AN IMPLEMENTATION OF NEURAL ORDINARY DIFFERENTIAL EQUATIONS

RODRIGO ARRIETA*

1. Introduction. In this report a review of Neural Ordinary Differential Equations (Neural ODEs), as well as an implementation coded in Julia, is presented. Neural ODEs are a class of models that combine neural networks with ordinary differential equations (ODEs). They were introduced by Chen et al. in 2018 [1] as a way to learn continuous-time dynamics directly from data.

Unlike traditional approaches, e.g. recurrent neural networks, where data is modeled through a discrete sequence of compositions and transformations, in Neural ODEs data is modeled continuously. In this case, a neural network is used to model the derivative of the data $\boldsymbol{u} : \mathbb{R} \to \mathbb{R}^N$, which is assumed to be differentiable and continuously dependent on time:

$$\frac{d}{dt}\boldsymbol{u}(t) = \mathrm{NN}(\boldsymbol{u}(t), t, \boldsymbol{\theta}), \tag{1.1}$$

where NN correspond to a neural network, t is the time variable and $\boldsymbol{\theta} \in \mathbb{R}^M$ are the neural network weights. Then, given an initial condition $\boldsymbol{u}_0 \equiv \boldsymbol{u}(t_0) \in \mathbb{R}^N$, the model prediction at time t_1 is obtained through any standard ODE solver, this is

$$\boldsymbol{u}(t_1) = \boldsymbol{u}_0 + \int_{t_0}^{t_1} \mathrm{NN}(\boldsymbol{u}(t), t, \boldsymbol{\theta})\, \mathrm{d}t \tag{1.2}$$

$$= \mathrm{ODESolver}(\mathrm{NN}, \boldsymbol{u}_0, t_0, t_1). \tag{1.3}$$

A key advantage of Neural ODEs is their ability to model continuous-time dynamics, making them suitable for tasks where time plays a crucial role, such as time series forecasting, trajectory prediction and generative modeling. By learning the system's latent dynamics, Neural ODEs can capture complex temporal dependencies and generalize well to unseen time points.

Neural ODEs have shown promising results in various domains, including computer vision, natural language processing, and physics, financial and biological simulations [6]. They provide a flexible framework for modeling continuous-time dynamics using neural networks, enabling a deeper understanding of temporal phenomena and latent dynamics in data.

This report is organized as follow. Section 2 explains how backpropagation is performed in Neural ODEs. Sec. 3 presents our in-house implementation of Neural ODE adjoint solver, coded in Julia from scratch and built on top of the Flux.jl library, alongside with some examples of interests. Finally, Sec. 4 presents Hamiltonian Neural Networks, a special case of Neural ODEs specially designed to deal with Hamiltonian systems. Our code and examples can be found in here[1].

2. Backpropagation of Neural ODEs. Training Neural ODEs involves solving an optimization problem to find the parameters that minimize a given loss function, which can be done using gradient-based methods. In the context of Neural ODEs,

*Mathematics Department, Massachusetts Institute of Technology, Cambridge, MA 02139, United States (rarrieta@mit.edu).

[1] https://github.com/Riarrieta/NeuralODEProject

to obtain the gradient of the loss function, a process called *backpropagation*, which is typically achieved through automatic differentiation in a typical neural network setting, is seldom performed. This is because backpropagating through the forward pass of the ODE solver (eq. 1.3) could be prohibitively expensive. Another reason is due to the potential limitation of the automatic differentiation engine's access to an external-library ODE solver. Instead, gradients are obtained through an adjoint sensitivity method, originally presented in [1]. This method is explained below.

Suppose the initial condition $\boldsymbol{u}_0$ is given and our loss function G has the form of an integral of our model prediction $\boldsymbol{u}$ in some interval $[0, T]$, i.e.

