mathbf{z_p}})^{t_0, t_1, .., t_K}$. These trajectories are compared to the encoded nominal trajectories $(\mathbf{q}, \mathbf{z_p})^{t_0, t_1, .., t_K}$ in the HNN loss function :

$$\mathcal{L}_{HNN-\text{Newtonian space}} = \frac{1}{NK} \sum_{k=1}^{T} \left\| \mathbf{w_1} \left(\widehat{\mathbf{q}}^{t_{k+1}} - \mathbf{q}^{t_{k+1}} \right) \right\|_2^2 + \left\| \mathbf{w_2} \left(\hat{\mathbf{z_p}}^{t_{k+1}} - \mathbf{z_p}^{t_{k+1}} \right) \right\|_2^2 \tag{8}$$

Note that the hat $\hat{}$ represents that the quantities are estimated using an approximation of the Hamiltonian function.

The predicted trajectories $(\hat{\mathbf{q}}, \hat{\mathbf{z_p}})^{t_0, t_1, .., t_K}$ are then decoded into the "newtonian" state representation $(\tilde{\hat{\mathbf{q}}}, \tilde{\hat{\dot{\mathbf{q}}}})^{t_0, t_1, .., t_K}$. They are evaluated in the prediction loss function:

$$\mathcal{L}_{HNN-\text{Latent space}} = \frac{1}{NK} \sum_{k=1}^{T} \left\| \mathbf{w_1} \left(\widetilde{\mathbf{q}}^{t_{k+1}} - \mathbf{q}^{t_{k+1}} \right) \right\|_2^2 + \left\| \mathbf{w_2} \left(\widetilde{\mathbf{q}}^{t_{k+1}} - \dot{\mathbf{q}}^{t_{k+1}} \right) \right\|_2^2 \tag{9}$$

Note that the tilde $\tilde{}$ represents that the coordinate has been decoded from the latent space.

Finally, the nominal trajectory $(\mathbf{q}, \mathbf{z_p})^{t_0, t_1, .., t_K}$ is encoded and decoded and evaluated using the reconstruction loss :

$$\mathcal{L}_{Autoencoder} = \frac{1}{NK} \sum_{k=1}^{T} \left\| \mathbf{w_1} \left(\widetilde{\mathbf{q}}^{t_k} - \mathbf{q}^{t_k} \right) \right\|_2^2 + \left\| \mathbf{w_2} \left(\widetilde{\mathbf{q}}^{t_k} - \dot{\mathbf{q}}^{t_k} \right) \right\|_2^2 \tag{10}$$

The autoencoder architecture and it's loss functions are summarized in the Figure 4:

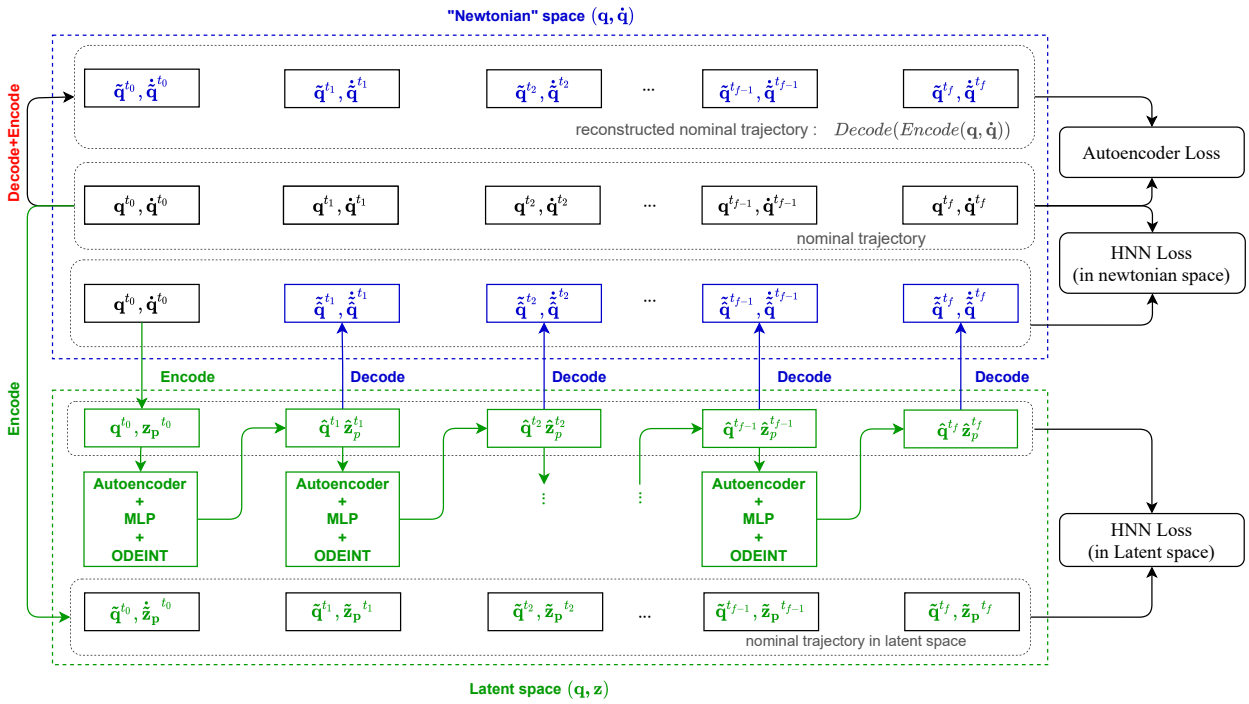


Figure 4: Autoencoder-HNN model and how losses are calculated

3.3 Input-HNN

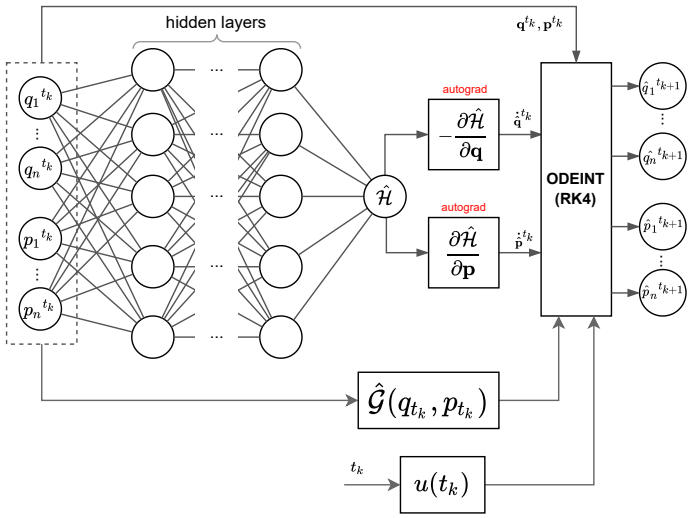


Figure 5: Input-HNN model

The Input-HNN architecture extends the simple-HNN architecture to include time dependent input in two ways.

The first method uses a neural network to approximate the input matrix $\mathcal{G}(\mathbf{q}, \mathbf{p})$. The trajectories used here are of the form $(\mathbf{q}, \mathbf{p}, \mathbf{u})^{t_0, t_1, .., t_K}$.

In the second method, $\mathcal{G}(\mathbf{q}, \mathbf{p})$ is known, meaning the training trajectories are in the form $(\mathbf{q}, \mathbf{p}, \mathcal{G}, \mathbf{u})^{t_0, t_1, .., t_K}$. For both methods, the loss function is the same as the one presented in Equation 7 for the Simple-HNN architecture.

Note that in the scope of this project, the output of $\mathcal{G}(\mathbf{q}, \mathbf{p})$ will be a constant matrix, of the form $\left[\mathbf{0}_n^T, \mathbf{1}_n^T\right]^T$ where $\mathbf{0}_n$ and $\mathbf{1}_n$ are vectors containing only zeros and ones in $\mathbb{R}^n$, respectively. Also note that this slightly differs from the work in [4] as they seek to learn a term $\mathbf{F}(t) = \mathcal{G}(\mathbf{q}, \mathbf{p})\mathbf{u}(t)$, by only keeping its time dependence, while we seek to either learn $\mathcal{G}(\mathbf{q}, \mathbf{p})$ or simply use this input matrix along with $\mathbf{u}(t)$to aid in training the MLP approximating $\mathcal{H}$.

3.4 Training challenges and strategies

3.4.1 Increasing horizon

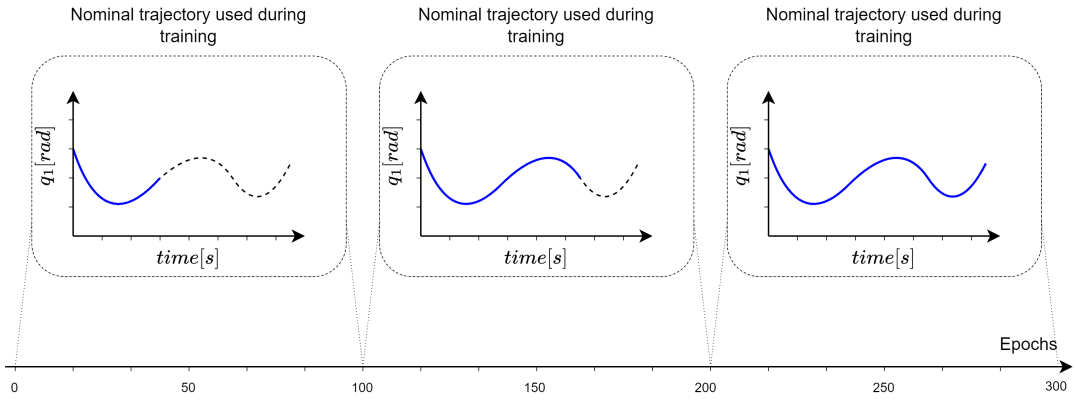


Figure 6: How the horizon is increased during training - at a training epoch, only the time-steps represented by the blue line are used as training samples

Training over a long horizon tends to result in a model that would not give any meaningful output. Most of the time this translates to a model that outputs a constant value, or a model that diverges from the nominal trajectories. Horizon schemes were used to make the training process easier. At first, the models are trained on trajectories of a certain number of time-steps. When the loss function decreases sufficiently and the predicted trajectories look close enough to the nominal trajectories, the training horizon is increased. This is done repeatedly until the horizon is equal to the maximum length of the nominal trajectories as illustrated in Figure 6.

