Solving Differential Equations Using Neural Network Solution Bundles

Cedric Flamant
Department of Physics
Harvard University
Cambridge, MA, 02138
cflamant@g.harvard.edu

Pavlos Protopapas
John A. Paulson School of Engineering and Applied Science
Harvard University
Cambridge, MA, 02138
pavlos@seas.harvard.edu

David Sondak
John A. Paulson School of Engineering and Applied Science
Harvard University
Cambridge, MA 02138
dsondak@seas.harvard.edu

Abstract

The time evolution of dynamical systems is frequently described by ordinary differential equations (ODEs), which must be solved for given initial conditions. Most standard approaches numerically integrate ODEs producing a single solution whose values are computed at discrete times. When many varied solutions with different initial conditions to the ODE are required, the computational cost can become significant. We propose that a neural network be used as a solution bundle, a collection of solutions to an ODE for various initial states and system parameters. The neural network solution bundle is trained with an unsupervised loss that does not require any prior knowledge of the sought solutions, and the resulting object is differentiable in initial conditions and system parameters. The solution bundle exhibits fast, parallelizable evaluation of the system state, facilitating the use of Bayesian inference for parameter estimation in real dynamical systems.

1 Introduction

Many dynamical systems are described by ordinary differential equations (ODEs) which relate the rates and values of state variables and external driving functions. While some simple ODEs have closed form solutions to them, the vast majority have to be solved approximately using discretization of the domain or by using spectral methods[1]. The former approximating methods are more general, with Runge-Kutta and multi-step methods as typical examples. These methods seek to numerically integrate the ODEs, starting from initial conditions and stepping forward until the desired final time is attained. While these conventional methods are generally efficient for determining the state of a system for a sequence of times, if we are only interested in the state at a specific later time, substantial computational effort must still be expended determining all the states at steps leading up to the state of interest. This causal order also limits parallelizability of the conventional single- and multi-step methods since the task cannot be parallelized in time—until the preceeding state is known, processors tasked with finding a segment of the system's evolution over a later time interval cannot start calculating the correct piece of the trajectory.

Previous work

The idea of approximating the solutions of differential equations with neural networks was first developed by Dissanayake and Phan-Thien, where training was accomplished by minimizing a

loss based on the network's satisfaction of the boundary conditions and the differential equations themselves[2]. Lagaris *et al.* showed that the form of the network could be chosen to satisfy boundary conditions by construction, and that automatic differentiation could be used to compute the derivatives appearing in the loss function[3]. After training, the approximate value of the solution at any point within the training range can be computed in constant time without having to compute previous states first. This method has been extended to systems with irregular boundaries[4, 5], applied to solving PDEs arising in fluid mechanics[6], and software packages have been developed to facilitate its application[7, 8, 9]. In the Lagaris approach, the neural network learns a single solution to the ODE. For different sets of initial conditions, or for different sets of parameters in the differential equation, the network has to be retrained on the new task.

Our contributions

We propose an extension of the Lagaris method[3] where the neural network is instead taught a variety of solutions to a parameterized differential equation. This increases the reusability of the trained network and can speed up tasks that require knowing many solutions, such as for Bayesian parameter inference, propagating uncertainty distributions in dynamical systems, or inverse problems. While it is straightforward to extend our approach to all the situations considered in the Lagaris paper, *i.e.* for problems containing various types of boundary conditions, for partial differential equations and higher derivatives, we focus on initial value problems in first-order ODEs. We show that our method has promise when applied to a variety of tasks requiring quick, parallel evaluation of multiple solutions to an ODE, and where it is useful to be able to differentiate the state at a particular time with respect to the initial conditions or ODE parameters. Our contributions also include weighting the loss function to improve global error and demonstrating an application of curriculum learning to help with training. With the rapid advances in neural network development, as well as its supporting hardware, employing this method will become cheaper and more efficient, further extending its applicability.

2 Solution Bundles

When working with ODEs it is common to require multiple solutions corresponding to different initial conditions. In dynamical systems, each of these solutions represents a trajectory, tracing out an alternate time evolution of the system state. In addition, when an ODE is parameterized, say by a physical constant whose value has an associated uncertainty, it can be useful to have different solutions for various values of the parameters.

2.1 Method Description

Consider the following general first-order differential equation parameterized by $\boldsymbol{\theta}$:

$$\mathbf{G}\left(\mathbf{x}, \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}, t \; ; \; \boldsymbol{\theta}\right) = 0, \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^n$ is a vector of state variables, t is time, and $\boldsymbol{\theta} \in \mathbb{R}^p$ are physical parameters associated with the dynamics. We assume that the ODE describes a deterministic system where initial conditions $\mathbf{x}_0$ of the state variables uniquely determines a solution. The solutions to Eq. (1) over the time range $[t_0, t_f]$, together with a subset of initial conditions $X_0 \subset \mathbb{R}^n$, and a set of parameters $\Theta \subset \mathbb{R}^p$, define a *solution bundle* $\mathbf{x}(t; \mathbf{x}_0, \boldsymbol{\theta})$, where $\mathbf{x_0} \in X_0, \boldsymbol{\theta} \in \Theta$.

Let the approximating function to Eq. (1) for the solution bundle over (X_0, Θ) be given by

$$\hat{\mathbf{x}}(t \; ; \; \mathbf{x}_0, \boldsymbol{\theta}) = \mathbf{x}_0 + a(t)\mathbf{N}(t \; ; \; \mathbf{x}_0, \boldsymbol{\theta} \; ; \; \mathbf{w}), \tag{2}$$

where $\mathbf{N} : \mathbb{R}^{1+n+p} \to \mathbb{R}^n$ is a neural network with weights $\mathbf{w}$, and $a : [t_0, t_f] \to \mathbb{R}$ satisfies $a(t_0) = 0$. This form explicitly constrains the trial solution to satisfy the initial condition $\mathbf{x}(t_0) = \mathbf{x}_0$. The choice of $a(t)$ can affect the ease of training the network. While $a(t) = t - t_0$ is sufficient, Mattheakis *et al.* demonstrated that $a(t) = 1 - e^{-(t-t_0)}$ results in better convergence in the Lagaris method[10], and we observe similar benefits in our extension, likely due to its diminishing effect farther from the initial time. We primarily use multilayer fully-connected neural networks in our experiments, but it is worth exploring other architectures in the future.

The unsupervised loss function used in training has the form

$$L = \frac{1}{|B|} \sum_{(t_i, \mathbf{x}_{0i}, \boldsymbol{\theta}_i) \in B} b(t_i) \left| \mathbf{G}\left(\hat{\mathbf{x}}(t_i \; ; \; \mathbf{x}_{0i}, \boldsymbol{\theta}_i), \frac{\partial \hat{\mathbf{x}}(t_i \; ; \; \mathbf{x}_{0i}, \boldsymbol{\theta}_i)}{\partial t}, t_i \; ; \; \boldsymbol{\theta}_i \right) \right|^2, \tag{3}$$

where the set $B = \{(t_i, \mathbf{x}_{0i}, \boldsymbol{\theta}_i)\}$ constitutes a training batch of size $|B|$, with $t_i \in [t_0, t_f]$, $\mathbf{x}_{0i} \in X_0$, and $\boldsymbol{\theta}_i \in \Theta$ drawn from some distribution over their respective spaces. The function $b : [t_0, t_f] \to \mathbb{R}$ appearing in Eq. (3) is used to weight data points based on their time, as will be discussed later. We typically use $b(t) = \exp(-\lambda(t - t_0))$, with hyperparameter $\lambda > 0$. We found that uniform sampling over the spaces $[t_0, t_f]$, X_0, and Θ, usually works well in practice, but there are situations where it is helpful to use a different distribution to generate batches. For example, for the FitzHugh-Nagumo model discussed in Section 5, a batch-number-dependent distribution over times $[t_0, t_f]$ was used for curriculum learning. Finally, in the loss function, the time derivative of $\hat{\mathbf{x}}(t)$ is computed using automatic differentiation.

There is no concept of an epoch in the training process of this method since every batch will be a unique sample of size $|B|$ from the distribution across times, initial conditions, and parameters. This is similar to meshfree implementations of the Lagaris method, which have been shown to work better than operating on a fixed grid[11]. The model cannot overfit as we are effectively operating in a regime of infinite data. The training ends when the approximation to the solution bundle is deemed acceptable, based on either the history of past losses or on some other metric, like its difference compared to a few solution curves computed via conventional methods. Ideally, the training process should be carried out until the network weights are locally optimal, though in general the approximation will still differ from the exact solution. To obtain incrementally better solution bundles, one could increase the complexity of the network since the universal approximation theorem[12, 13] guarantees the existence of a better approximating network to the true solution bundle.

At the end of training, the neural network solution bundle $\hat{\mathbf{x}}(t; \mathbf{x}_0, \boldsymbol{\theta})$ can be used to evaluate, in constant execution time, the approximate value of state $\mathbf{x}$ at any time $t \in [t_0, t_f]$ for any initial condition $\mathbf{x}_0 \in X_0$ and parameters $\boldsymbol{\theta} \in \Theta$. Our approach is highly amenable to parallelization both during the training and inference stages. The training is parallelized in time by distributing the points in batch B across each processing unit, as seen in Section 3.1.1. During the inference stage, the evaluation of the neural network can be parallelized, unlike in stepping methods where steps have to be computed sequentially, and each step consists of too few operations to benefit from parallelization. These strengths of our method are useful when the behavior of a distribution of solutions is desired, such as for propagating state uncertainty or for performing Bayesian inference.