$$G(\boldsymbol{u}, \boldsymbol{\theta}) = \int_0^T g(\boldsymbol{u}(t, \boldsymbol{\theta}))\, \mathrm{d}t, \tag{2.1}$$

where we have made explicit the dependence of the weights $\boldsymbol{\theta}$ in $\boldsymbol{u}$. In a typical setting, we have access to a discrete set of observations $\{\tilde{\boldsymbol{u}}(t_i)\}_{i=1}^P$, with $t_i \in [0, T]$, and we wish to minimize the discrete L^2 error loss, or mean squared error, given by

$$G(\boldsymbol{u}, \boldsymbol{\theta}) = \frac{1}{P} \sum_{i=1}^{P} \left\| \tilde{\boldsymbol{u}}(t_i) - \boldsymbol{u}(t_i, \boldsymbol{\theta}) \right\|^2, \tag{2.2}$$

in which case

$$g(\boldsymbol{u}(t, \boldsymbol{\theta})) = \frac{1}{P} \sum_{i=1}^{P} \left\| \tilde{\boldsymbol{u}}(t_i) - \boldsymbol{u}(t, \boldsymbol{\theta}) \right\|^2 \delta(t - t_i), \tag{2.3}$$

where δ correspond to the Dirac delta function.

The goal of backpropagation is to obtain the gradient $\partial G(\boldsymbol{u}, \boldsymbol{\theta}) / \partial \boldsymbol{\theta} \in \mathbb{R}^M$ and use it to update the weights of our neural network. To do this we resort to the adjoint method, which consists in the following steps:

1. **Forward pass.** Solve for $\boldsymbol{u}$ in the interval $[0, T]$ using the ODE solver (eq. 1.3) and evaluate the loss function (2.1).
2. **Backward pass.** Define the adjoint variables $\boldsymbol{\lambda} : [0, T] \to \mathbb{R}^N$ and $\boldsymbol{\mu} : [0, T] \to \mathbb{R}^M$. They are solutions of the adjoint ODE

$$\frac{d}{dt}\boldsymbol{\lambda}(t) = -\left[\frac{\partial}{\partial \boldsymbol{u}} NN(\boldsymbol{u}(t), t, \boldsymbol{\theta})\right]^\top \boldsymbol{\lambda}(t) - \frac{d}{d\boldsymbol{u}} g(\boldsymbol{u}(t)), \tag{2.4a}$$

$$\frac{d}{dt}\boldsymbol{\mu}(t) = -\left[\frac{\partial}{\partial \boldsymbol{\theta}} NN(\boldsymbol{u}(t), t, \boldsymbol{\theta})\right]^\top \boldsymbol{\lambda}(t), \tag{2.4b}$$

$$\boldsymbol{\lambda}(T^+) = \mathbf{0}, \tag{2.4c}$$

$$\boldsymbol{\mu}(T^+) = \mathbf{0}. \tag{2.4d}$$

   Solve the adjoint ODE backwards in time, from T^+ to 0^-. Note the terms $\frac{\partial}{\partial \boldsymbol{u}} NN^\top \boldsymbol{\lambda}$ and $\frac{\partial}{\partial \boldsymbol{\theta}} NN^\top \boldsymbol{\lambda}$ can be efficiently evaluated using reverse-mode automatic differentiation and vector-Jacobian products.
3. **Evaluate gradient.** The gradient of the loss function with respect to the parameters is given by

$$\frac{\partial}{\partial \boldsymbol{\theta}} G(\boldsymbol{u}, \boldsymbol{\theta}) = \boldsymbol{\mu}(0^-). \tag{2.5}$$

4. **Update weights.** Finally, update the neural network weights using the previously obtained gradient.

Note that the backward pass requires the values of $\boldsymbol{u}(t)$ for $t \in [0, T]$ in order to solve the adjoint ODE (2.4). Therefore, it is necessary to either (A) store values of $\boldsymbol{u}$ in the forward pass and then use an interpolant of $\boldsymbol{u}$ during the backward pass, or (B) solve for $\boldsymbol{u}$ backward in time during the backward pass. Clearly, method (A) is more memory intensive, whereas method (B) is more compute intensive. Nevertheless, for method (B) it is important to mention that, although ODEs are theoretically reversible in time, it is possible that solving for $\boldsymbol{u}$ backward in time yields a different solution compared to the forward pass, due to the ODE solver discretization errors, thus resulting in an incorrect backward pass solution. For this reason, we have chosen method (A) for our in-house implementation of the Neural ODE adjoint solver (see Sec. 3).