3.4.2 Pre-processing and re-scaling

The initial outputs of the models are usually far from the nominal trajectories. This can be a problem in many ways as different generalized coordinates can have different magnitudes. During the first training epochs, the error terms in the loss function for the coordinates with small magnitudes can be neglected compared to coordinates with larger magnitudes. Multiple re-scaling schemes are proposed as a possible solution :

- **Rescaling using standard deviation :** Each generalized coordinate in a trajectory can be re-scaled using a manually chosen value that is close to the standard deviation of that trajectory. The standard deviations are calculated over all of the trajectories, for each coordinate separately. For example, in the case of the Furuta pendulum, we will determine 4 coefficients (w_1, w_2, w_3, w_4) and multiply (q_1, p_1, q_2, p_2) with those coefficients, where $w_1 \approx \frac{1}{\text{std}(q_1)}$.
- **Min-max scaling of trajectories :** Calculate the minimum and maximum of each trajectory for each coordinate, and re-scale the coordinate using the formula : $x_{rescaled} = \frac{x - min(x)}{abs(max(x) - min(x))}$. This method bounds all of the trajectories between zero and one.
- **Min-max scaling of the error term in the loss function:** Every time the horizon is increased, the minimum and maximum of every trajectory and every coordinate are calculated. Then the error term in the loss function is rescaled: $e_{rescaled} = \frac{e}{abs(max(x) - min(x))}$. Although this differs from the min-max formula usually found in literature, this variation intends to bring the error terms of different coordinates to the same magnitude.

  Note that both min-max scaling methods might not be rigorous enough due to the fact that all the trajectories are rescaled with their respective min-max values. This means the model is not learning real trajectories anymore.

3.4.3 Learning rate scheduling and gradient clipping

In some cases predicted trajectories seem close enough to the nominal trajectories but around the end of training the model starts diverging and predicts trajectories more and more poorly. As a remedy, two solutions were proposed : gradient clipping and learning rate scheduling. The idea behind these two methods is that if the model is already predicting well, and supposedly has good parameters, we do not want it to diverge from those parameters too quickly to make it robust against a possible local minimum. We apply gradient clipping by clipping the total norm of the gradients if it reaches a value higher than 1.0 [11]. Learning rate scheduling is used to keep the learning rate constant for a number of epochs and then linearly decreasing it until it reaches a certain value.

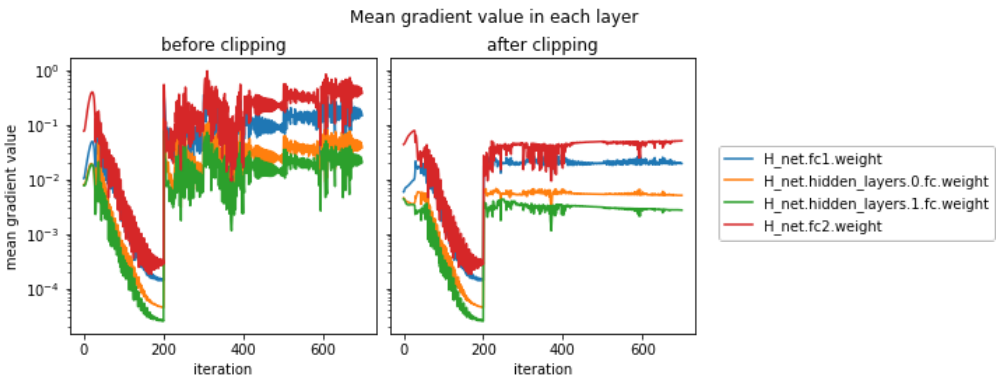


Figure 7: Mean gradient value in different layers of an MLP that approximates $\mathcal{H}$

As you can see in Figure 7 after increasing the horizon at iteration 200, gradient clipping

has a visible effect. As will be discussed in Section 4.1.1 and 4.2.1, gradient clipping and learning rate scheduling did not always bring about improvements. This is coherent with a hypothetical case where the gradients were "good" gradients, clipping them will slow down training. The same applies to learning rate scheduling.

3.4.4 Training time

Because of the sequential nature of the model, training can have a long duration. To gain additional performance, a GPU was used for part of the experiments but it did not consistently improve training times, notably when models that had 17 residual blocks were used. As a result most of the time a CPU was used to train the models presented in this report.

3.4.5 Activation function

The activation function $f(x) = x + \sin^2(x)$ that was proposed in [7] is used for any MLP that approximates Hamiltonian functions in this work. As was mentioned in [7], this activation function helps the model learn the periodic nature of the dynamical systems we are studying, and help the model give better predictions at horizons longer than the training horizon. For some models such as the autoencoder, the hyperbolic tangent activation function is used.

3.4.6 Multi-level strategies

The methods presented previously had results that could be improved, therefore, new multi-level strategies are presented. These strategies are inspired by the work of [9]. This idea comes from the intuition that larger models are harder to train, but can perform better than smaller (and easier to train) models. All of these methods introduce new parameters to the model during training, and initialise them in a specific way. The next 4 models replace the MLP that approximates the Hamiltonian function by a residual neural network to accomplish the same task.

3.4.6.1 Expanding-HNN

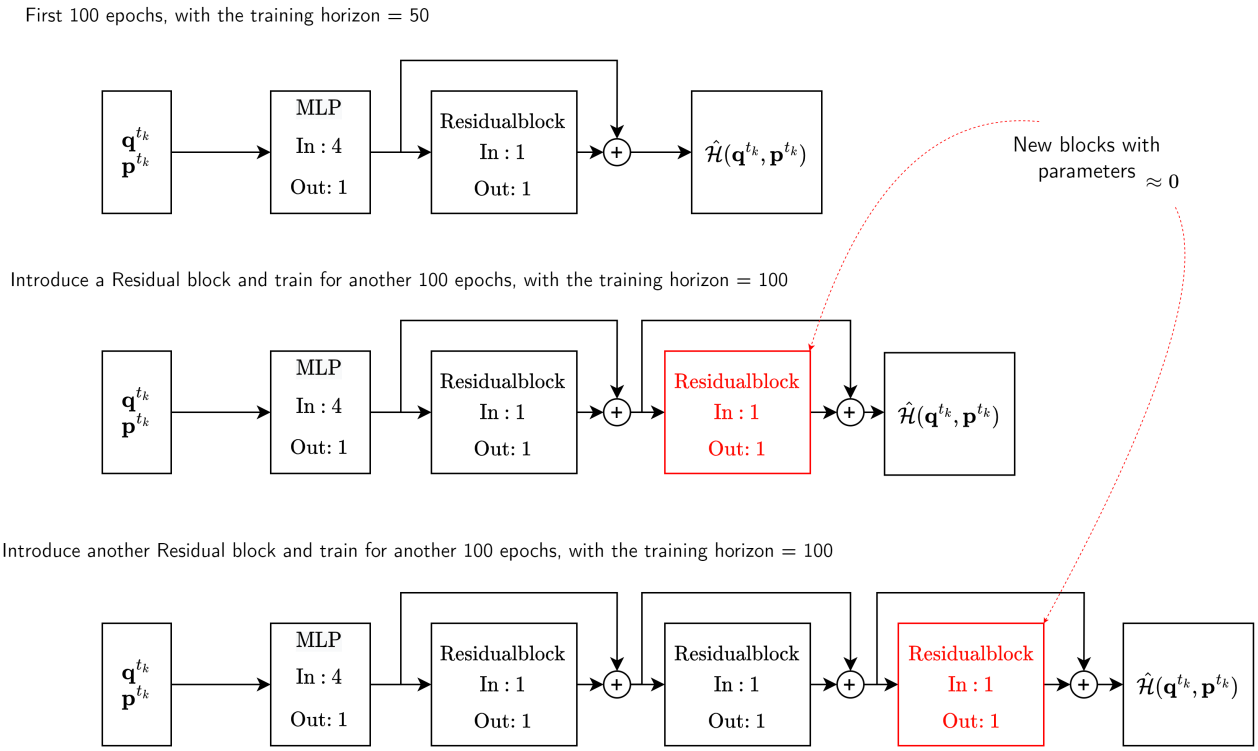


Figure 8: Expanding-HNN architecture

To explain this model, we take the example of the Furuta pendulum, that has 4 states (q_1, q_2, p_1, p_2). Initially, we have a model that utilizes both an MLP and Residual block to approximate $\mathcal{H}$. Note that the MLP takes the 4 states as input and outputs a scalar, while the Residual block outputs a scalar and takes as input a scalar. The output of a residual block is expressed as $y = \alpha f(x) + x$ where α is equal to 1, and $f(x)$ is a neural network.

Then, every 100 epochs, when the training horizon is increased, an additional residual block is introduced after the last residual block of the model. The new residual block is illustrated in red in Figure 8. Additionally, the new residual block has its parameters initialized randomly to very small values, such that initially it will mostly serve as a pass-through for the previous model's output.