The neural network solution bundle also has a closed analytic form and is differentiable in all of its inputs. This capability can be used for a variety of useful tasks. For example, in Bayesian estimation differentiability in the initial conditions and ODE parameters enables gradient-based maximization, simplifying the calculation of maximum a posteriori (MAP) estimates of these quantities. It also simplifies the application of "shooting methods," where a condition at a later time is known and the parameters and initial conditions that are consistent with the constraint are sought. Differentiability can also be useful for more general optimization tasks and sensitivity studies.

2.1.1 Discussion of the Weighting Function.

Given that the approximating function Eq. (2) will generally not be able to perfectly satisify the target ODE everywhere, there will always be some bias error or epistemic uncertainty regardless of the choice of $\mathbf{w}$. If we do not weight the loss function Eq. (3), *i.e.* $b(t) = 1$, we do not get to influence how the local error

$$\boldsymbol{\varepsilon}(t_i \; ; \; \mathbf{x}_{0i}, \boldsymbol{\theta}_i) \equiv \mathbf{G}\left(\hat{\mathbf{x}}(t_i \; ; \; \mathbf{x}_{0i}, \boldsymbol{\theta}_i), \frac{\partial \hat{\mathbf{x}}(t_i \; ; \; \mathbf{x}_{0i}, \boldsymbol{\theta}_i)}{\partial t}, t_i \; ; \; \boldsymbol{\theta}_i \right) \tag{4}$$

is distributed across the training region. However, by applying an exponentially decaying weight $b(t) = e^{-\lambda(t-t_0)}$, we can convey in the loss function the larger contribution of early-time local errors to our metric of interest, the global error. An appeal for using a decaying exponential weighting comes from considering an upper-bound on the global error. Consider a differential equation of the form $dx / dt = f(t, x)$. The local error Eq. (4) is $\varepsilon(t) = d\hat{x} / dt - f(t, \hat{x})$, and it can be shown

(Appendix A) that the global error $\epsilon(t) = \hat{x}(t) - x(t)$ is bounded by

$$|\epsilon(t)| \leq \frac{\varepsilon_{t'}}{L_f}\left(e^{L_f(t-t_0)} - 1\right),\tag{5}$$

where L_f is the Lipschitz constant of f, and $\varepsilon_{t'} \equiv \max_{t_0 \leq t \leq t'}|\varepsilon(t)|$. This bound shows that global error can grow exponentially as a result of an early local error, so an exponential form for the weighting function $b(t)$ incorporates this relative importance in the loss.

Note that in the above discussion we assume scarce knowledge about the specifics of an ODE, showing that an exponential weighting is generally sensible. Specific weighting functions can likely be tailored to outperform exponential weighting for a given ODE.

3   Propagating a Distribution

A neural network solution bundle provides a mapping from initial states to states at later times. This can be useful for time-evolving a distribution over initial conditions to obtain a probability distribution over states at later time t. Given a probability density over initial states $p_0(\mathbf{x}_0)$, we note that the solution bundle $\mathbf{x}(t \,;\, \mathbf{x}_0)$ at time t describes a coordinate transformation from $\mathbf{x}_0$ to $\mathbf{x}_t$. In practice we can simply sample the initial state space and construct a histogram of output $\mathbf{x}_t$ states using the solution bundle. The sampling of initial states can be done according to $p_0(\mathbf{x}_0)$ using Markov chain Monte Carlo (MCMC) methods, or if the dimensionality of the state vector is low enough, by simply performing a uniform sampling over the initial states and constructing a weighted histogram of $\mathbf{x}_t$, weighting each sample by $p_0(\mathbf{x}_0)$ for the $\mathbf{x}_0$ that generated it. Appendix B explains how an analytic expression for the transformed probability density can also be obtained by making use of the differentiability of the solution bundle.

3.1   Planar Circular Restricted Three-Body Problem

The planar circular restricted three-body problem describes a special case of the motion of three masses under Newton's law of universal gravitation. This special case describes the motion of the third body, which is assumed to have negligible mass, in the co-rotating frame of the first two bodies in circular orbits around their barycenter. All three bodies are also assumed to lie in the same plane (x-y plane), with no velocity in the z direction. For clarity of discussion, let body 1 be the Earth, body 2 be the Moon, and body 3 be an asteroid. The asteroid has position $\mathbf{r}(t) = (x(t), y(t))^\mathsf{T}$ and velocity $\mathbf{u}(t) = (u(t), v(t))^\mathsf{T}$. We will call the full state vector $\mathbf{q} = (\mathbf{r}^\mathsf{T}, \mathbf{u}^\mathsf{T})^\mathsf{T}$. The nondimensionalized mass of the Earth is given by $m_1 = 1 - \mu$, and the mass of the Moon is $m_2 = \mu$, where μ is the ratio of the mass of the the Moon to the total mass of the pair. The nondimensionalized equations of motion of the asteroid are given by[14]:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = u\,, \qquad \frac{\mathrm{d}u}{\mathrm{d}t} = x - \mu + 2v - \left[\frac{\mu(x-1)}{\left((x-1)^2 + y^2\right)^{3/2}} + \frac{(1-\mu)x}{\left(x^2 + y^2\right)^{3/2}}\right],\tag{6}$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = v\,, \qquad \frac{\mathrm{d}v}{\mathrm{d}t} = y - 2u - \left[\frac{\mu y}{\left((x-1)^2 + y^2\right)^{3/2}} + \frac{(1-\mu)y}{\left(x^2 + y^2\right)^{3/2}}\right].\tag{7}$$

3.1.1   Training the Solution Bundle

We trained a fully-connected MLP with 8 hidden layers of 128 neurons per layer uniformly over the x_0, y_0, u_0, v_0 initial condition space $X_0 = [1.05, 1.052] \times [0.099, 0.101] \times [-0.5, -0.4] \times [-0.3, -0.2]$ and times $[-0.01, 5]$, and used a fixed parameter $\mu = 0.01$. Even though we only intend to evaluate the solution bundle at times $t \in [0, 5]$, we found that including times slightly earlier than t_0 in training helps improve accuracy. This makes the approximating function satisfy the ODE on both sides of t_0, resulting in a more accurate value of the derivative term in Eq. (3) around $t = t_0$. We used batchsize $|B| = 10,000$, the Adam optimizer[15], and learning rate $\eta = 0.001$, which we reduced on plateau. For the weighting function $b(t)$ in the loss, Eq. (3), we chose $b(t) = \exp\left(-\lambda t\right)$ where $\lambda = 2$.

Figure 1 shows the adaptation of the neural network solution bundle during training with increasing batch number. It also demonstrates the parallelization in time of the method, with the curvature of
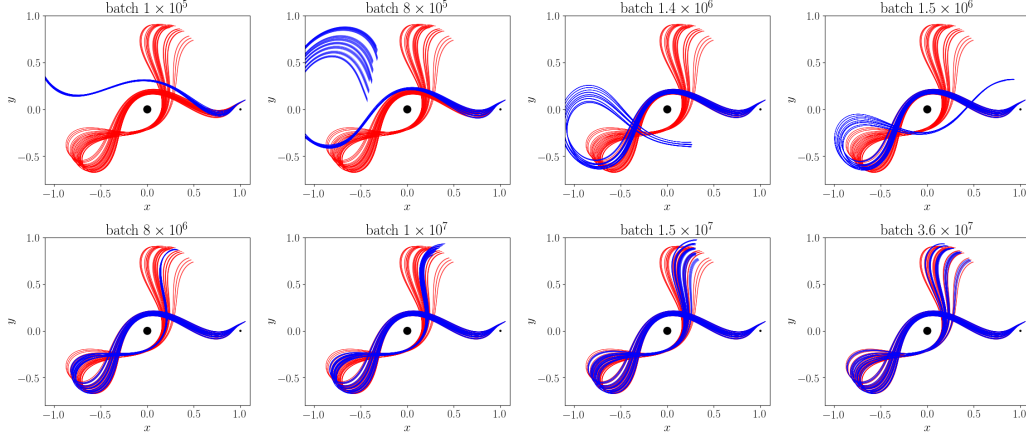
Figure 1: Plots of a few trajectories from the neural network solution bundle at various points in the training. Red trajectories are calculated with fourth-order Runge-Kutta, and the neural network solutions are shown in blue.

later parts of the trajectory adjusting even before the earlier states have settled. This is reminiscent of the dominant approach for parallel-in-time solving of differential equations, Parareal[16], which also involves computation of approximate later trajectories before earlier path segments are precisely known.

3.1.2 Propagation of Uncertainty

With the neural network solution bundle trained, we can use it to propagate distributions in time. Suppose we have two measurements of the position of an asteroid at two different times, along with some uncertainty. If the majority of the probability mass of these uncertainty distributions falls within the solution bundle, it is straightforward to compute a probability distribution for the future position of the asteroid by constructing a histogram weighted by the uncertainty distribution of the observations. Details of the calculation can be found in Appendix B.
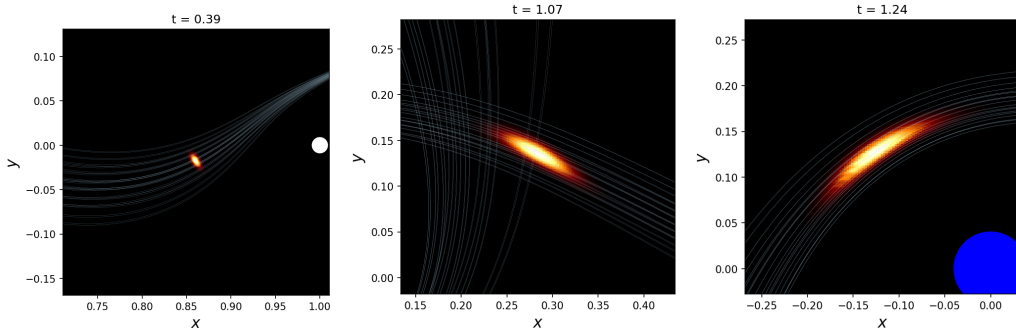

Figure 2: Probability distribution $p(\mathbf{x}(t) = (x, y)^\intercal)$ at various times. A few trajectories in the bundle are shown in white, and the distribution is shown as a heatmap. The full path is shown in Figure 1 and the full trajectory of the distribution is in Appendix B.1.