3. An in-house implementation of a Neural ODE adjoint solver. In this section we present an in-house implementation of a Neural ODE adjoint solver. It is written in Julia and built on top of the machine learning library Flux.jl [5], the reverse-mode automatic differentiation library Zygote.jl [4] and the ODE solver library OrdinaryDiffEq.jl [9]. There are already existing packages that implement Neural ODE adjoint solvers, e.g. DiffEqFlux.jl [8], but we have decided to implement an adjoint solver from scratch for pedagogical purposes, not only to learn about Neural ODEs, but also with the purpose of us getting acquainted with the Flux.jl library. Our code and examples can be found in here[2].

In the following subsections, we will showcase some examples of interest conducted using our code. In all of these examples, we employed a neural network with a single hidden layer consisting of 64 hidden units, *tanh* activation functions, and a loss function given by the mean square error (2.2). We used the ODE solver *tsit5* of the OrdinaryDiffEq.jl package.

3.1. Example: linear system. Consider the linear constant-coefficient ODE

$$\frac{d}{dt}\boldsymbol{u}(t) = A\boldsymbol{u}(t) \tag{3.1}$$

in the interval $[0, 1]$, with

$$A = \begin{pmatrix} -0.1 & 2 \\ -2 & -0.1 \end{pmatrix}. \tag{3.2}$$

We trained our neural network with 11 uniformly spaced observation in $[0, 1]$ for various normal random initial conditions $\boldsymbol{u}_0 \sim \mathcal{N}(\mu = 0, \sigma = 4)$. A comparison between the true solution, for a random initial condition, and our Neural ODE solution is shown in Fig. 1. We observe that after enough training our Neural ODE solution is able to follow the true solution quite well inside the training interval. Furthermore, if we extend the ODE and Neural ODE solution outside the training interval, as shown in Fig. 2, the Neural ODE keeps up with the true solution if we go forward in time, where the amplitude of the true solution decreases. Nevertheless, if we go backward in time, the amplitude of the true solution increases and the Neural ODE is not able to follow. This in an expected behavior. For large amplitudes the nonlinear activation functions start to kick in, removing the neural network from its linear behavior,

[2] https://github.com/Riarrieta/NeuralODEProject

thus the (nonlinear) Neural ODE cannot keep up with the true linear ODE. Clearly, this issue can be overcome by training the neural network with initial conditions of larger and larger amplitude, however, for a fixed neural network, a sufficiently large amplitude will remove it from its linear behavior. This simple example illustrates the limitation of using Neural ODEs to model ODEs with unbounded solutions over time.

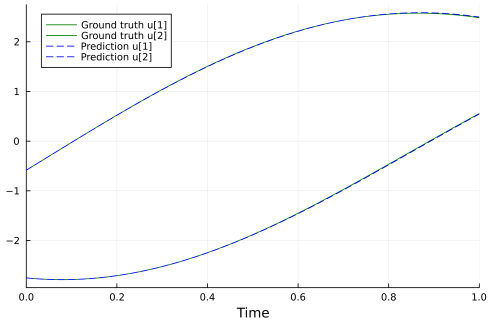

FIGURE 1. *Example 3.1. Comparison between the true solution and our Neural ODE solution in the training interval* $[0, 1]$.

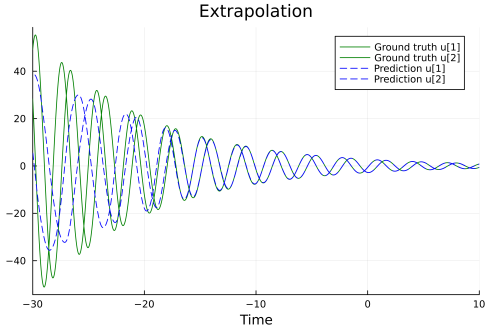

FIGURE 2. *Example 3.1. Extrapolation of the ODE and Neural ODE solution outside the training interval.*

3.2. Example: Lotka-Volterra equations. Consider the Lotka-Volterra equations

$$\frac{d}{dt}x(t) = \alpha x(t) - \beta x(t)y(t), \tag{3.3}$$

$$\frac{d}{dt}y(t) = \delta x(t)y(t) - \gamma y(t), \tag{3.4}$$

in the interval $[0, 10]$ with parameters $(\alpha, \beta, \delta, \gamma) = (2/3, 4/3, 1, 1)$ and initial condition $(x_0, y_0) = (1, 1)$. This nonlinear system of ODEs is used to model prey-predator dynamics, with the variables x and y representing population densities of prey (e.g. rabbits) and predator (e.g. foxes), respectively. Solutions of the Lotka-Volterra eqs. are periodic, with period $T = 2\pi/\sqrt{\alpha\gamma}$. In our case, $T \approx 7.7$, and the training interval $[0, 10]$ is slightly longer than the period.