3.4.6.2 Expanding-wide-HNN

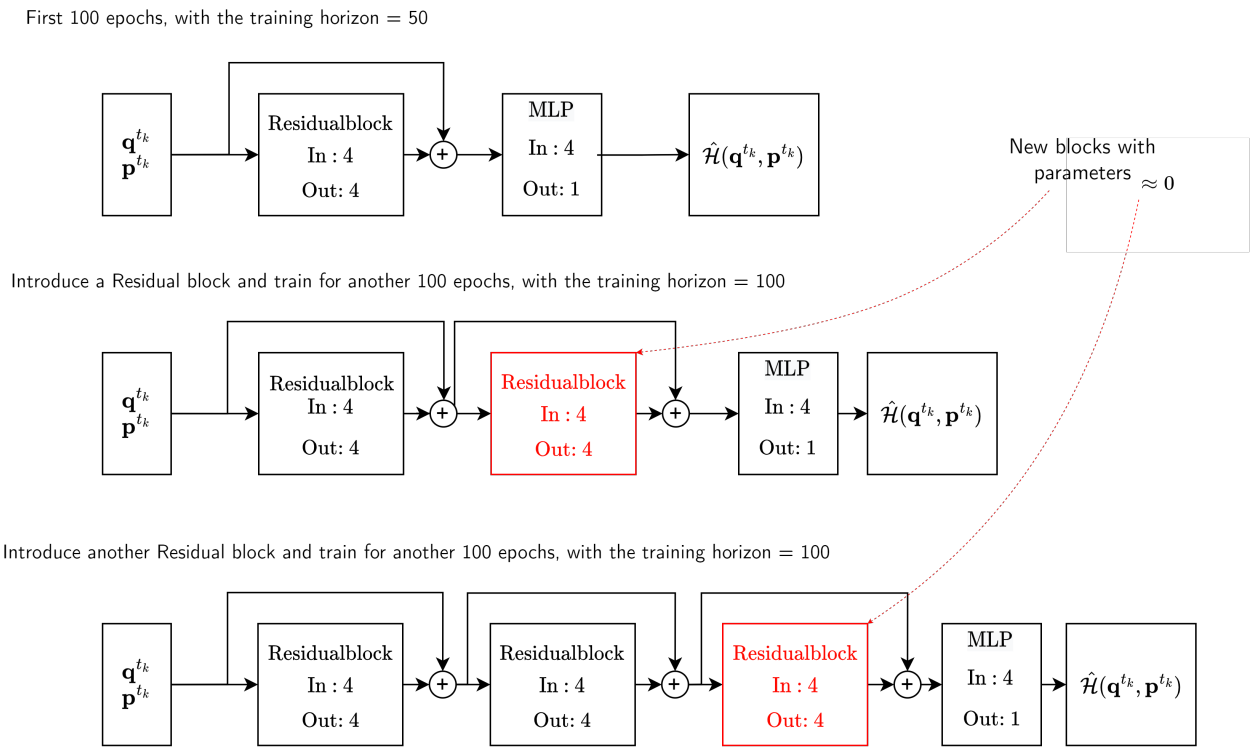


Figure 9: Expanding-wide-HNN architecture, red represent new parameters

This model is very similar to the Expanding-HNN model, but has different dimensionality. The MLP is placed just before the output of the $\mathcal{H}$ function, and the residual blocks have input and output vectors of the same dimension as $(\mathbf{q}, \mathbf{p})^T$. Here new residual blocks are introduced between the last residual block and the MLP as illustrated in red in Figure 9

3.4.6.3 Interp-HNN

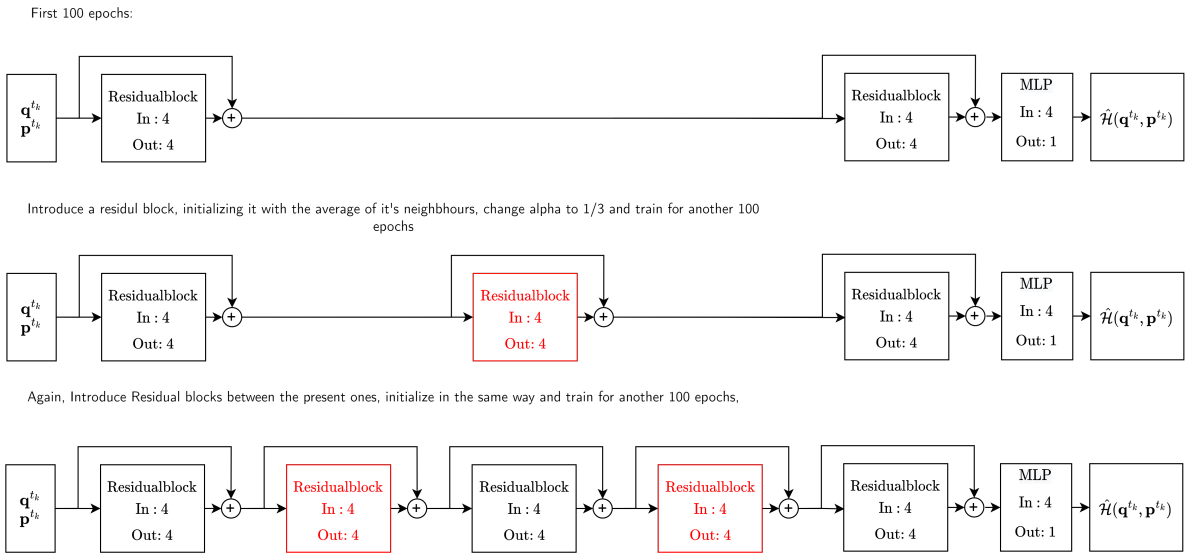


Figure 10: Interp-HNN architecture, red represent new parameters

Contrary to the Expanding-HNN model, Interp-HNN keeps a constant training horizon during training. After a certain number of epochs, a new residual block is introduced between every pair of residual blocks in the previous model. This is illustrated in red in Figure 10. Every new residual block is initialised using the average of the parameters of its two neighboring residual blocks. Also, when the number of residual blocks is increased, the residual block parameter α in the formula $y = \alpha f(x) + x$ is changed to $\frac{1}{n_{residual\_blocks}}$ where $n_{residual\_blocks}$ is the number of residual blocks after the increase. Even though the number of parameters of the model was increased, the parameter initialization method and the change of α are intended to make the model's output similar to what it was before. This however, is not necessarily the case due to the non-linear nature of the activation functions. The number of residual blocks this model will have is, in order, $2, 3, 5, 9, 17$, if the model size is increased 4 times.

3.4.6.4 Interp-horizon-HNN

This model is very similar to the Interp-HNN model, but uses the increasing horizon method. It was introduced because Interp-HNN failed to learn a meaningful Hamiltonian with Furuta pendulum trajectories. Here, the new residual blocks are introduced when the training horizon is increased, in a similar fashion to the Expanding-HNN model.

3.5 Trajectory generation

To generate the training trajectories, the analytical expressions of the Hamiltonian functions for the simple pendulum and Furuta pendulum from [7] are inserted into equation 3 and then

the equation is solved for a number of time-steps using the numerical solver ODEint from the `torchdiffeq` library [10] [5].

4 Experiments, results and discussion

In this section, the Simple-HNN, and Input-HNN architectures are tested on trajectories generated by the simple pendulum and the Furuta Pendulum. The Autoencoder-HNN architecture is evaluated only on the Furuta pendulum.

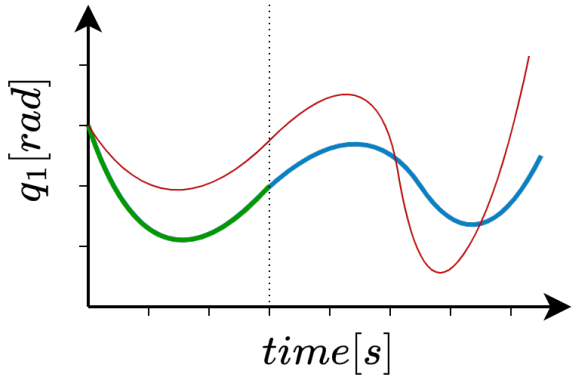


Figure 11: Color code used in the plots of Section 4 : Blue represents nominal trajectories, green represents the part of the nominal trajectory used in training, and red represent the trajectory predicted by one of our models

The Figures in this section follow the same color code that is illustrated in Figure 11. The blue and green trajectory is a nominal trajectory where the green portion is used as training data and the blue portion is unseen by model and is only used to evaluate its performance. The red trajectory is predicted by the model being evaluated, using the same initial state as the nominal trajectory.

The nominal energy trajectories shown in the figures (for example Figure 13 and 14) are calculated using the true energy function for the studied dynamical system : $\mathbf{E}(\mathbf{q}, \mathbf{p})$. The predicted energy is calculated using the true energy function of the system, with the states predicted by the model that is being evaluated: $\mathbf{E}(\hat{\mathbf{q}}, \hat{\mathbf{p}})$. Additionally, when any rescaling is used on training trajectories, the inverse rescaling is used for both the predicted and nominal coordinates $(\mathbf{q}, \mathbf{p})$ before evaluating the energy function.

4.1 Simple pendulum

4.1.1 Overview

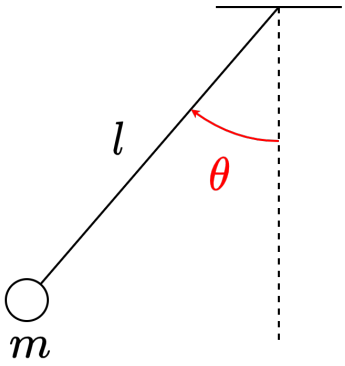


Figure 12: Simple pendulum diagram

The simple pendulum is a dynamical system that can be used as a baseline to test the performance of the different architectures. The simple nature of the model causes it to perform relatively well in the majority of experiments. It is described by its mass m, arm length l and angle θ. The generalized coordinate of the simple pendulum is θ, as illustrated in Figure 12. The generalized momentum is equal to $P_\theta = ml^2\dot{\theta}$ [7].

The simple pendulum experiments are performed using 100 trajectories with initial points uniformly sampled in the two intervals $[-2, -0.3]$ and $[0.3, 2]$. The intervals are chosen such that trajectories that oscillate very close to zero and have very small amplitudes are not used.

The AdamW [12] optimizer is used with a learning rate of 10^{-3} and weight decay of 10^{-4} (L_2 regularization coefficient). The number of parameters for the experiments varies between $7.5k$ and $7.7k$ parameters, allowing the models to be compared.

The training horizon starts at 20 time-steps and is increased by 5 timesteps every 100 epochs until it reaches 40 time-steps. For two experiments with the Interp-HNN and Simple-HNN architectures, the training horizon starts and is kept at 40 time-steps for 500 epochs.

The intergration method used in the numerical solver ODEint is Rung-Kutta 4 with an integration step $T_s = 0.05[s]$.

Initially, the loss weights $\mathbf{w_1}$ and $\mathbf{w_2}$ as shown in equation (7) were set to different values. In later experiments, keeping them equal to 1.0 appeared to be enough to obtain satisfactory results for the simple pendulum. Gradient clipping, learning rate scheduling and trajectory rescaling are not used when performing the experiments on the simple pendulum.

All of the models incorporating an input signal use a chirp signal that is generated using the following formula :

$$u(t) = \sin\left[2\pi\left(\frac{f_1 - f_0}{2T}t^2 + f_0 t\right)\right] C_{scale}\ , \tag{11}$$

where $T = 2.0[s]$, $f_0 = 0[Hz]$, $f_1 = 1[Hz]$ and $C_{scale} = 1.0$.