4 Bayesian Parameter Inference

Another common task that requires computing many solutions to an ODE is Bayesian parameter inference in systems described by differential equations. In the physical sciences and other fields employing mathematical modeling of data, it is often necessary to estimate parameters of a system based on experimental measurements, as well as to determine their uncertainties. If the system is described by differential equations, these parameters modify terms in the equations, resulting in

different families of solutions. The probability density of the initial conditions and parameters $\mathbf{x}_0, \boldsymbol{\theta}$ given a set of observed data $\{(t_i, \mathbf{x}_i)\}$ and prior $p(\mathbf{x}_0, \boldsymbol{\theta})$ can be computed from Bayes' theorem,

$$p\big(\mathbf{x}_0, \boldsymbol{\theta} \mid \{(t_i, \mathbf{x}_i)\}\big) = \frac{p\big(\{(t_i, \mathbf{x}_i)\} \mid \mathbf{x}_0, \boldsymbol{\theta}\big) p(\mathbf{x}_0, \boldsymbol{\theta})}{p(\{(t_i, \mathbf{x}_i)\})} \propto p\big(\{(t_i, \mathbf{x}_i)\} \mid \mathbf{x}_0, \boldsymbol{\theta}\big) p(\mathbf{x}_0, \boldsymbol{\theta}). \quad (8)$$

Determination of the likelihood $p\big(\{(t_i, \mathbf{x}_i)\} \mid \mathbf{x}_0, \boldsymbol{\theta}\big)$ is typically the computationally intensive step as it requires computing $\big\{\big(t_i, \mathbf{x}(t_i ; \mathbf{x}_0, \boldsymbol{\theta})\big)\big\}$ for parameters $\mathbf{x}_0, \boldsymbol{\theta}$ to compare to the data $\{(t_i, \mathbf{x}_i)\}$. Evaluating $\mathbf{x}(t_i ; \mathbf{x}_0, \boldsymbol{\theta})$ with conventional methods would require forward stepping from the initial conditions all the way to time t_i, and this process would have to be repeated for every different set of initial states and parameters $\mathbf{x}_0, \boldsymbol{\theta}$. The greater the desired precision of the posterior distribution for the parameters, the more often the differential equation has to be solved, which could be potentially millions of times. However, if a neural network solution bundle has been trained over X_0 and Θ containing the expected range of initial conditions and parameters, $\hat{\mathbf{x}}(t_i ; \mathbf{x}_0, \boldsymbol{\theta})$ can be calculated in constant time for any $t_i \in [t_0, t_f]$, and the entire set of points $\big\{\big(t_i, \hat{\mathbf{x}}(t_i ; \mathbf{x}_0, \boldsymbol{\theta})\big)\big\}$ can be computed in parallel. This allows for rapid likelihood evaluation and more efficient Bayesian inference. The training cost of the solution bundle can be further offset if it is used for many different sets of data. In effect, a trained neural network solution bundle for an often-used ODE could be shared amongst research groups, cutting back on the number of redundant calculations performed globally.
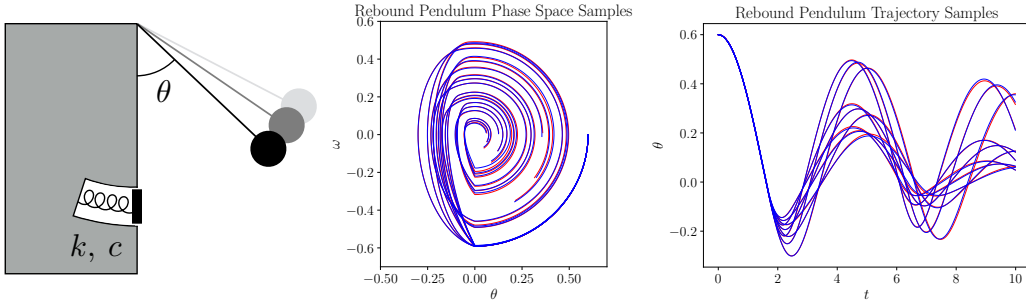
4.1 Rebound Pendulum

Figure 3: **Left:** Diagram of a rebound pendulum. **Center and Right:** A selection of solutions within the rebound pendulum solution bundle. The red reference curves are computed with Runge-Kutta and the trajectories from the solution bundle are overlayed in blue.
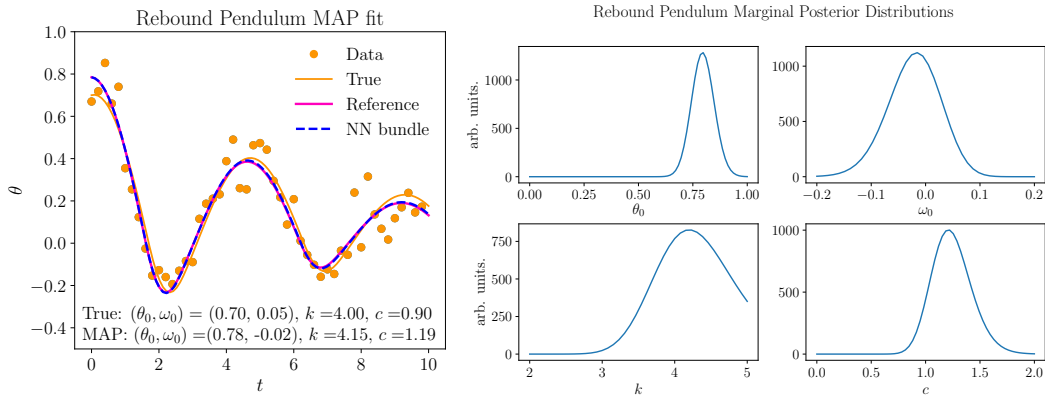

Figure 4: **Left:** Rebound pendulum fit corresponding to maximum a posteriori estimate. A Runge-Kutta reference curve using the MAP parameters is also plotted. **Right:** Unnormalized marginal posterior distributions of the unknown initial conditions and parameters.

A rebound pendulum consists of a simple pendulum that can collide with a damped spring at the bottom of its swing. A diagram of the setup is shown in Figure 3. The equations of motion for the

state vector $\mathbf{x} = (\theta, \omega)^\intercal$ of the pendulum are given by

$$\frac{\mathrm{d}\theta}{\mathrm{d}t} = \omega, \qquad \frac{\mathrm{d}\omega}{\mathrm{d}t} = -\frac{g}{\ell}\sin\theta + H(-\theta)\mathrm{ReLU}(-k\theta - c\omega), \tag{9}$$

where $\mathrm{ReLU}(x) = \max(x, 0)$, $H(x)$ is the Heaviside step function, g is the gravitational acceleration, ℓ is the length of the pendulum, k is the spring constant, and c is the damping coefficient. Note that here θ is the angle of the pendulum and not a parameter. A few example solutions are shown in Figure 3, where the convergence of the neural network approximation can be compared to reference solutions computed with Runge-Kutta.

Using the trained solution bundle, we can fit simulated angle measurements which have Gaussian error. The result of Bayesian inference of the initial conditions and parameters is shown in Figure 4, presenting both the MAP estimate and the marginal posterior distributions for a uniform prior.

5 Curriculum Learning: FitzHugh-Nagumo Model

The FitzHugh-Nagumo model[17, 18] is a relaxation oscillator which can be used as a simple model of a biological neuron. Its state is described by a membrane voltage v, and a recovery variable w, and the ODE has parameters a, b, τ, and I. The differential equation for the FitzHugh-Nagumo model is given by

$$\frac{\mathrm{d}v}{\mathrm{d}t} = v - \frac{v^3}{3} - w + I, \qquad \frac{\mathrm{d}w}{\mathrm{d}t} = \frac{1}{\tau}(v + a - bw). \tag{10}$$

The nullclines of Eq. (10), *i.e.* where $\mathrm{d}v\,/\,\mathrm{d}t = 0$ and $\mathrm{d}w\,/\,\mathrm{d}t = 0$, are plotted in black in Figure 5.

For this system, we found that it could be tricky to train the neural network solution bundle. When we simply sampled times uniformly from the time interval $[t_0, t_f]$, we found that the solution bundle could get "stuck" to the nullclines (as seen in Figure 5a).

(a) Without curriculum learning   (b) With curriculum learning
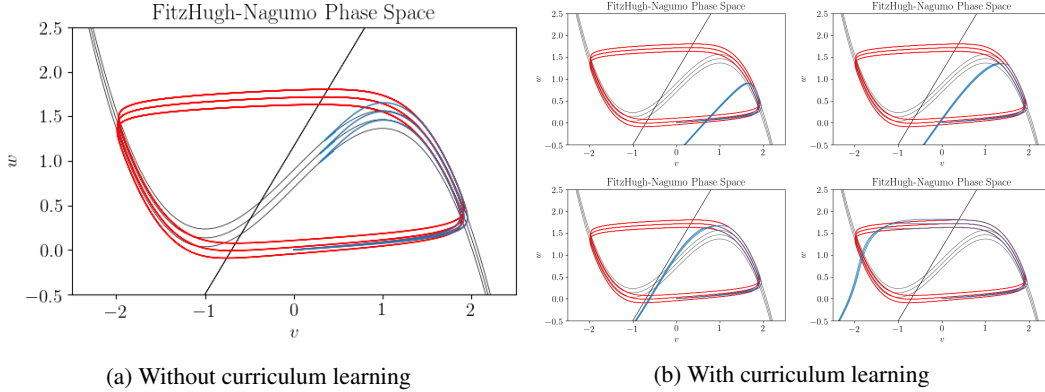
Figure 5: Reference trajectories for three values of I are shown in red, the corresponding nullclines for those values are shown in black, and the approximate solution bundle is shown in blue. Notice that without curriculum learning the neural network gets stuck on the cubic nullcline, while with curriculum learning this sticking point is safely passed during training.

