Unsupervised Reservoir Computing for Solving Ordinary Differential Equations

Marios Mattheakis[*], **Hayden Joy, Pavlos Protopapas**
John A. Paulson School of Engineering and Applied Sciences, Harvard University
Cambridge, Massachusetts 02138, United States
mariosmat@seas.harvard.edu, hnjoy@mac.com, pavlos@seas.harvard.edu

Abstract

There is a wave of interest in using unsupervised neural networks for solving differential equations. The existing methods are based on feed-forward networks, while recurrent neural network differential equation solvers have not yet been reported. We introduce an unsupervised reservoir computing (RC), an echo-state recurrent neural network capable of discovering approximate solutions that satisfy ordinary differential equations (ODEs). We suggest an approach to calculate time derivatives of recurrent neural network outputs without using backpropagation. The internal weights of an RC are fixed, while only a linear output layer is trained, yielding efficient training. However, RC performance strongly depends on finding the optimal hyper-parameters, which is a computationally expensive process. We use Bayesian optimization to efficiently discover optimal sets in a high-dimensional hyper-parameter space and numerically show that one set is robust and can be used to solve an ODE for different initial conditions and time ranges. A closed-form formula for the optimal output weights is derived to solve first order linear equations in a backpropagation-free learning process. We extend the RC approach by solving nonlinear system of ODEs using a hybrid optimization method consisting of gradient descent and Bayesian optimization. Evaluation of linear and nonlinear systems of equations demonstrates the efficiency of the RC ODE solver.

1 Introduction

Neural networks (NNs) have been widely applied recently to study various kinds of differential equations. Physics-informed NNs can be trained on data to learn nonlinear differential operators [1], discover differential equations [2, 3], and find approximate solutions for those equations [4]. These data-driven supervised networks have been applied to a variety of real-world problems such as learning the dynamics of mechanical systems [5–7] and designing meta-materials for nano-photonics [8]. Unsupervised NNs have been used to solve a variety of differential equations such as ordinary differential equations (ODEs) [9–12], partial differential equations [12–15], and eigenvalue problems [16]; these networks do not use any labeled data. Semi-supervised models have been applied to learn general solutions of differential equations and extract solutions which best fit given data [17]. Unsupervised NN solvers present exceptional advantages over traditional integrators: they suffer less from the "curse of dimensionality" in solving high-dimensional partial differential equations [13, 18], the numerical solutions are obtained in a closed and differentiable form [9], numerical errors are not accumulated in the solutions [12], initial and boundary conditions are identically satisfied [9, 16], and the solutions can be inverted [8, 17, 19]. Despite the success of the aforementioned models, all approaches are based on feed-forward NNs, while, to the best of our knowledge, recurrent neural networks (RNNs) for solving differential equations in an unsupervised fashion have not been reported

[*] https://scholar.harvard.edu/marios_matthaiakis/home

Preprint. Under review.

yet. In this study, we fill the gap by introducing an unsupervised RNN, in the context of reservoir computing (RC) [20], which is able to discover approximate solutions to systems of ODEs.

RC is an echo state RNN where the internal parameters (weights and biases) are fixed, while only a linear output layer needs to be trained yielding fast and computationally efficient training [20–22]. Fixing the internal weights eliminates gradient exploding/vanishing problems during the training [20]. RC has been widely used for studying dynamical systems such as weather forecasting [23], predicting chaotic [21, 24] and irregular behavior [22], and classifying time series [25]. Moreover, the RC architecture has been adopted to build physical hardware NNs for neuromorphic computing [26–31]. Neuromorphic devices are currently being developed for intelligent and energy-efficient devices providing extremely fast real-time computing with very low energy cost. This study suggests an avenue to design neuromorphic differential equation solvers.

The proposed RC solver is an extension to NN differential equation solvers and consequently, acquires all the benefits that network solvers have over numerical integrators. Moreover, RNNs perform better than feed-forward NNs on sequential data. Considering that the solutions of ODEs are time series and, therefore, similar to sequential data, we expect RNNs to generalize better than feed-forward NNs. The internal weights of an RC are fixed and randomly initialized from distributions that have certain statistical properties which yield the echo-state property [20, 21]. Training an RC is efficient, however, finding the optimal hyper-parameters that determine the weight distributions and the network architecture is challenging. RC performance strongly depends on these hyper-parameters and, therefore, finding good hyper-parameters is crucial. The RC has many hyper-parameters making this a computationally expensive process. Conventionally, simple grid-search in the hyper-parameter space is used [21], however, this method is prohibitively expensive for more than three hyper-parameters. Tuning hyper-parameters by using Bayesian Optimization (BO) is an active field of research [32–36]. In this study, we use TURBO-1, a BO method introduced in [33]. The strength of this method is discussed in the supplementary material (SM). We find that a single set of hyper-parameters is robust and can be used to solve ODEs for different initial conditions and time ranges.