For the experiments in this section, the dissipation matrix D was set to zero.

In Table 1 we summarize the parameters used in the experiments for the Furuta pendulum task.

Parameter	Value
Learning rate	0.001
Weight decay	10^{-4}
Integration method	Rung-Kutta 4
Integration time-step	0.05
Batch size	$n_{trajectories} = 100$
Training epochs	500
Activation function	$x + \sin^2(x)$
Loss weight vector	$\mathbf{w_1} = 1.0, \mathbf{w_2} = 1.0$
m	1[kg]
l	1[m]

Table 1: Simple pendulum experiment parameters

4.1.2 Result comparisons

The multi-level strategy models (Expanding-HNN and Interp-HNN) are designed such that at the end of training, their maximum number of parameters is approximately equal to $7.7k$ parameters.

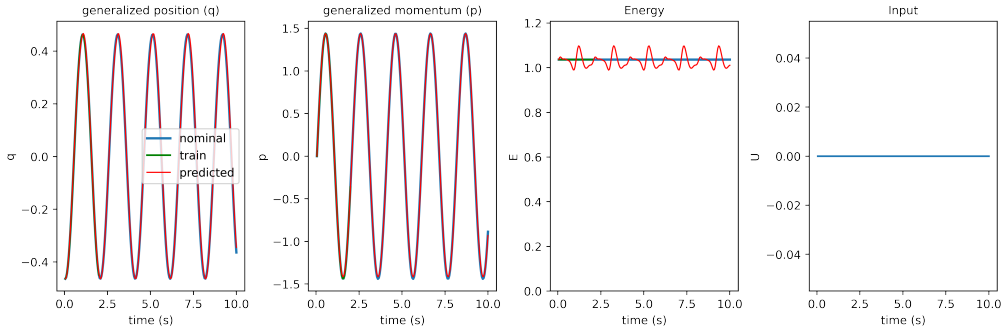
As a baseline, training the Simple-HNN model on a constant horizon of 40 time-steps led to predicted trajectories that drastically diverged from the nominal trajectories. However, training the Simple-HNN model with the horizon scheme described in Section 4.1.1 lead to the results observed in Figure 13. The predicted trajectories are close to the nominal trajectories even for a horizon exceeding the training horizon.

The Input-HNN model (see Figure 14) where the input $\mathcal{G}$ is known outperforms the Input-HNN model, where $\mathcal{G}$ is approximated by a neural network (see Figure 15). This is expected since it can be argued that adding more parameters can make training take longer, and here the same number of epochs was used in both cases. Furthermore, the Input-HNN model where $\mathcal{G}$ is known has a comparable performance to the Simple-HNN model.

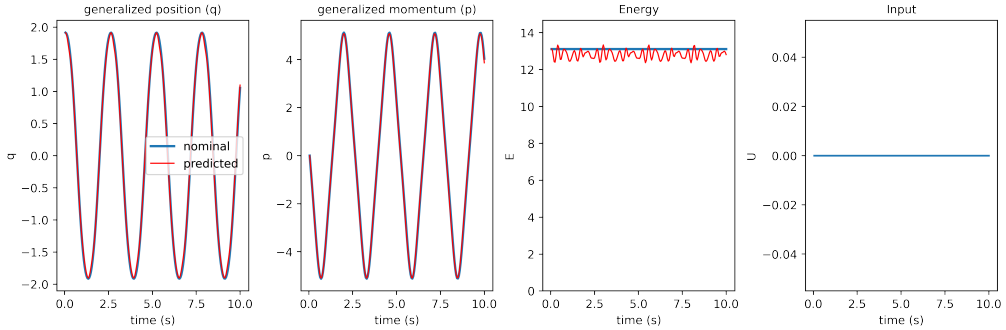
The Expanding-wide-HNN model is outperformed by the 3 previous models. When subjected to a 0.5 Hz input signal, its predictions are close to the nominal trajectories. When a 1 Hz sine input signal is used, as seen in Figure 16c, the nominal trajectory can sometimes have a special shape that most likely was not present in the training set. Therefore, it does not perform well on that type of trajectory. The results of the Expanding-HNN model and the Expanding-wide-HNN model have an almost identical performance and thus only one of the two models was presented in this section.

Unfortunately, for the same number of parameters, the Interp-HNN model was only able

to generate trajectories that strongly diverged from the nominal trajectories. The Interp-horizon-HNN model was able to predict trajectories close to the nominal trajectories but still showed some drift at longer horizons, as seen in the Figure 17.



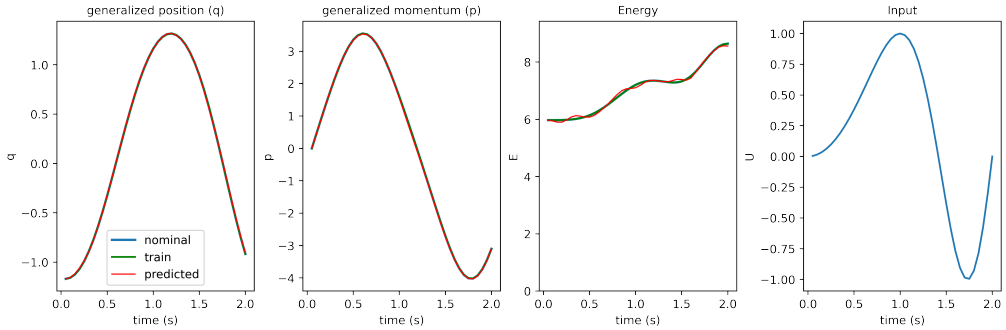
(a) Train trajectories



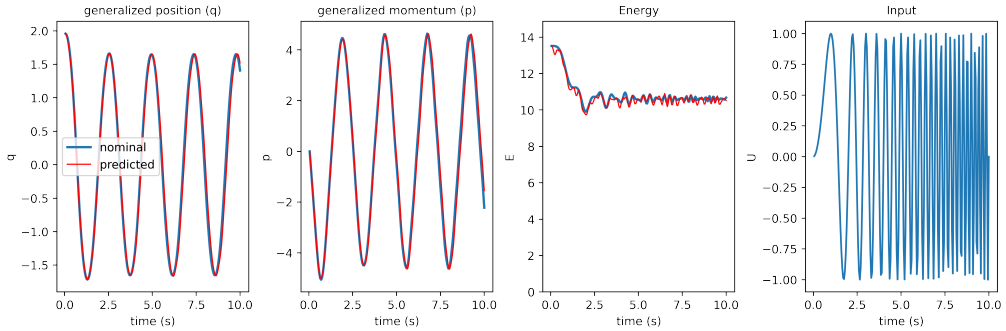
(b) Test trajectories

Figure 13: Simple-HNN experiments with the simple pendulum

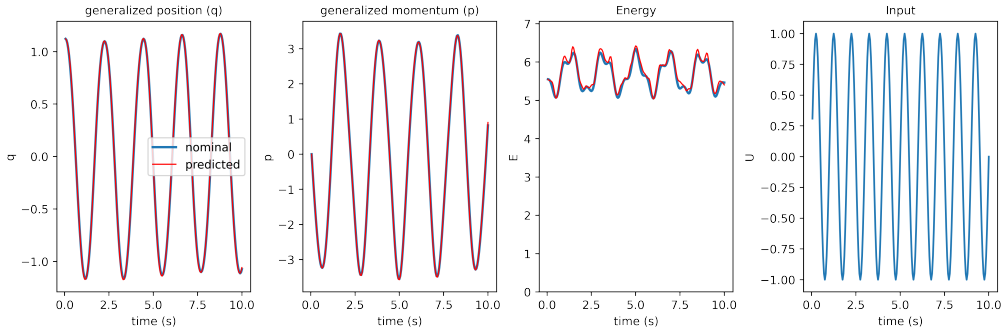
In Figure 13 we can see the predictions of the Simple-HNN architecture on training and testing trajectories. In this case there is no input and the maximum training horizon is 40 time-steps. The training horizons are progressively increased during training as described in Section 4.1.1. The predicted trajectories are very close to the nominal trajectories, even for a horizon longer than the training horizon.



(a) Train trajectories



(b) Test trajectories



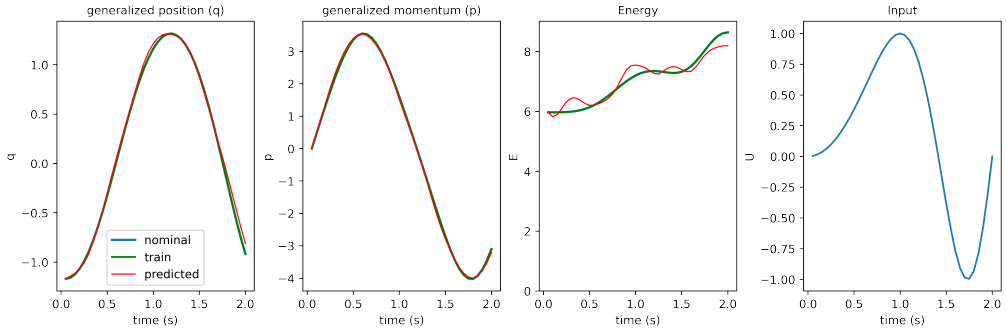
(c) Test trajectories with a 1Hz sine input signal

Figure 14: Input-HNN model experiments with the simple pendulum when $\mathcal{G}$ is known

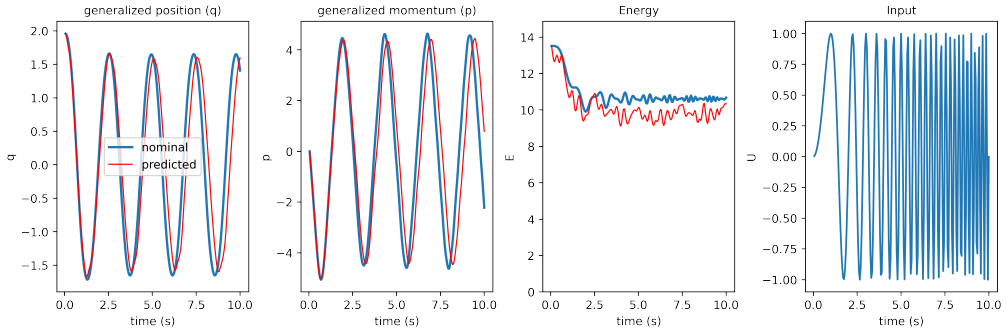
In Figure 14 we can see the results for the Input-HNN architecture, when it is trained on a chirp input as described in Section 4.1.1. Also to train this model, training samples of the form $(\mathbf{q}, \mathbf{p}, \mathcal{G}, \mathbf{u})^{t_0, t_1, ..., t_K}$ were used, so $\mathcal{G}$ is known in this case.