This seems to happen because while the approximate solution bundle is dragged around phase space during early training, the tail end can encounter a nullcline by chance, where it is easier to satisfy the ODE. Even though an error at an earlier time has to be made in the solution bundle for the tail to be on the nullcline, if the later times have very low error due to the ease of predicting a constant value whilst on the nullcline, the network weights can become stuck in this local minimum.

This pitfall can be avoided by applying curriculum learning[19] to the training process. When training starts, we can restrict the time samples to come from $[t_0, t_m]$, where $t_m < t_f$ and m is the batch number. As training progresses and the batch number m is increased, we can make the task a bit more difficult by increasing t_m, requiring the network to learn a greater length of the solution bundle. The result of curriculum learning is shown in Figure 5b. The entire solution bundle is plotted in each frame, but in the loss calculation only times up to t_m are sampled.

6 Efficiency

To get a sense of the computational and memory efficiency of neural network solution bundles, we compare them to Runge-Kutta and lookup tables, respectively. The equations of motion of a harmonic oscillator are given by

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v, \qquad \frac{\mathrm{d}v}{\mathrm{d}t} = -\frac{k}{m}x, \tag{11}$$

and we choose $m = 1$ for simplicity. We trained four dense neural networks with two hidden layers, with widths 4, 8, 16, and 32, over initial conditions $(x_0, v_0) \in [-1, 1] \times [-1, 1]$, parameters $k \in [0.5, 2]$, and times $[-0.01, 2\pi]$. To evaluate the performance of these networks, we uniformly sampled points from $(x_0, v_0, k, t) \in [-1, 1] \times [-1, 1] \times [0.5, 2] \times [0, 2\pi]$ and computed the absolute error, which we define as $(|\hat{x} - x| + |\hat{v} - v|)/2$, relative to the exact solution (x, v) for each point. In Figure 6a, the absolute errors for these same points are determined with Runge-Kutta (RK4) and the Euler method for increasingly small step sizes, and hence more floating point operations (FLOPs). The solid lines give the average absolute error and the bands encompass 90% of the computed errors. While RK4 slightly outperforms the neural network approach, the operations in the evaluation of the dense neural network can be easily parallelized while RK4's steps cannot. As such, the simultaneous FLOPs per processor for the neural network can be kept nearly constant as the network complexity is increased, which can make the neural network solution bundle substantially faster in practice than a comparable RK4 approach, especially for an ODE with harder function evaluations than the multiplication in Eq. (11).

Figure 6b compares the neural network performances to simple uniformly-spaced lookup tables with the same number of divisions in x_0, v_0, k, and t. The neural network is a very compact representation of the solution bundle, achieving comparable performance to lookup tables taking up orders of magnitude more space. This is promising for the reusability of trained neural network solution bundles, as the smaller size makes it easier to share with others.

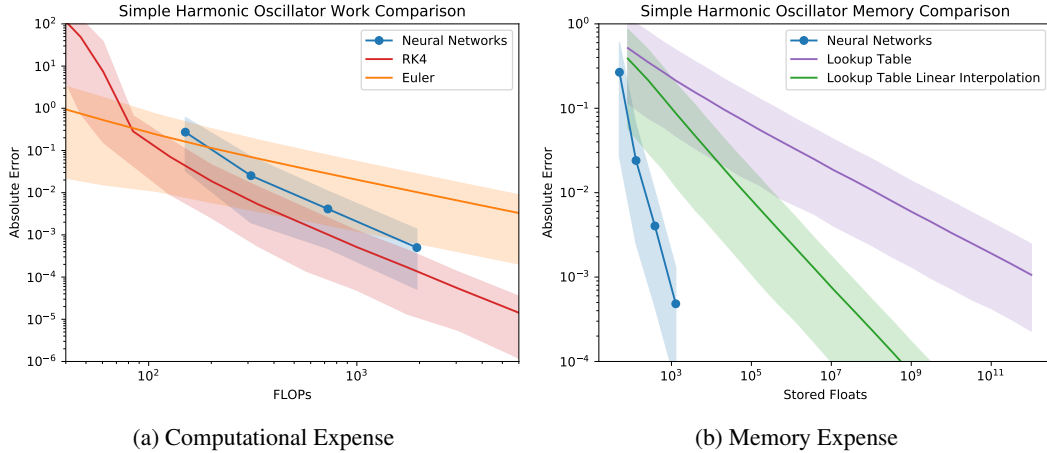(a) Computational Expense   (b) Memory Expense

Figure 6: Computational and memory efficiency comparisons of neural network solution bundles.

7 Conclusion

Backpropagation and the universal approximation theorem grants neural networks a unique ability to adapt and represent nearly any function. We extend the Lagaris method by introducing the concept of a neural network solution bundle, a group of approximate solutions to an ODE over a range of initial conditions and parameters. This allows for greater reuse of the trained network since it learns a variety of solutions. In addition, the solution bundle is differentiable in initial conditions and parameters, which can be useful for optimization tasks dependent on the value of the solution at given times. Other tasks that would require solving the differential equations repeatedly are also simplified, such as the propagation of uncertainty distributions across initial states, and for Bayesian inference in dynamical systems. The trained neural networks also have the potential to be used in

low-power limited-memory settings that require quickly solving differential equations, as in drones or IoT devices. While the number of calculations involved in the training of neural network solution bundles is substantially higher than for computing a single solution using conventional methods, the cost can eventually be recouped if enough individual solutions are required, especially if the trained network is shared with other users. In addition, future advances in neural network training, evaluation, and specialized hardware will directly benefit this method.

Broader Impact

The work presented here has very general applicability since differential equations are widely employed in the sciences and engineering. As such, many ethical aspects of this research are neutral and primarily depend on specific applications in the future and their impacts. However, if optimally-trained networks for common differential equations are shared and made freely available, this method has the potential to reduce the cost of redundant calculations, helping with efficient energy use. If this work is used in systems that could affect lives or the environment, critical calculations should be checked against other numerical approaches for safety.

Acknowledgments and Disclosure of Funding

References

[1] Christine Bernardi and Yvon Maday. Spectral methods. In *Handbook of Numerical Analysis*, volume 5, pages 209–485. Elsevier, 1997.

[2] M. W. M. G. Dissanayake and N. Phan-Thien. Neural-network-based approximations for solving partial differential equations. *Communications in Numerical Methods in Engineering*, 10(3):195–201, March 1994.

[3] I.E. Lagaris, A. Likas, and D.I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, September 1998.

[4] I.E. Lagaris, A.C. Likas, and D.G. Papageorgiou. Neural-network methods for boundary value problems with irregular boundaries. *IEEE Transactions on Neural Networks*, 11(5):1041–1049, September 2000. Conference Name: IEEE Transactions on Neural Networks.

[5] Kevin Stanley McFall and James Robert Mahan. Artificial Neural Network Method for Solution of Boundary Value Problems With Exact Satisfaction of Arbitrary Boundary Conditions. *IEEE Transactions on Neural Networks*, 20(8):1221–1233, August 2009. Conference Name: IEEE Transactions on Neural Networks.

[6] Modjtaba Baymani, Asghar Kerayechian, and Sohrab Effati. Artificial Neural Networks Approach for Solving Stokes Problem. *Applied Mathematics*, 01(04):288–292, 2010.

[7] Lu Lu, Xuhui Meng, Zhiping Mao, and George E. Karniadakis. DeepXDE: A deep learning library for solving differential equations. *arXiv:1907.04502 [physics, stat]*, February 2020. arXiv: 1907.04502.

[8] Alexander Koryagin, Roman Khudorozkov, and Sergey Tsimfer. PyDEns: a Python Framework for Solving Differential Equations with Neural Networks. *arXiv:1909.11544 [cs, stat]*, September 2019. arXiv: 1909.11544.

[9] Feiyu Chen, David Sondak, Pavlos Protopapas, Marios Mattheakis, Shuheng Liu, Devansh Agarwal, and Marco Di Giovanni. NeuroDiffEq: A Python package for solving differential equations with neural networks. *Journal of Open Source Software*, 5(46):1931, February 2020.

[10] Marios Mattheakis, David Sondak, Akshunna S. Dogra, and Pavlos Protopapas. Hamiltonian Neural Networks for solving differential equations. *arXiv:2001.11107 [physics]*, February 2020. arXiv: 2001.11107.

[11] Justin Sirignano and Konstantinos Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, December 2018.

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

[13] Sho Sonoda and Noboru Murata. Neural network with unbounded activation functions is universal approximator. *Applied and Computational Harmonic Analysis*, 43(2):233–268, September 2017.

[14] Victor Szebehely. *Theory of orbits*. Acad. Pr, New York, 1967. OCLC: 164462279.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, January 2017. arXiv: 1412.6980.