A comparison between the true solution and our Neural ODE solution in the training interval is shown in Fig. 3, and a comparison outside the training interval

is shown in Fig. 4. Theu neural network was trained with 101 uniformly spaced observations. Again, it is observed that the Neural ODE is able to keep up with the true ODE solution in the training interval, and outside of it if we extrapolate forward in time. It is extraordinary the fact that the Neural ODE is able to capture the nonlinear dynamics of the system in the training interval and then replicate the periodic behavior of the solution forward in time. However, the Neural ODE is not able to replicate the true solution backward in time. The causes of this are unknown, but we believe this issue can be addressed by increasing the length of the training interval.

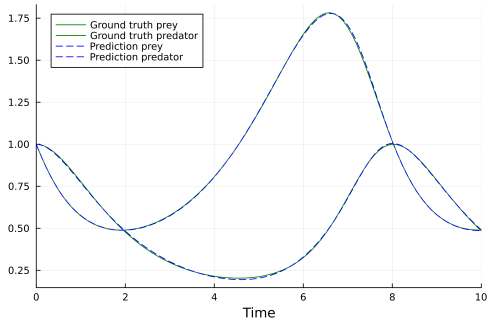

FIGURE 3. *Example 3.2. Comparison between the true solution and our Neural ODE solution in the training interval* $[0, 10]$.

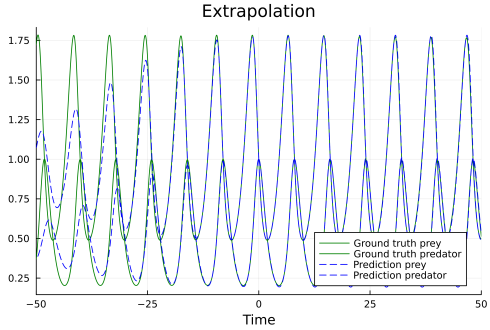

FIGURE 4. *Example 3.2. Extrapolation of the ODE and Neural ODE solution outside the training interval.*

3.3. Example: Lotka-Volterra equations with noise. In this example we test the performance of the Neural ODE when the observations contain noise. Here, we take the same Lotka-Volterra eqs. and parameters of Sec. 3.2, but we add white gaussian noise $\eta \sim \mathcal{N}(\mu = 0, \sigma = 0.1)$ to each of the observation points, and we reduce the number of observations from 101 to 41.

A comparison between the true solution, the noisy observations and our Neural ODE solution in the training interval is shown in Fig. 5, and a comparison outside the training interval is shown in Fig. 6. We observe that the Neural ODE possess robustness to noise, since it is still able to follow closely the true solution and replicate its periodic behavior, even in the presence of noisy observations.

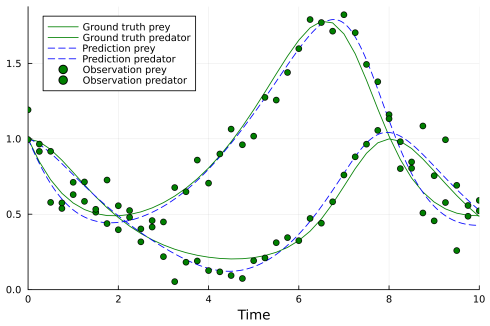

FIGURE 5. *Example 3.3. Comparison between the true solution, the noisy observations and our Neural ODE solution in the training interval* $[0, 10]$.