The model's predictions are close to to the nominal trajectories both in training and testing, even when presented with an unseen input such as the higher frequency (higher than 1 Hz) portion of the chirp input in Figure 14b, or a 1 Hz sinusoidal signal.

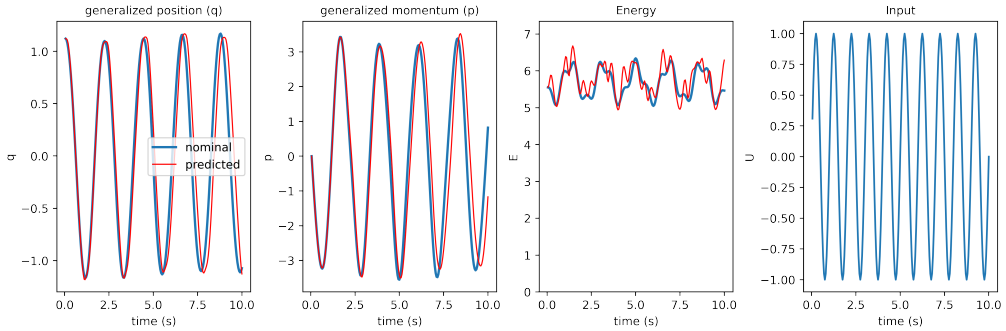
Note that the input $\mathbf{u}(\mathbf{t})$ is represented on the far right of every plot.



(a) Train trajectories



(b) Test trajectories

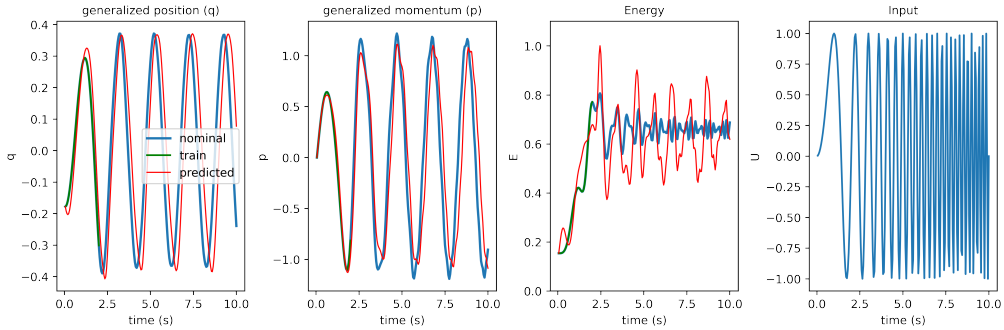


(c) Test trajectories with a 1Hz sine input signal

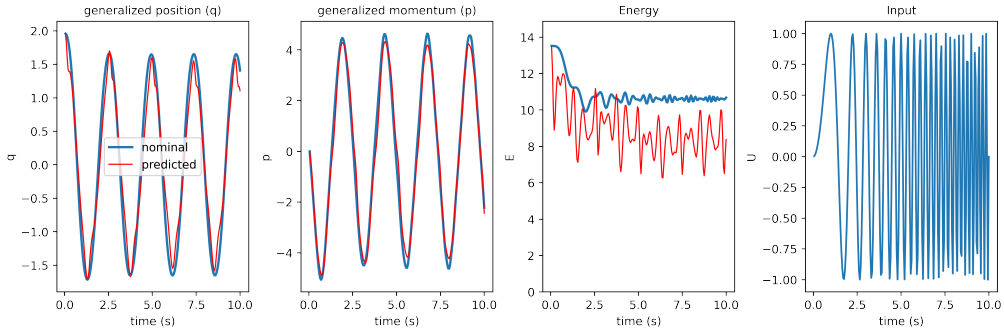
Figure 15: Input-HNN model experiments with the simple pendulum when $\mathcal{G}$ is approximated by an MLP

In Figure 15 we can see the results for the Input-HNN architecture, when it is trained with a chirp input as described in Section 4.1.1, and where both the Hamiltonian function and $\mathcal{G}(\mathbf{q}, \mathbf{p})$ are approximated by a neural network. The training samples used here are in the form $(\mathbf{q}, \mathbf{p}, \mathbf{u})^{t_0, t_1, .., t_K}$.

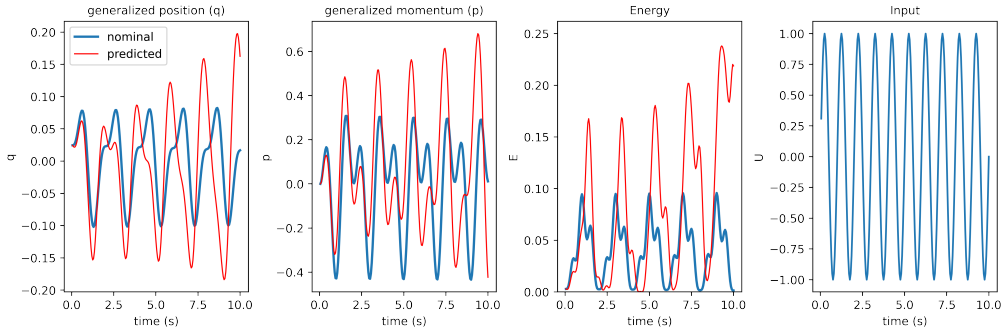
Although the training trajectories are very close to the predicted trajectories as we can observe in Figure 15a the predicted trajectories drift from the nominal trajectories when the prediction horizon exceeds the training horizon as can observed in Figure 15b. The same kind of drift is observed when the model is subjected to test trajectories that were generated using an input of 1 Hz in Figure 15c.



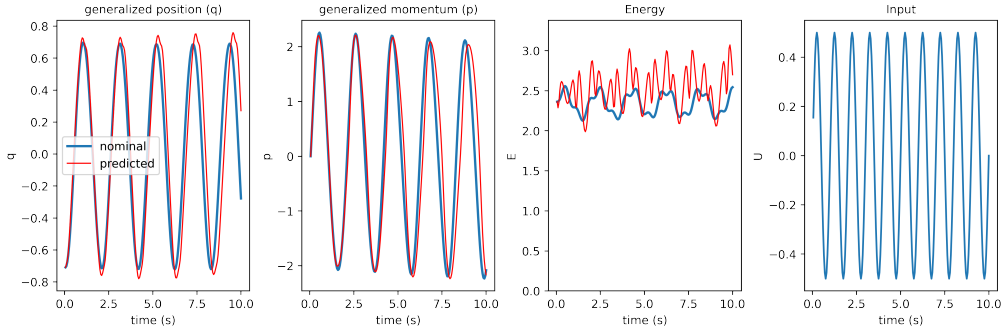
(a) Train trajectories



(b) Test trajectories



(c) Test trajectories with a 1Hz input signal



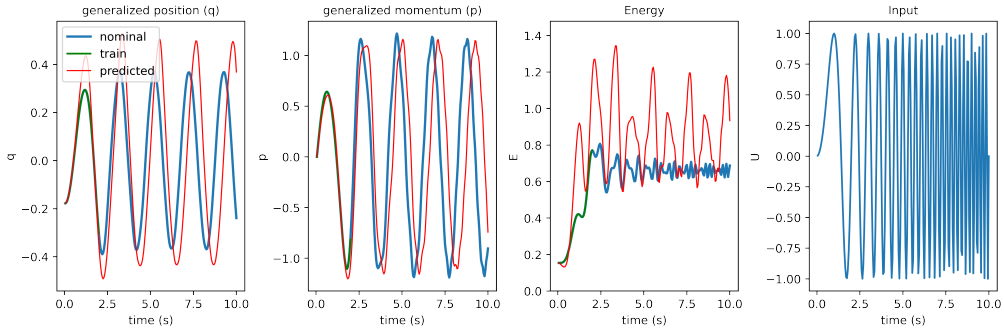
(d) Test trajectories with a 0.5Hz input signal

Figure 16: Expanding-wide-HNN model experiments with the simple pendulum when $\mathcal{G}$ is known

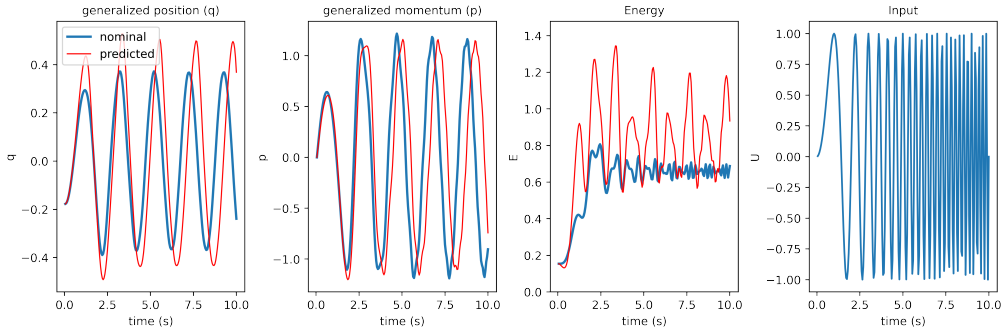
In Figure 16 we can see the results for the Expanding-wide-HNN architecture, when it is

trained with a chirp input. The training samples used here and in the next result are in the form $(\mathbf{q}, \mathbf{p}, \mathcal{G}, \mathbf{u})^{t_0, t_1, .., t_K}$.