[16] Jacques-Louis Lions, Yvon Maday, and Gabriel Turinici. Résolution d'EDP par un schéma en temps «pararéel ». *Comptes Rendus de l'Académie des Sciences - Series I - Mathematics*, 332(7):661–668, April 2001.

[17] R. Fitzhugh. Impulses and Physiological States in Theoretical Models of Nerve Membrane. *Biophysical Journal*, 1(6):445–466, July 1961.

[18] J. Nagumo, S. Arimoto, and S. Yoshizawa. An Active Pulse Transmission Line Simulating Nerve Axon. *Proceedings of the IRE*, 50(10):2061–2070, October 1962.

[19] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*, pages 1–8, Montreal, Quebec, Canada, 2009. ACM Press.

[20] Endre Süli and D. F. Mayers. *An introduction to numerical analysis*. Cambridge University Press, Cambridge ; New York, 2003. OCLC: ocm50525488.

8 Appendix

A Proof of Global Error Bound

One appeal for using a decaying exponential weighting comes from comparison to one-step methods like Runge-Kutta. Given a differential equation of the form

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f(t, x), \tag{12}$$

a one-step method will generate a sequence through

$$x_{n+1} = x_n + h\Phi(t_n, x_n; h), \tag{13}$$

where the second term is the increment, which for example in the Euler method would be $h\Phi(t_n, x_n; h) = hf(t_n, x_n)$. The truncation (local) error is given by

$$\varepsilon_n = \frac{x(t_{n+1}) - x(t_n)}{h} - \Phi(t_n, x(t_n); h), \qquad h = t_{n+1} - t_n. \tag{14}$$

The global error in a one-step method, $\epsilon_N = x(t_N) - x_N$, which is due to the accumulated truncation error, is bounded by[20]

$$|\epsilon_N| \le \frac{\varepsilon}{L_\Phi}\left(e^{L_\Phi(t_N - t_0)} - 1\right) \tag{15}$$

where $\varepsilon = \max_{0 \le n \le N-1} |\varepsilon_n|$ and, Φ is Lipschitz continuous with Lipschitz constant L_ϕ, $|\Phi(t, u; h) - \Phi(t, v; h)| \le L_\Phi|u - v|$.

We can derive an analogue of this bound on global error for the neural network solution bundle. The local error of Eq. (14) should be compared to

$$\varepsilon(t_i \ ; \ \mathbf{x}_{0i}, \boldsymbol{\theta}_i) \equiv \mathbf{G}\left(\hat{\mathbf{x}}(t_i \ ; \ \mathbf{x}_{0i}, \boldsymbol{\theta}_i), \frac{\partial\hat{\mathbf{x}}(t_i \ ; \ \mathbf{x}_{0i}, \boldsymbol{\theta}_i)}{\partial t}, t_i \ ; \ \boldsymbol{\theta}_i\right), \tag{16}$$

which for a differential equation of the form in Eq. (12) would give

$$\varepsilon(t) = \frac{\mathrm{d}\hat{x}(t)}{\mathrm{d}t} - f(t, \hat{x}). \tag{17}$$

In our approach, Eq. (17) is effectively a truncation error for an infinitesimal timestep. The global error $\epsilon(t)$ is given by

$$\epsilon(t) = \hat{x}(t) - x(t), \tag{18}$$

which we can substitute into Eq. (17) to obtain

$$\frac{\mathrm{d}\epsilon}{\mathrm{d}t} = \varepsilon(t) - \frac{\mathrm{d}x}{\mathrm{d}t} + f\big(t, x(t) + \epsilon(t)\big). \tag{19}$$

Assuming f is a Lipschitz-continuous function, $\left|f\big(t, x(t) + \epsilon(t)\big) - f\big(t, x(t)\big)\right| \le L_f|\epsilon(t)|$, so the absolute value of the right-hand side can be written

$$\left|\varepsilon(t) - \frac{\mathrm{d}x}{\mathrm{d}t} + f\big(t, x(t) + \epsilon(t)\big)\right| = \left|\varepsilon(t) + f\big(t, x(t) + \epsilon(t)\big) - f\big(t, x(t)\big)\right| \tag{20}$$

$$\le |\varepsilon(t)| + L_f|\epsilon(t)| \tag{21}$$

$$\le \varepsilon_{t'} + L_f|\epsilon(t)|, \tag{22}$$

where we have made use of the differential equation $\mathrm{d}x / \mathrm{d}t = f\big(t, x(t)\big)$, the triangle inequality, and $\varepsilon_{t'} = \max_{t_0 \le t \le t'} |\varepsilon(t)|$. Since $\epsilon(t_0) = 0$, to find a bound on $|\epsilon(t)|$ we can consider the solution to the ODE for upper bound $E(t) \ge |\epsilon(t)| \ge 0$,

$$\frac{\mathrm{d}E}{\mathrm{d}t} = \varepsilon_{t'} + L_f E(t) \tag{23}$$

where $E(t_0) = 0$, which is $E(t) = \frac{\varepsilon_{t'}}{L_f}(\exp[L_f(t - t_0)] - 1)$. Thus,

$$|\epsilon(t)| \le \frac{\varepsilon_{t'}}{L_f}\left(e^{L_f(t - t_0)} - 1\right), \tag{24}$$

which can be compared to the global error bound of the discrete case, Eq. (15).

B Propagating a Probability Distribution

A neural network solution bundle provides a mapping from initial conditions to the state at later times. This can be useful for time-evolving a distribution over initial conditions to obtain a probability distribution over states at later time t. Given a probability density over initial states $p_0(\mathbf{x}_0)$, we note that the solution bundle $\mathbf{x}(t \, ; \, \mathbf{x}_0)$ at time t describes a coordinate transformation from $\mathbf{x}_0$, transforming the coordinates $\mathbf{x}_0$ to $\mathbf{x}_t$. If this transformation is invertible and differentiable from the subset of initial state space X_0 to the final state space X_t, we can write out the probability density of later states,

$$p_t(\mathbf{x}_t) = p_0\big(\mathbf{f}^{-1}(\mathbf{x}_t)\big)|\mathbf{J}_{\mathbf{f}^{-1}}|, \tag{25}$$

where $\mathbf{f}^{-1}(\mathbf{x}_t)$ is the inverse of $\mathbf{f}(\mathbf{x}_0) \equiv \mathbf{x}(t \, ; \, \mathbf{x}_0)$, and $\mathbf{J}_{\mathbf{f}^{-1}} = \frac{\partial \mathbf{x}_0}{\partial \mathbf{x}_t}$ is the Jacobian of $\mathbf{f}^{-1}$. The opposite task, *i.e* where a probability distribution over later states $p_t(\mathbf{x}_t)$ is known and the distribution over initial states is desired, takes on an even more convenient form where the neural network solution bundle does not have to be inverted:

$$p_0(\mathbf{x}_0) = p_t(\mathbf{f}(\mathbf{x}_0))|\mathbf{J}_{\mathbf{f}}|, \tag{26}$$

where $\mathbf{J}_{\mathbf{f}} = \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_0}$ is the Jacobian of $\mathbf{f}$, which can be calculated exactly using automatic differentiation of the solution bundle. This gives a closed analytic form for the desired probability density. Note that if the ODE is time-reversible, Eq. (25) can be converted to the easier form Eq. (26) by simply treating the target space as the input space and training the solution bundle on the time-reversed equations of motion.

In the case of an energy-conserving Hamiltonian dynamical system where all the state variables are canonical coordinates, Liouville's theorem guarantees that $|\mathbf{J}_{\mathbf{f}}| = |\mathbf{J}_{\mathbf{f}^{-1}}|$, resulting in particularly simple relationships for the probability densities at different times.

B.1 Planar Circular Restricted Three-Body Problem

If we have two measurements of the position of an asteroid at two different times, along with some uncertainty,

$$(t_0, x_0, y_0) = (0.00, 1.0510 \pm 0.0003, 0.1000 \pm 0.0003) \tag{27a}$$
$$(t_1, x_1, y_1) = (0.05, 1.0276 \pm 0.0003, 0.0878 \pm 0.0003) \tag{27b}$$

we can compute the probability distribution of future positions with the neural network solution bundle. Let $p\big(\mathbf{r}(t) = (x, y)^\intercal \mid \{\mathbf{r}_1, \mathbf{r}_0\}\big)$ be the probability density of the position being $(x, y)^\intercal$ at time t, given the position measurements $\mathbf{r}_0$ and $\mathbf{r}_1$. By marginalizing over the final velocities, we obtain

$$p\big(\mathbf{r}(t) = (x, y)^\intercal \mid \{\mathbf{r}_1, \mathbf{r}_0\}\big) = \iint p\big(\mathbf{q}(t) = (x, y, u, v)^\intercal \mid \{\mathbf{r}_1, \mathbf{r}_0\}\big) \, \mathrm{d}u \, \mathrm{d}v \, . \tag{28}$$

To compute the integrand, we can use Bayes' theorem,

$$p\big(\mathbf{q}(t) = (x, y, u, v)^\intercal \mid \{\mathbf{r}_1, \mathbf{r}_0\}\big) = \frac{p\big(\{\mathbf{r}_1, \mathbf{r}_0\} \mid \mathbf{q}(t) = (x, y, u, v)^\intercal\big)p\big(\mathbf{q}(t) = (x, y, u, v)^\intercal\big)}{p\big(\{\mathbf{r}_1, \mathbf{r}_0\}\big)}$$
$$\propto p\big(\mathbf{r}_1 \mid \mathbf{q}(t) = (x, y, u, v)^\intercal\big)p\big(\mathbf{r}_0 \mid \mathbf{q}(t) = (x, y, u, v)^\intercal\big), \tag{29}$$

where in the last step we have assumed a uniform prior, and that the errors in the two position measurements $\mathbf{r}_0$ and $\mathbf{r}_1$ are independent.