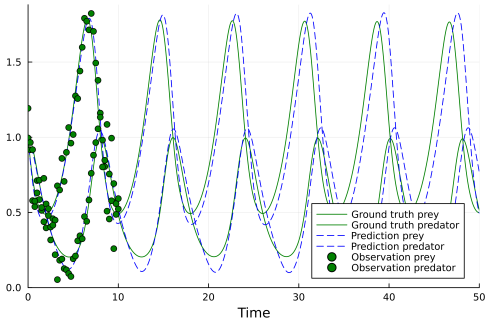

FIGURE 6. *Example 3.3. Extrapolation of the ODE and Neural ODE solution outside the training interval. The noisy observations inside the training interval are also shown.*

3.4. Example: double pendulum. In this example we consider the dynamics of a double pendulum, as depicted in Fig. 7. Following the Hamiltonian formalism of classical mechanics, we have that the double pendulum has a Hamiltonian given by

$$H(\boldsymbol{q},\boldsymbol{p}) = \frac{m_2 l_2^2 p_{\theta_1}^2 + (m_1+m_2)\, l_1^2 p_{\theta_2}^2 - 2m_2 l_1 l_2 p_{\theta_1} p_{\theta_2} \cos(\theta_1-\theta_2)}{2m_2 l_1^2 l_2^2 \left[m_1 + m_2 \sin^2(\theta_1-\theta_2)\right]} - (m_1+m_2)\, g l_1 \cos\theta_1 - m_2 g l_2 \cos\theta_2, \tag{3.5}$$

where $\boldsymbol{q} = (\theta_1, \theta_2)$ are the pendulums angles, $\boldsymbol{p} = (p_{\theta_1}, p_{\theta_2})$ the canonical momenta of the system, (m_1, m_2) the pendulum masses, (l_1, l_2) the pendulum lengths and g the gravitational acceleration. The canonical momenta $(p_{\theta_1}, p_{\theta_2})$ have analytical expressions given in terms of $(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2)$, but those expressions are not relevant for our purposes. The dynamics of the double pendulum follow Hamilton's equations, given by

$$\frac{d\boldsymbol{q}}{dt} = \frac{\partial H(\boldsymbol{q},\boldsymbol{p})}{\partial \boldsymbol{p}}, \tag{3.6a}$$

$$\frac{d\boldsymbol{p}}{dt} = -\frac{\partial H(\boldsymbol{q},\boldsymbol{p})}{\partial \boldsymbol{q}}. \tag{3.6b}$$

The double pendulum equations are nonlinear, exhibit chaotic behavior, and the solutions are in general aperiodic.

We compared the true ODE solution against a Neural ODE trained in the interval $[0, 0.25]$ with initial conditions $\boldsymbol{q} = (0.05, -0.05)$, $\boldsymbol{p} = (0, 0)$ and 26 uniformly spaced observations. We took $(m_1, m_2, l_1, l_2, g) = (1, 1, 1, 1, 9.8)$. Our results are shown in Figs. 8 and 9, for comparisons inside and outside the training interval, respectively. As in the previous examples, the Neural ODE is able to follow the true solution inside the training interval, and even replicate the aperiodic and complex behavior outside of it.

In a Hamiltonian system, such as the double pendulum, if Hamiltonian (3.5) is independent of time, then the Hamiltonian itself must be a conserved quantity. A comparison of the Hamiltonians of the true ODE and the Neural ODE is shown in Fig. 10. Although in theory the Hamiltonian of the true ODE solution should be conserved, we see it is not the case, in fact, it is slowing increasing. This is due to the ODE solver, which is not designed to conserve the Hamiltonian over time. A class of ODE solvers denominated symplectic solvers are able to maintain conserved quantities over time [7], but they were not used in this work.

In the Neural ODE case we observe the Hamiltonian decreases over time, it is not conserved. This is mainly due to the fact that Neural ODE does not follow a Hamiltonian structure, such as 3.6, hence the Hamiltonian is not necessary conserved. This poses a challenge as it enables the Neural ODE to evolve towards prohibited states. The system's allowed states are limited to those that possess the same Hamiltonian value as the initial condition. We will explain in Sec. 4 how this difficulty can be overcome with the use of Hamiltonian Neural Networks.