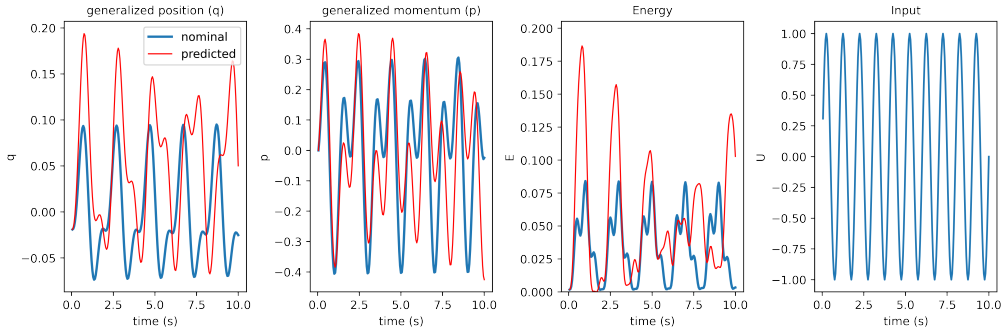
The predicted trajectories in this case present a small drift even in the training trajectory. When a 1 Hz input signal is used the predicted trajectories sometimes diverge as it can be observed in Figure 16c. Note that in this particular case, the simple pendulum produces a trajectory with a special shape that might be unseen in the training set, which could explain why the predictions diverge from the nominal trajectory.



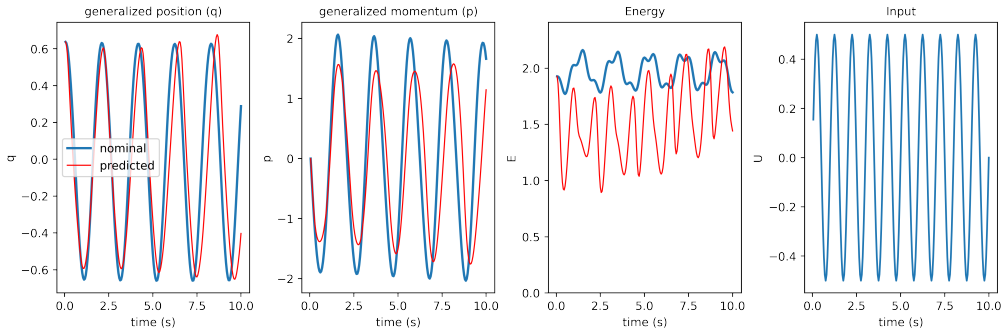
(a) Train trajectories



(b) Test trajectories



(c) Test trajectories with a 1Hz sine input signal



(d) Test trajectories with a 0.5Hz sine input signal

Figure 17: Interp-HNN model experiments with the simple pendulum when $\mathcal{G}$ is known

In Figure 17 we can see the results for the Expanding-wide-HNN architecture when it is trained with a chirp input. Notice that the predicted trajectories here are the not as close

to the nominal trajectories as with the previous models. Overall this model presents a lower performance compared to the previous ones.

4.2 Furuta pendulum

4.2.1 Overview

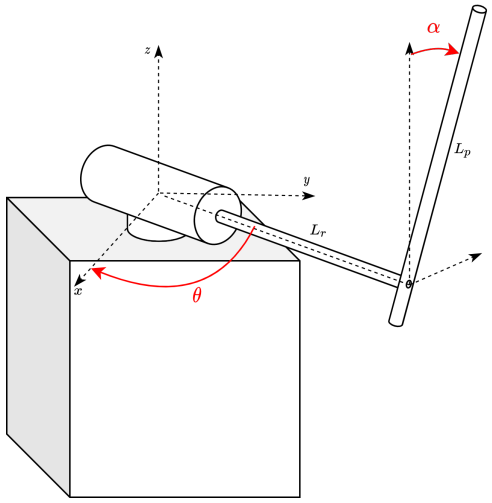


Figure 18: Furuta pendulum diagram

The Furuta pendulum is a chaotic dynamical system, and identifying its highly non-linear dynamics is a non-trivial task[13]. The parameters used during training are summarised in Table 2. The generalized coordinates for the Furuta pendulum are α and θ and are illustrated in Figure 18. At the lowest equilibrium position we have : $\alpha = \pi$. The Furuta pendulum model parameters used to generate the Furuta pendulum trajectories are the same as described in [7]. These parameters are similar to a real Furuta pendulum's parameters from Quanser, the Qube servo 2.

A different horizon scheme was used for the Furuta pendulum. Therefore, more time-steps were needed to have at least one "trajectory period" in the training trajectory. For the first 200 epochs, a horizon of 50 time-steps is maintained. The horizon is then increased by 50 time-steps every 100 epochs until a horizon of 300 is reached (meaning the horizon takes the following values (50,100,150,300), we can call this a horizon scheme). To improve training, different methods for increasing the horizon, such as one that follows the scheme (20,40,60,...,300), were tested. Since the results did not consistently improve with other schemes, the initial scheme was kept as it empirically yielded a satisfactory performance.

Training the proposed architectures was challenging and required experimenting with gradient clipping, learning rate scheduling, different horizon schemes, and rescaling the training trajectories. Results for the models that had satisfactory results will be presented, along with how the model was trained. Additionally, learning rate scheduling was not used since it did not appear to bring about more improvements than rescaling the trajectories did.

All of the models that incorporate an input signal use a chirp signal that is generated by the formula in equation 11 with $T = 1.5[s]$, $f_0 = 0[Hz]$, $f_1 = 2[Hz]$ and $C_{scale} = 0.0001$. All the models presented in the results in the following sections have between $33k$ and $33.7k$ parameters.

Parameter	Value
Learning rate	0.001
Weight decay	10^{-4}
Integration method	Rung-Kutta 4
Integration time-step	0.005
Batch size	$n_{trajectories} = 100$
Training epochs	700
Activation function	$x + \sin^2(x)$
Loss weight vector	$\mathbf{w_1} = [1.0, 1.0], \mathbf{w_2} = [1.0, 1.0]$

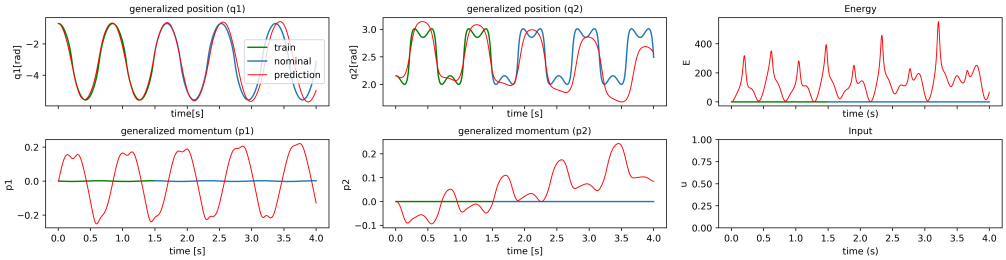
Table 2: Furuta pendulum experiment parameters

4.2.2 Result comparisons

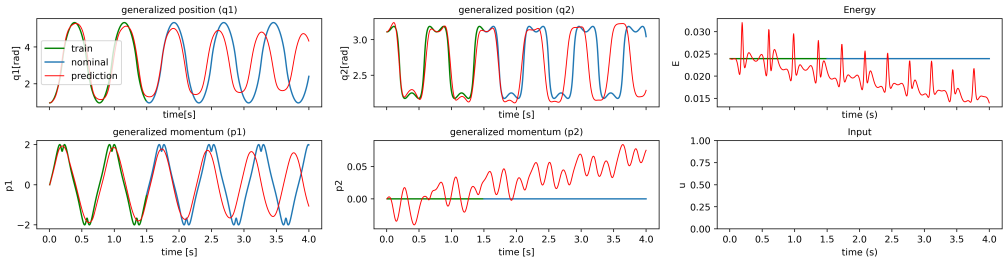
For the experiments in this section, the dissipation matrix D was set to zero. The autoencoder-HNN model uses the weight vectors $\mathbf{w_1} = [1, 10]$ $\mathbf{w_2} = [5, 15]$. These coefficients were chosen for two reasons. The first one was increasing the weight of the states q_2, p_2 compared to q_1, p_1 in the loss function improved the results. The second is to increase the weight of the states p_1 and p_2 because their trajectories had much smaller magnitudes than the other coordinates, which also improved the results. For the other models, $\mathbf{w_1}$ and $\mathbf{w_2}$ are set to 1 since they use other rescaling methods.

The experiments were structured in the following way. First we train the Simple-HNN model using different combinations of number of parameters, methods to rescale the training trajectories, and horizon schedules. For brevity we only report two examples in Figure 19 to show which combination of rescaling and parameters worked best. Using $33k$ parameters for the MLP approximating $\mathcal{H}$, standard deviation rescaling of the trajectories and min-max scaling in the loss function (as presented in Section 3.4.2) was the best combination. It is therefore used when training the Input-HNN, Expanding-HNN and Interp-HNN architectures and their variants. Then Input-HNN is trained in two ways, one where $\mathcal{G}$ is approximated by a neural network (and trained at the same time), and one where the input matrix $\mathcal{G}$ is known. Since the results showed that training with a known input matrix $\mathcal{G}$ is better, the multi-level strategy architectures, Expanding-HNN and Interp-HNN, are trained that way.

4.2.2.1 Simple-HNN



(a) With no rescaling



(b) With standard deviant rescaling of the training trajectories and loss function rescaling during training

Figure 19: Simple-HNN : Predictions of the model for Furuta pendulum test trajectories when the model has been trained using different rescaling strategies

Two experiments are presented for the Simple-HNN architecture. In Figure 19a we attempted to train the model without gradient clipping or any of the rescaling methods, and set the weight vectors $\mathbf{w_1}$ and $\mathbf{w_2}$ to one. The results show that the Simple-HNN architecture does not learn the correct representation of p_1 and p_2. This is likely because the magnitudes of p_1 and p_2 are very small compared to the magnitudes q_1 and q_2. When the training trajectories are rescaled using the standard deviation rescaling discussed in Section 3.4.2, the Simple-HNN model predicts the generalized momenta more accurately, as can be seen in Figure 19b. Since there is no input and no dissipation in this experiment, the coordinate p_2 in the nominal trajectory is equal to zero.

In the top right sub-figure in Figure 19a we can see that the predicted energy diverges from the nominal energy. It also presents periodic peaks. At this stage of understanding, we believe that this is due to the approximate discretization (Runge-Kutta 4) used in all of the architectures.