Iterating over a uniform grid of initial positions and velocities, evaluating the solution at time t_1 and t, weighting the samples with the probability densities given by Eq. (29), and forming a weighted histogram of positions, we have the approximate distribution of the asteroid's location at time t. Figure 7 shows the position distribution at various final times t. When uniform sampling becomes infeasible due to high dimensionality of the inputs, it is straightforward to use MCMC instead.
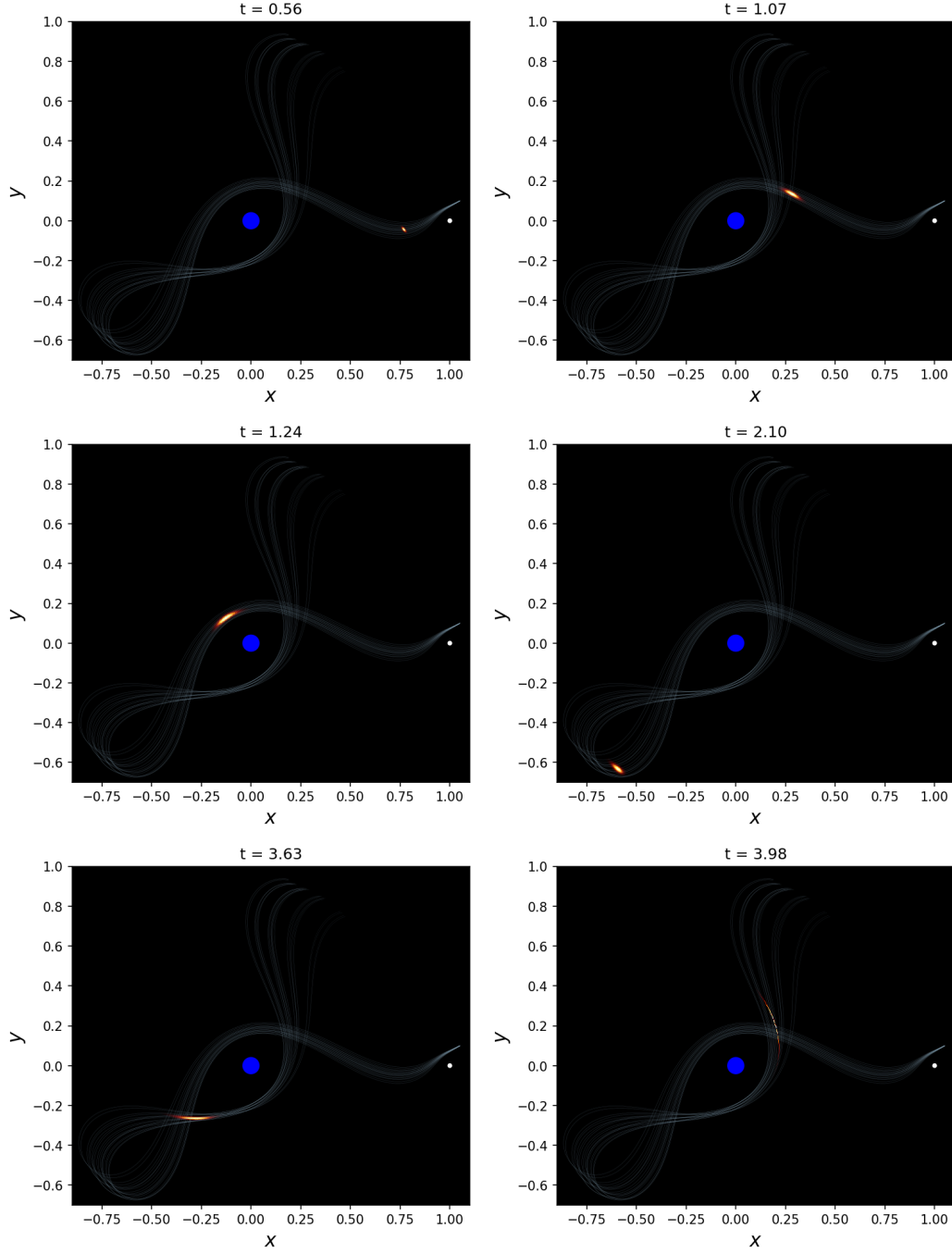
Figure 7: Probability distribution $p(\mathbf{x}(t) = (x, y)^\mathsf{T})$ at various times. A few trajectories in the bundle are shown in white, and the distribution is shown as a heatmap.

C Computational Details

All calculations were performed with PyTorch 1.3.1 on a single node equipped with a GPU:

- Model name: Intel Xeon Gold 6126 CPU @ 2.60GHz
- CPUs: 24
- RAM: 191898 MB

- GPU: Nvidia Tesla V100-PCIE-16GB

C.1 Planar Circular Restricted Three-Body Problem

ODE:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = u \,, \qquad \frac{\mathrm{d}u}{\mathrm{d}t} = x - \mu + 2v - \left[\frac{\mu(x-1)}{\left((x-1)^2 + y^2\right)^{3/2}} + \frac{(1-\mu)x}{(x^2+y^2)^{3/2}} \right], \tag{30}$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = v \,, \qquad \frac{\mathrm{d}v}{\mathrm{d}t} = y - 2u - \left[\frac{\mu y}{\left((x-1)^2 + y^2\right)^{3/2}} + \frac{(1-\mu)y}{(x^2+y^2)^{3/2}} \right]. \tag{31}$$

The above equation is in the co-rotating frame in the plane of the two heavy bodies (Earth and Moon) about their barycenter. The position of the third body of negligible mass is given by (x, y), with velocity (u, v). The parameter μ is the ratio of the Moon mass to the Earth plus Moon combined mass.

Network Architecture		
Input layer	5 neurons	tanh
Hidden layers	8 dense layers of 128 neurons each	tanh
Output layer	4 neurons	linear

Table 1: Planar circular restricted three-body problem network architecture.

Quantity	Value	
Weighting function	$b(t) = \exp(-2t)$	
Initial condition ranges	$(x_0, y_0, u_0, v_0) \in [1.05, 1.052] \times [0.099, 0.101] \times [-0.5, -0.4] \times [-0.3, -0.2]$	
ODE parameter	$\mu = 0.01$	
Time range	$[-0.01, 5]$	
Optimizer	Adam	
Batch size	10,000	
Learning rate	batch 0 to 3,999,999	$\eta = 0.001$, ReduceLROnPlateau
		factor=0.5, patience=200000, threshold = 0.5,
		threshold_mode = 'rel', cooldown = 0,
		min_lr = 1e-6, eps = 1e-8
	batch 4,000,000 to 7,999,999	$\eta = 0.000001$
	batch 8,000,000 to 35,999,999	$\eta = 0.00001$
Training rate	56 batches/sec	
Training time	178.6 hours	

Table 2: Planar circular restricted three-body problem training hyperparameters and other details.

We found that reducing the learning rate during training greatly helped in decreasing the loss for this system. In Figure 8 every time the learning rate was halved, the loss immediately decreased, forming downward steps. However, the learning rate was made too small through this process, and the network approached a suboptimal minimum by batch 8×10^6. Curiously, after increasing the learning rate, despite temporarily increasing the loss, the solution bundle immediately started converging better to the true trajectories determined with Runge-Kutta. This is due to the loss assessing the solution bundle's satisfaction of the ODE instead of its accuracy compared to the true solution. As seen in Figure 8, if small errors are made at early times in the solution bundle such that the neural network has an easier task at later times, the earlier errors might not be corrected. This is seen by the trajectories at later times not fanning out.
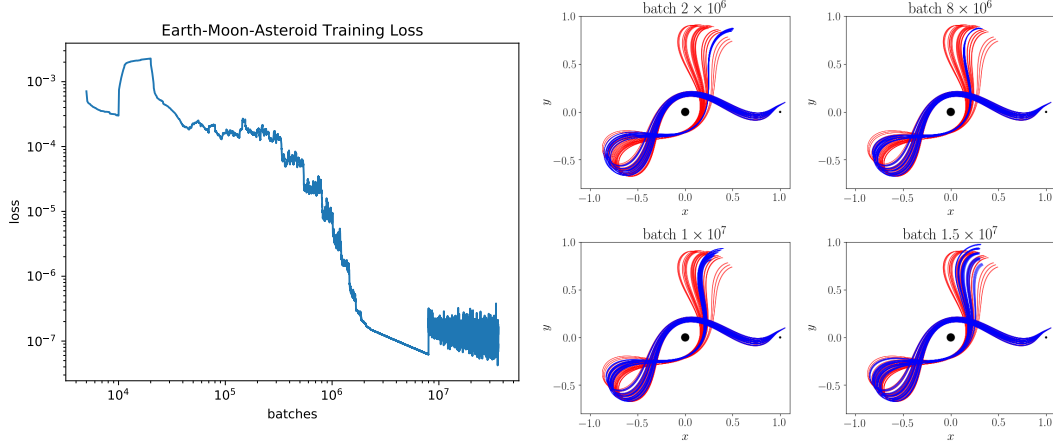
14

Figure 8: **Left:** Loss during training for the planar circular restricted three-body problem, smoothed with a moving average of window size 10,000. **Right:** Solution bundle after select batches during training. These are presented to demonstrate why we increased the learning rate starting at batch 8×10^6, causing the temporary increase in the loss. At batch 8×10^6 notice that the tail of the solution bundle is not fanning out, and it appears that the neural network is trapped in a local minimum. Increasing the learning rate helped the network get out, and the solution bundle once again began approaching the reference trajectories in red.