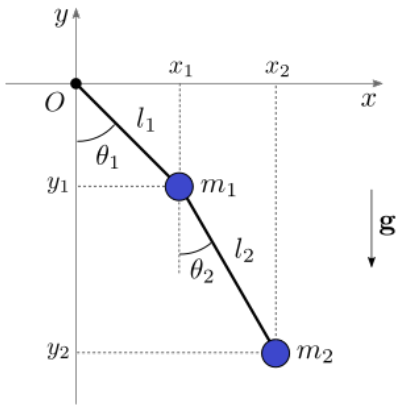

FIGURE 7. *Example 3.4. Schematic of a double pendulum.*

4. Hamiltonian Neural Networks. Hamiltonian Neural Networks (HNNs) are a subclass of Neural ODEs specifically design to deal with Hamiltonian system, proposed by Greydanus et at. in 2019[3]. Unlike the Neural ODEs approach, which would replace the right-hand-side of Hamilton's eqs. (3.6) with a neural network, in HNNs the Hamiltonian itself is modeled with a neural network, this is

$$H(\boldsymbol{q}, \boldsymbol{p}) = \mathrm{NN}(\boldsymbol{q}, \boldsymbol{p}, \boldsymbol{\theta}), \tag{4.1}$$

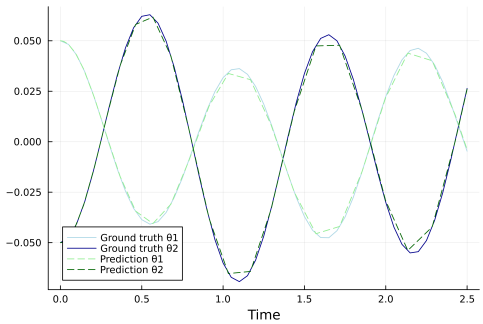

FIGURE 8. *Example 3.4. Comparison between the true solution and our Neural ODE solutions in the training interval* $[0, 10]$.

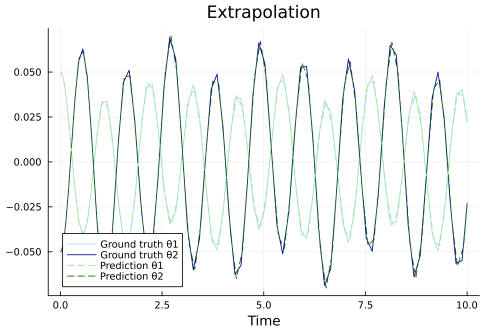

FIGURE 9. *Example 3.4. Extrapolation of the ODE and Neural ODE solutions outside the training interval.*

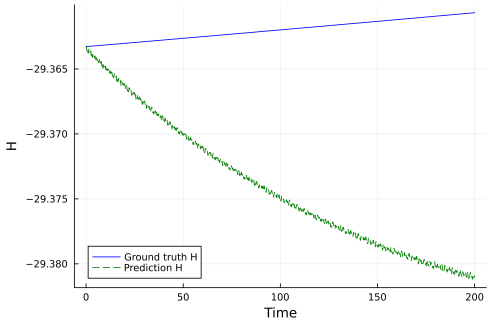

FIGURE 10. *Example 3.4. Hamiltonian of ODE and Neural ODE solutions outside the training interval.*

and thus Hamilton's eqs. now read

$$\frac{d\boldsymbol{q}}{dt} = \frac{\partial}{\partial \boldsymbol{p}} \mathrm{NN}(\boldsymbol{q}, \boldsymbol{p}, \boldsymbol{\theta}), \tag{4.2a}$$

$$\frac{d\boldsymbol{p}}{dt} = -\frac{\partial}{\partial \boldsymbol{q}} \mathrm{NN}(\boldsymbol{q}, \boldsymbol{p}, \boldsymbol{\theta}). \tag{4.2b}$$

The fact that HNNs preserve the underlying Hamiltonian structure of the system

lead to unique advantages, such as the (theoretical) conservation energy and long-term stability, useful in accurate long-term predictions.

Some difficulties of HNNs are, first, that in order to perform backpropagation we require to take second order derivatives of our neural network. To see this, note that we need first order derivatives in the forward pass (eq. 4.2), and then take derivatives one more time in the backward pass (eq. 2.4). This is expensive, and some automatic differentiation libraries, such as Zygote.jl, have limited capabilities with respect to second or higher order derivatives. Second, in some system it is not obvious how to compute or measure the canonical momenta $\boldsymbol{p}$, hence in those cases we cannot formulate the dynamics of the system of interest with the Hamiltonian formalism. An alternative to HNNs, namely Lagrangian Neural Networks [2], uses the Lagrangian formalism instead of the Hamiltonian formalism, and thus avoids the computation of canonical momenta, it only requires generalized coordinates $\boldsymbol{q}$ and generalized velocities $\dot{\boldsymbol{q}}$. Nevertheless, it presents other difficulties on its own: it requires third order derivatives and to solve linear systems in both the forward and backward passes.