4.2.2.2 Autoencoder-HNN

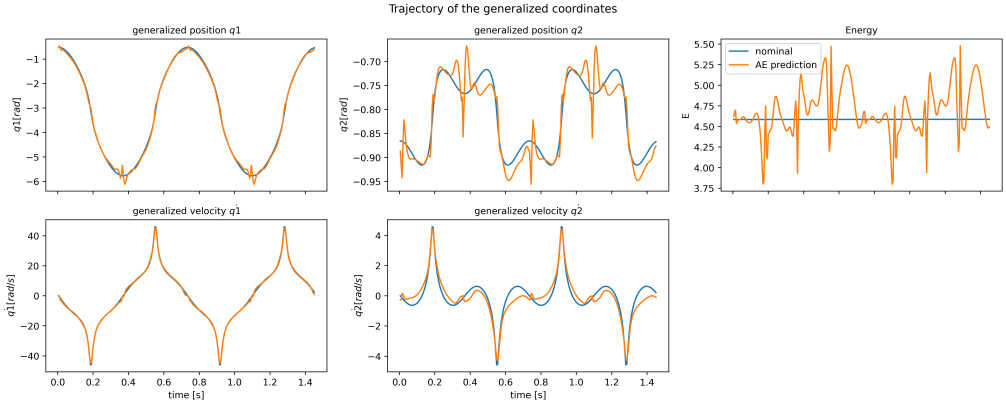


Figure 20: Autoencoder model : Trajectories that have been Encoded then Decoded are in orange. The nominal trajectories are in blue

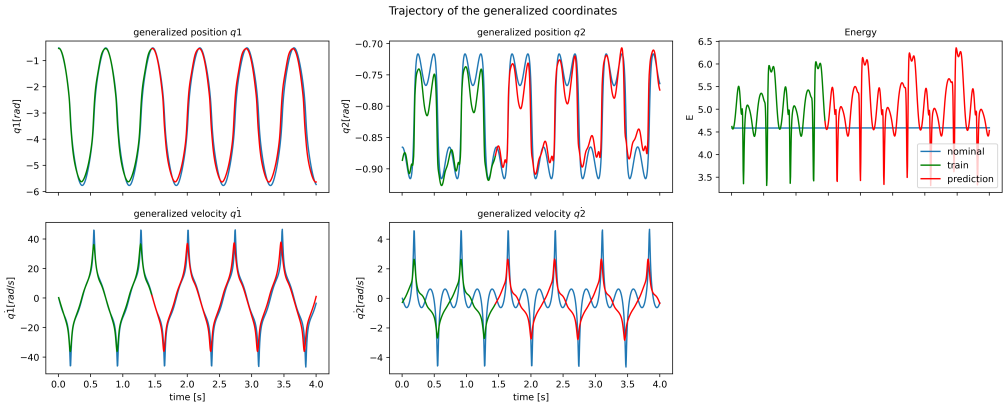
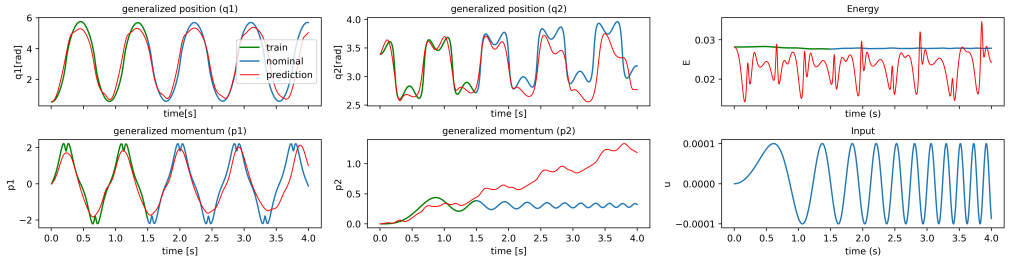


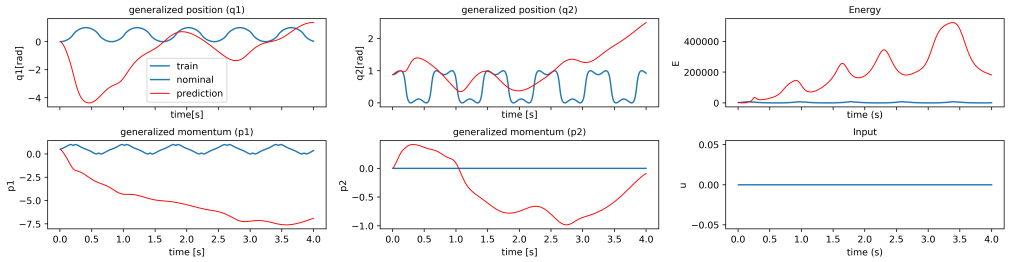
Figure 21: Autoencoder model : predictions of the model on the test set

Figures 21 and 20 show that the Autencoder-HNN model's predicted trajectories are close to the nominal trajectories. Figure 20 shows that the autoencoder reconstruction is accurate for q_1 and p_1 but less accurate for (q_2, p_2). The predicted generalized coordinates and momenta show the same inaccuracies in Figure 20. Additionally, the special trajectory shape (S shape) in the trajectory of p_2 is not captured by the model.

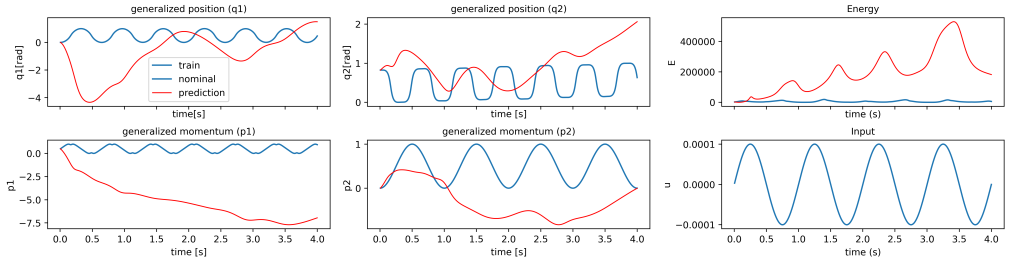
4.2.2.3 Input-HNN



(a) Chirp input



(b) No input



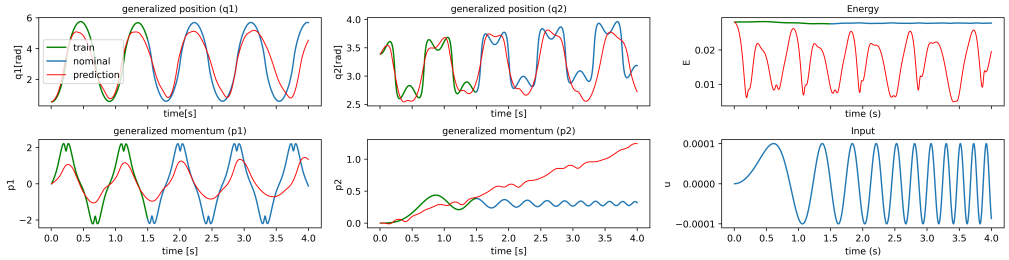
(c) 1 Hz sine input

Figure 22: Input-HNN : Predictions of the model for test trajectories on different types of inputs with the Furuta pendulum

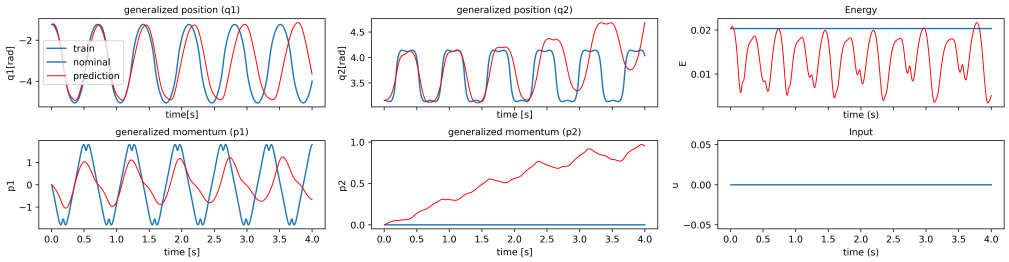
The Input-HNN model where $\mathcal{G}$ approximated by a neural network is outperformed by the version where $\mathcal{G}$ is known, so only the results of the Input-HNN model that has been trained with trajectories of the from $(\mathbf{q}, \mathbf{p}, \mathcal{G}, \mathbf{u})^{t_0, t_1, .., t_K}$ are presented. Standard deviation rescaling of the training trajectories and rescaling of the loss function error term provided the best results with the Simple-HNN model. As a result, both methods were used to train the Input-HNN model.

In Figures 22a, 22b and 22c, the predictions of the Input-HNN model on respectively, a chirp input, no input and a 1 Hz sine input are presented. The model's predicted trajectories for both train and test trajectories are close to the nominal trajectories but the predictions drift away from the nominal trajectory when the prediction horizon increases as can be seen in Figure 22a. However, with no input signal or a sine input signal of 1 Hz the Input-HNN model's predicted trajectories diverge completely from the nominal trajectories, as is illustrated in Figure 22c and 22b.