C.2 Rebound Pendulum

ODE:

$$\frac{\mathrm{d}\theta}{\mathrm{d}t} = \omega, \qquad \frac{\mathrm{d}\omega}{\mathrm{d}t} = -\frac{g}{\ell}\sin\theta + H(-\theta)\mathrm{ReLU}\left(-\frac{k\ell}{m}\theta - c\omega\right), \tag{32}$$

where $H(x)$ is the Heaviside step function, and $\mathrm{ReLU}(x)$ is the rectifier function. The position of the pendulum is described by its angle θ, and its angular velocity is ω. The parameters in the equation are: gravitational acceleration g, pendulum length ℓ, spring constant k, damping coefficient c, and mass m. For simplicity we set g, ℓ, and m to unity. The second term in the second differential equation describes the interaction of the spring with the pendulum, capturing the fact that the spring only interacts with the pendulum for negative angles and that it can only push the pendulum away.

Network Architecture		
Input layer	5 neurons	tanh
Hidden layers	8 dense layers of 128 neurons each	tanh
Output layer	2 neurons	linear

Table 3: Rebound pendulum network architecture.

Quantity	Value	
Weighting function	$b(t) = \exp(-0.5t)$	
Initial condition ranges	$(\theta_0, \omega_0) \in [0.0, 1.0] \times [-0.2, 0.2]$	
ODE parameter ranges	$(k, c) \in [2.0, 5.0] \times [0.0, 2.0]$	
Time range	$[-0.01, 10]$	
Optimizer	Adam	
Batch size	10,000	
Learning rate	batch 0 to 1,999,999	$\eta = 0.001$, ReduceLROnPlateau factor=0.5, patience=1500000, threshold = 0.5, threshold_mode = 'rel', cooldown = 0, min_lr = 0, eps = 1e-8
	batch 0 to 1,999,999	$\eta = 0.00001$, ReduceLROnPlateau factor=0.5, patience=2000000, threshold = 0.5, threshold_mode = 'rel', cooldown = 0, min_lr = 0, eps = 1e-8
Training rate	98 batches/sec	
Training time	11.37 hours	

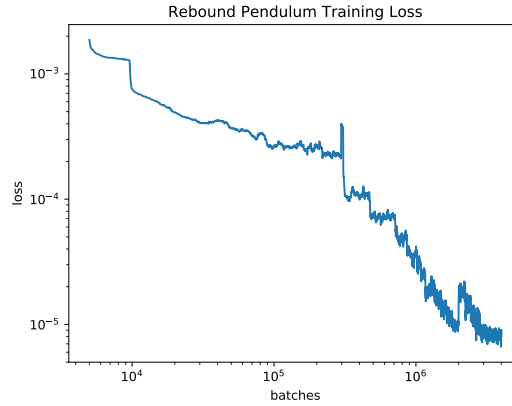Table 4: Rebound pendulum training hyperparameters and other details.

Figure 9: Rebound pendulum loss during training, smoothed with a moving average of window size 10,000.

C.3 FitzHugh-Nagumo Model

ODE:

$$\frac{\mathrm{d}v}{\mathrm{d}t} = v - \frac{v^3}{3} - w + I, \qquad \frac{\mathrm{d}w}{\mathrm{d}t} = \frac{1}{\tau}(v + a - bw). \tag{33}$$

The two state variables are v and w which are membrane voltage and recovery, respectively. The parameters of the system are a, b, which control the shape of the trajectory, τ, which sets the timescale, and I, the external stimulus. The nullclines of the ODE are

$$w = v - \frac{v^3}{3} + I, \qquad\qquad \Rightarrow \frac{\mathrm{d}v}{\mathrm{d}t} = 0 \tag{34}$$

$$w = \frac{v + a}{b}, \qquad\qquad \Rightarrow \frac{\mathrm{d}w}{\mathrm{d}t} = 0 \tag{35}$$

which are helpful for understanding the direction reversals in each state variable of the trajectory in phase space.

Network Architecture		
Input layer	7 neurons	tanh
Hidden layers	4 dense layers of 128 neurons each	tanh
Output layer	2 neurons	linear

Table 5: FitzHugh-Nagumo (without curriculum learning) network architecture.

Quantity	Value	
Weighting function	$b(t) = \exp\left[-\lambda_m t\right]$	
	$\lambda_m = \exp\left[-\ln\left(100\right)\frac{m}{M}\right]$	where m is the batch number, M is total batches
Initial condition ranges	$(v, w) \in [-0.2, 0.2] \times [-0.2, 0.2]$	
ODE parameter ranges	$(a, b, \tau, I) \in [0.6, 0.9] \times [0.6, 0.9] \times [10., 14.] \times [0.6, 1.0]$	
Time range	$[-0.10, 100]$	
Optimizer	Adam	
Batch size	10,000	
Learning rate	batch 0 to 499,999	$\eta = 0.001$
Training rate	151 batches/sec	
Training time	55 minutes	

Table 6: FitzHugh-Nagumo (without curriculum learning) training hyperparameters and other details.

Network Architecture		
Input layer	7 neurons	tanh
Hidden layers	8 dense layers of 121 neurons each	tanh
	skip connections from input concatenated to output of every hidden layer	linear
Output layer	2 neurons	linear

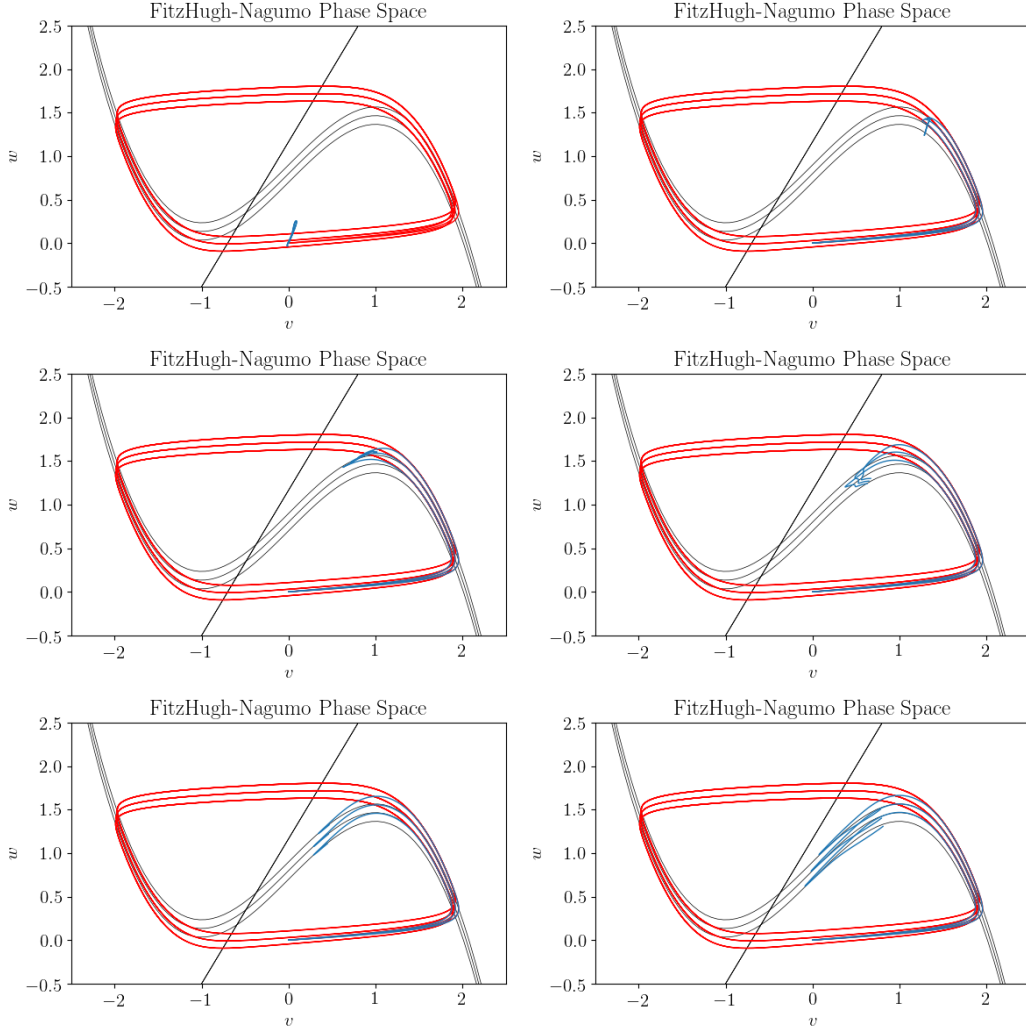Table 7: FitzHugh-Nagumo (curriculum learning) network architecture.

Figure 10: Training FitzHugh-Nagumo solution bundles without curriculum learning. The trajectories bind to the nullclines early on and the solution bundle does not converge properly.