4.1. Example: hamiltonian double pendulum. Here we repeat Example 3.4, but using a HNN instead. This time we resort to the DiffEqFlux.jl package for a HNN implementation, instead of an in-house implementation of our own, just for showcasing purposes, and additionally because the Zygote.jl library presents difficulties to obtain second order derivatives of neural networks with respect to their weights, which are required for backpropagation. Furthermore, for simplicity we utilize the collocation method, detailed in [10], to train the HNN.

The HNN is able to keep up with the true solution, both inside the training interval and forward in time, just like the Neural ODE counterpart. However, this time our model posses a Hamiltonian structure, hence the Hamiltonian should be conserved, at least theoretically. A comparison between the Hamiltonians of the ODE solution and the HNN is shown in Fig. 11. We observe that both Hamiltonians grow slowly at the same rate. As pointed out in Example 3.4, this growing is due to numerical errors in the ODE solver, and not due to the lack of Hamiltonian structure. It can be said with certain then that our HNN is able to correctly reproduce the Hamiltonian structure of the system and to conserve the Hamiltonian. We are secure that, if we employed a symplectic solver, we would observe a constant Hamiltonian, in both the ODE and HNN solutions.

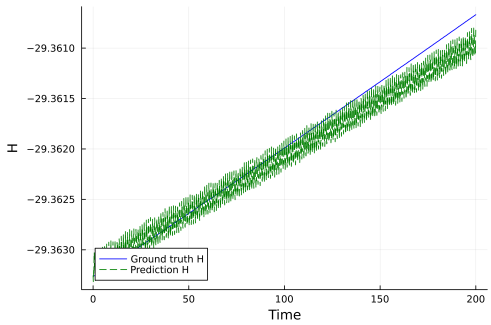

FIGURE 11. *Example 4.1. Hamiltonian of ODE and HNN solutions outside the training interval.*

REFERENCES

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

[2] M. Cranmer, S. Greydanus, S. Hoyer, P. Battaglia, D. Spergel, and S. Ho, *Lagrangian neural networks*, arXiv preprint arXiv:2003.04630, (2020).

[3] S. Greydanus, M. Dzamba, and J. Yosinski, *Hamiltonian neural networks*, Advances in neural information processing systems, 32 (2019).

[4] M. Innes, *Don't unroll adjoint: Differentiating ssa-form programs*, CoRR, abs/1810.07951 (2018), http://arxiv.org/abs/1810.07951, https://arxiv.org/abs/1810.07951.

[5] M. Innes, E. Saba, K. Fischer, D. Gandhi, M. C. Rudilosso, N. M. Joy, T. Karmali, A. Pal, and V. Shah, *Fashionable modelling with flux*, CoRR, abs/1811.01457 (2018), https://arxiv.org/abs/1811.01457, https://arxiv.org/abs/1811.01457.

[6] P. Kidger, *On neural differential equations*, arXiv preprint arXiv:2202.02435, (2022).

[7] B. Leimkuhler and S. Reich, *Simulating hamiltonian dynamics*, no. 14, Cambridge university press, 2004.

[8] C. Rackauckas, M. Innes, Y. Ma, J. Bettencourt, L. White, and V. Dixit, *Diffeqflux.jl - A julia library for neural differential equations*, CoRR, abs/1902.02376 (2019), https://arxiv.org/abs/1902.02376, https://arxiv.org/abs/1902.02376.

[9] C. Rackauckas and Q. Nie, *Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia*, The Journal of Open Research Software, 5 (2017), https://doi.org/10.5334/jors.151, https://app.dimensions.ai/details/publication/pub.1085583166andhttp://openresearchsoftware.metajnl.com/articles/10.5334/jors.151/galley/245/download/. Exported from https://app.dimensions.ai on 2019/05/05.

[10] E. Roesch, C. Rackauckas, and M. P. Stumpf, *Collocation based training of neural ordinary differential equations*, Statistical Applications in Genetics and Molecular Biology, 20 (2021), pp. 37–49.