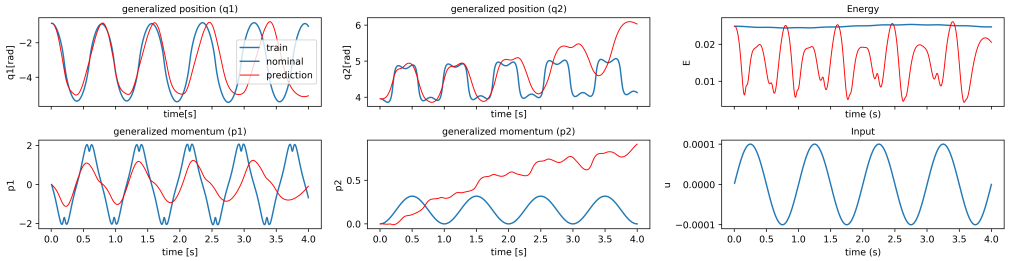
4.2.2.4 Multi-level strategies



(a) Chirp input

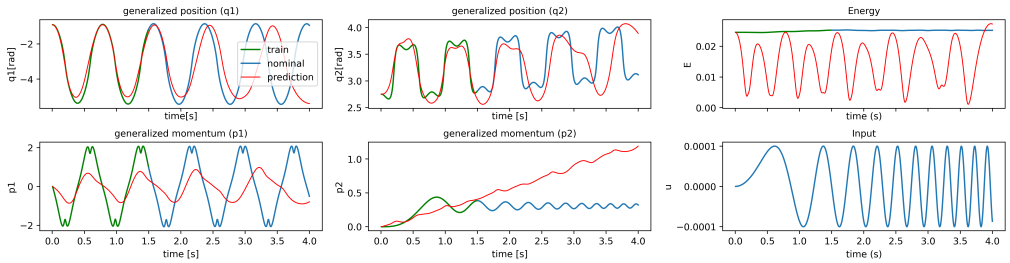


(b) No input

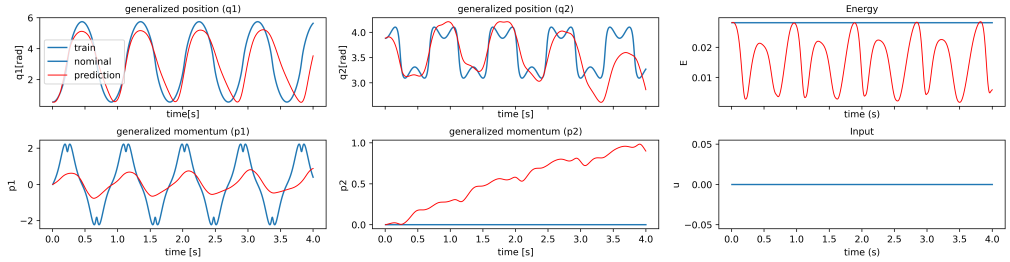


(c) 1 Hz sine input

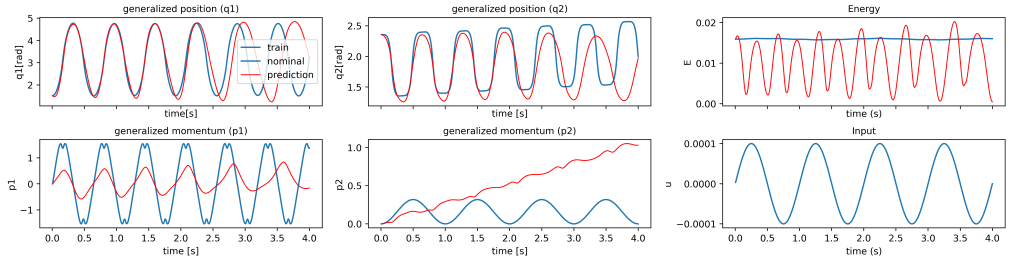
Figure 23: Expanding-wide-HNN : Predictions of the model for test trajectories on different types of inputs with the Furuta pendulums



(a) Chirp input



(b) No input



(c) 1 Hz sine input

Figure 24: Interp-horizon-HNN : Predictions of the model for test trajectories on different types of inputs with the Furuta pendulum

To improve the results of the Input-HNN model, multi-level training strategies were tested on the Furuta pendulum. Similarly to the Input-HNN model, the training trajectories are rescaled using their standard deviation and the loss function error term is min-max rescaled during training as was discussed in Section 3.4.2 during training for the multi-level strategies. The models are also trained with trajectories of the from $(\mathbf{q}, \mathbf{p}, \mathcal{G}, \mathbf{u})^{t_0, t_1, .., t_K}$, and evaluted on a chirp input, no input, and a 1 Hz sine signal.

The Expanding-HNN and Interp-HNN models predicted trajectories are close to the nominal ones on the testing trajectories with chirp input as can be observed in Figures 23a and 24a. Additionally they also predict trajectories close to the nominal trajectories when there is no input and a 1 Hz sine input as can be seen in Figures 23b, 23c, 24b and 24c. Note that for all of the multi-level strategies after the prediction horizon is increased beyond the training horizon the model's predictions drift away from the nominal trajectory. Also, the p_1 and p_2 trajectories are very poorly approximated, in every figure we can see that their predicted trajectories diverge from the nominal p_1 and p_2 (this is more pronounced for p_2).

The other multi-level strategies performed worse and are, therefore, not presented here.

5 Conclusion

In this project, three new frameworks have been introduced: Input-HNN, Expanding-HNN and Interp-HNN. These models have been tested and evaluated on the simple pendulum and Furuta pendulum. The simple pendulum results were satisfactory even with inputs that were not observed in the training set. In the case of the simple pendulum, the multi-level strategies did not give significantly better results but yielded results comparable to the Input-HNN architecture. The Interp-HNN architecture failed to give satisfactory results with both the Furuta and simple pendulum. Its variant, Interp-horizon-HNN, improved the results in experiments with the Furuta pendulum. Overall, the presented architectures had lower performances with the Furuta pendulum task than with the simple pendulum task.

Using the standard deviation rescaling and min-max rescaling in the loss function has proven to be beneficial on the Furuta pendulum task, particularly with the Expanding-HNN and Interp-HNN architecture. Furthermore, training a neural network that approximates $\mathcal{G}$ at the same time as the neural network that approximates $\mathcal{H}$, has been shown to be possible but did not improve the trajectory predictions with both dynamical systems.

There are limitations to the results presented in this report. Since training the Furuta pendulum models required a considerable amount of time, a grid-search on all of the experimental parameters such as the weights w_1 and w_2 or the size of the model was not performed. A method to quantitatively evaluate the differences between these models could involve running their training procedure n times and reporting the MSE of the predicted trajectories.

Future work on this project can examine one of the following tasks:

- **Using real trajectories for training:** Training the presented models or other models on trajectories generated by a real Furuta pendulum.
- **Finding a faster approach to solve neural ODEs:** Currently, we are using the `torchdiffeq` package to solve the previously formulated neural ODE at every time step. Increasing the number of time steps slows down training time. Unfortunately, since the ODEs must be solved sequentially, there is no way of parallelizing this process with respect to the number of time-steps. Therefore, using a GPU has limited benefits as it only gives a performance boost if we were using batches of trajectories. If benchmarks like [14] are correct, it may be interesting to see what the programming language Julia has to offer, as these benchmarks show a 39x speed improvement in their specific use case.
- **Defining a new optimization problem :** In [15] a new method for re-scaling the training targets in reinforcement learning is presented. This method consists of an optimization algorithm variant learning an optimal scaling for a score target while training a model to maximize it. Perhaps a variation of this method, where the parameters w_1 and w_2 in the loss function used in our models could be introduced as trainable parameters, could improve training.
- **Training one model but with multiple types of inputs :** Simply extending the current methods proposed but using multiple types of inputs such as chirp, step,

sinusoidal or a square signal. Having different types of inputs might make the learned Hamiltonian function give better predictions as the trajectories produced with different inputs will be richer in information.

- **Using an embedded form of the generalized coordinates :** As presented in [2] the embedding $(\cos \mathbf{q}, \sin \mathbf{q})$ can be used instead of $\mathbf{q}$, which could be beneficial for systems with angle coordinates.
- **Using a different framework that looks at a sequence of states to predict the next state:** Inspired from a discussion with my project supervisors, Clara Galimberti and Muhammad Zakwan, it could be interesting to try using another architecture to learn Hamiltonian dynamics. For example, a transformer neural network could take as input a sequence of N previous states with the last one being at time t_k, to predict the state at the time-step t_{k+1}. As these types of networks scale quadratically with the input sequence [16] and require large amounts of training data, the proposed idea has its limitations but should nonetheless be considered. There is also already some literature on this specific topic presented in [17]. In this context, it would be interesting to use linear transformers to speed up training and inference [16].

References

[1] S. Greydanus, M. Dzamba, and J. Yosinski, "Hamiltonian neural networks," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[2] Y. D. Zhong, B. Dey, and A. Chakraborty, "Symplectic ode-net: Learning hamiltonian dynamics with control," *arXiv preprint arXiv:1909.12077*, 2019.

[3] ——, "Dissipative symoden: Encoding hamiltonian dynamics with dissipation and control into deep learning," *arXiv preprint arXiv:2002.08860*, 2020.

[4] S. A. Desai, M. Mattheakis, D. Sondak, P. Protopapas, and S. J. Roberts, "Port-hamiltonian neural networks for learning explicit time-dependent dynamical systems," *Physical Review E*, vol. 104, no. 3, p. 034312, 2021.

[5] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, "Neural ordinary differential equations," *Advances in neural information processing systems*, vol. 31, 2018.

[6] PyTorch, "PyTorch Autograd," 2020. [Online]. Available: https://pytorch.org/docs/stable/autograd.html

[7] J. Perolini, "Learning hamiltonian functions from data semester project report," 2021.

[8] A. Rahman, J. Drgoňa, A. Tuor, and J. Strube, "Neural ordinary differential equations for nonlinear system identification," *arXiv preprint arXiv:2203.00120*, 2022.

[9] E. Haber and L. Ruthotto, "Stable architectures for deep neural networks," *Inverse problems*, vol. 34, no. 1, p. 014004, 2017.

[10] R. T. Q. Chen, "torchdiffeq," 6 2021. [Online]. Available: https://github.com/rtqichen/torchdiffeq

[11] PyTorch, "PyTorch gradient norm clipping source code," 2022. [Online]. Available: https://pytorch.org/docs/stable/_modules/torch/nn/utils/clip_grad.html#clip_grad_norm_

[12] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[13] A. Wadi, J.-H. Lee, and L. Romdhane, "Nonlinear sliding mode control of the furuta pendulum," in *2018 11th International Symposium on Mechatronics and its Applications (ISMA)*. IEEE, 2018, pp. 1–5.

[14] C. Rackauckas, "torchdiffeq vs Julia DiffEqFlux Neural ODE Training Benchmark," 2020. [Online]. Available: https://gist.github.com/ChrisRackauckas/4a4d526c15cc4170ce37da837bfc32c4

[15] H. P. van Hasselt, A. Guez, M. Hessel, V. Mnih, and D. Silver, "Learning values across many orders of magnitude," *Advances in Neural Information Processing Systems*, vol. 29, 2016.

[16] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are rnns: Fast autoregressive transformers with linear attention," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5156–5165.

[17] N. Geneva and N. Zabaras, "Transformers for modeling physical systems," *Neural Networks*, vol. 146, pp. 272–289, 2022.