Quantity	Value	
Dynamic end time	$t_m = \frac{100}{\ln(11)} \ln\left(10\frac{m}{M} + 1\right)$	where m is the batch number, M is total batches
Weighting function	$b(t) = \exp\left[-\lambda_m t\right]$	
	$\lambda_m = \frac{4}{t_m + 5}$	
Initial condition ranges	$(v, w) \in [-0.1, 0.1] \times [-0.1, 0.1]$	
ODE parameter ranges	$(a, b, \tau, I) \in [0.6, 0.8] \times [0.5, 0.7] \times [11., 14.] \times [0.7, 0.9]$	
Time range	$[-0.10, 100]$	
Optimizer	Adam	
Batch size	10,000	
Learning rate	batch 0 to 9,999,999	$\eta = 0.0001$
Training rate	80 batches/sec	
Training time	34.8 hours	

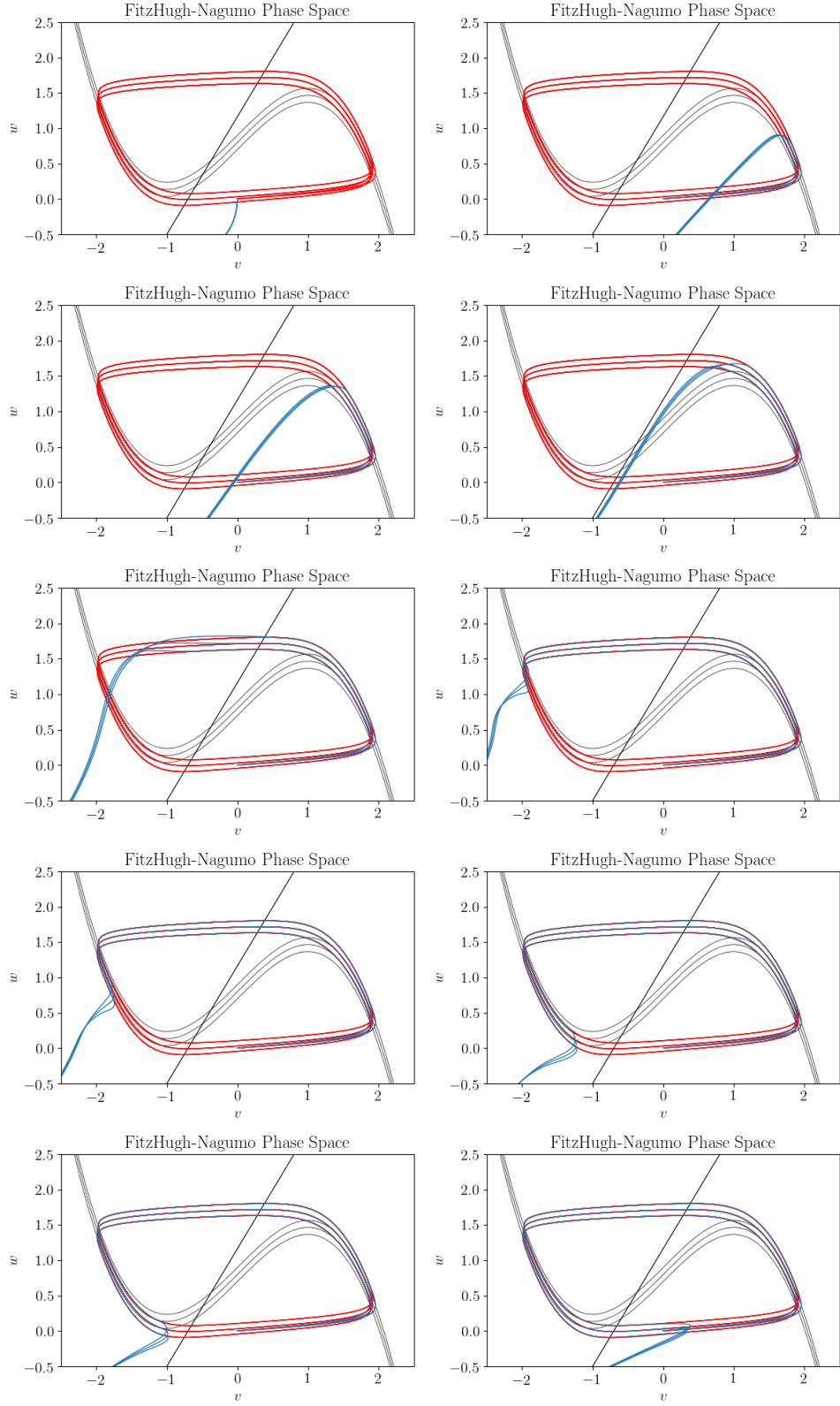Table 8: FitzHugh-Nagumo (curriculum learning) training hyperparameters and other details.

Figure 11: Training FitzHugh-Nagumo solution bundles with curriculum learning. Unlike Figure 10, the nullclines are safely passed.

C.4 Efficiency Tests: Simple Harmonic Oscillator

ODE:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v, \qquad \frac{\mathrm{d}v}{\mathrm{d}t} = -\frac{k}{m}x, \tag{36}$$

where the position of the harmonic oscillator is given by x, its velocity is v. The parameters of the system, spring constant k and mass m, are both fixed at unity.

Network	Architecture		
1	Input layer	3 neurons	tanh
	Hidden layers	2 dense layers of 4 neurons each	tanh
	Output layer	2 neurons	linear
2	Input layer	3 neurons	tanh
	Hidden layers	2 dense layers of 8 neurons each	tanh
	Output layer	2 neurons	linear
3	Input layer	3 neurons	tanh
	Hidden layers	2 dense layers of 16 neurons each	tanh
	Output layer	2 neurons	linear
4	Input layer	3 neurons	tanh
	Hidden layers	2 dense layers of 32 neurons each	tanh
	Output layer	2 neurons	linear

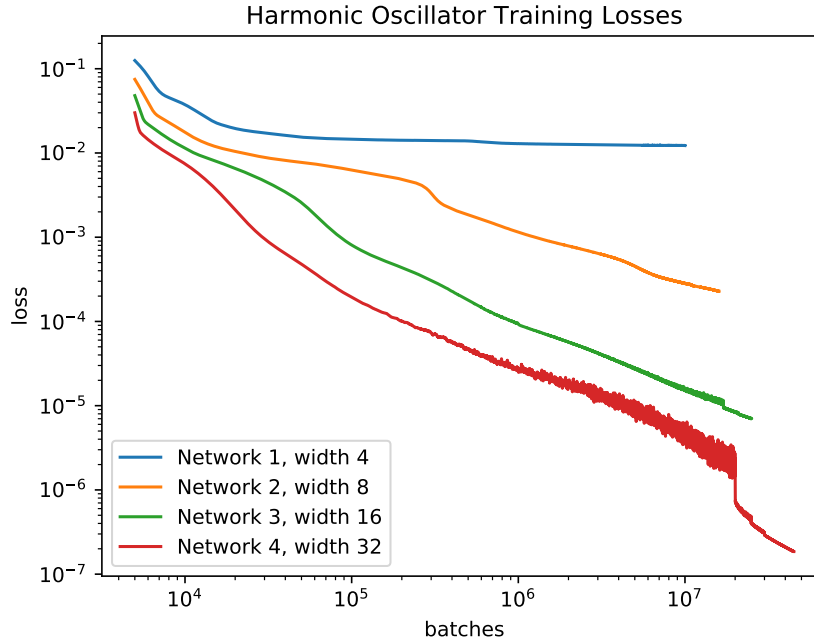Table 9: Simple harmonic oscillator network architectures.

Figure 12: Training losses of the four harmonic oscillator solution bundles, smoothed with a moving average with window size 10,000.

20

Network	Quantity	Value	
1	Weighting function	$b(t) = 1$	
	Initial condition ranges	$(x, v) \in [-1.0, 1.0] \times [-1.0, 1.0]$	
	ODE parameter ranges	$k \in [0.5, 2.0]$	
	Time range	$[-0.01, 2\pi]$	
	Optimizer	Adam	
	Batch size	100,000	
	Learning rate	batch 0 to 4,999,999	$\eta = 0.0001$
	Learning rate	batch 5,000,000 to 9,999,999	$\eta = 0.00002$
	Training rate	179 batches/sec	
	Training time	15.5 hours	
2	Weighting function	$b(t) = 1$	
	Initial condition ranges	$(x, v) \in [-1.0, 1.0] \times [-1.0, 1.0]$	
	ODE parameter ranges	$k \in [0.5, 2.0]$	
	Time range	$[-0.01, 2\pi]$	
	Optimizer	Adam	
	Batch size	100,000	
	Learning rate	batch 0 to 10,999,999	$\eta = 0.0001$
	Learning rate	batch 11,000,000 to 15,999,999	$\eta = 0.00002$
	Training rate	168 batches/sec	
	Training time	26.5 hours	
3	Weighting function	$b(t) = 1$	
	Initial condition ranges	$(x, v) \in [-1.0, 1.0] \times [-1.0, 1.0]$	
	ODE parameter ranges	$k \in [0.5, 2.0]$	
	Time range	$[-0.01, 2\pi]$	
	Optimizer	Adam	
	Batch size	100,000	
	Learning rate	batch 0 to 16,999,999	$\eta = 0.00005$
	Learning rate	batch 17,000,000 to 20,999,999	$\eta = 0.00002$
	Learning rate	batch 21,000,000 to 24,999,999	$\eta = 0.00001$
	Training rate	152 batches/sec	
	Training time	45.7 hours	
4	Weighting function	$b(t) = 1$	
	Initial condition ranges	$(x, v) \in [-1.0, 1.0] \times [-1.0, 1.0]$	
	ODE parameter ranges	$k \in [0.5, 2.0]$	
	Time range	$[-0.01, 2\pi]$	
	Optimizer	Adam	
	Batch size	100,000	
	Learning rate	batch 0 to 19,999,999	$\eta = 0.0001$
	Learning rate	batch 20,000,000 to 24,999,999	$\eta = 0.00002$
	Learning rate	batch 25,000,000 to 29,999,999	$\eta = 0.00001$
	Learning rate	batch 30,000,000 to 44,999,999	$\eta = 0.000002$
	Training rate	126 batches/sec	
	Training time	99.2 hours	

Table 10: Simple harmonic oscillator networks' training hyperparameters and other details.