Main contributions: First, in the continuous time limit we suggest an approximation to calculate time derivatives of the outputs of an RNN without using backpropagation, which is a substantial ingredient in ODE-solver neural networks. Second, we introduce an unsupervised RC and show that it is capable of solving ODEs. Third, we derive a closed-form formula for the RC trainable parameters for solving first order non-autonomous linear ODEs in a backpropagation-free learning method. Fourth, we develop a hybrid method consisting of gradient descent and BO which is able to find optimal hyper-parameters and weights of an RC for solving systems of nonlinear ODEs. For this study, we use the `rcTorch` library, an RC framework written in pytorch with embedded BO that utilizes BoTorch [36]. A public open-source github repository is available at https://github.com/blindedjoy/RcTorch.

2 Background: Neural network differential equations solvers

Several software libraries of NN differential equations solvers have been recently developed, including NeuroDiffEq [15], DeepXDE [37], and SimNet [38], indicating that developing NN solvers is an active area of research [39]. All of these libraries are based on feed-forward NNs and developed with the pytorch or tensorflow software platforms, where the automatic differentiation mechanism is employed to compute analytical derivatives used to define the loss function. In this context NN solvers are unsupervised learning methods. Since we want to solve differential equations, we do not know the corresponding target solutions and thus, we lack labeled or ground truth data. The only accessible information is the differential equation and the associated initial or boundary conditions. The loss function solely depends on the network predictions including the differential equation, while the initial/boundary conditions are embedded in the structure of the network and are thus identically satisfied. Neural network solvers are able to solve ordinary and partial differential equations of an arbitrary order. We nevertheless focus on ODEs in this study — particularly on systems of first order ODEs since higher order ODEs can be decomposed into systems of first order ODEs. We review here the approach of developing NNs for solving a first order ODEs subjected to certain initial conditions (ICs). Consider a general ODE of the form

$$D\left(t, \psi, \dot{\psi}\right) - f(t) = 0, \tag{1}$$

where t is the independent variable time, and $\psi(t)$ is the dependent variable subjected to a certain IC $\psi(0) = \psi_0$. In Eq. (1), D is an arbitrary function of $\psi(t)$ and its first time derivative $\dot{\psi}$, and $f(t)$ is a forcing function of t. Equation (1) describes a non-autonomous system since it explicitly dependents on time. Considering known D and f functions, we are seeking $\psi(t)$ that solves Eq. (1) and satisfies a given ψ_0. Specifically, the goal is to construct a numerical solution y which approximates the unknown ground truth solution ψ. To achieve that, we employ a NN that takes an input $t = (t_1, t_2, \ldots, t_K)$ and returns an output $N(t, p)$; K indicates the total number of the input data points, and p denotes the trainable parameters of the network. An efficient way to impose ICs was introduced in Ref. [9] and suggests using of a parametric solution $y(t)$ of the form:

$$y(t) = \psi_0 + g(t)N(t, p) \tag{2}$$

where $g(t)$ can be any arbitrary function of t with the constraint $g(0) = 0$. The parametrization of Eq. (2) generates NN solutions that identically satisfy the ICs [9, 10, 12], namely $y(0) = \psi_0$. Having the parametric solution of Eq. (2), which is a function of N, solving the ODE of Eq. (1) is reduced to an optimization problem of the form

$$\arg\min_{p} \left(\sum_{n=0}^{K} \Big(D\left(t_n, y_n, \dot{y}_n\right) - f(t_n) \Big)^2 \right), \tag{3}$$

where $y_n = y(t_n)$. The sum in Eq. (3) defines a loss function whose minimization yields p that constructs a neural solution y which approximately solves the ODE (1). It is worth noting that $y(t)$ can approximate $\psi(t)$ with arbitrary small error due to the universal approximation theorem of NNs [40]. Furthermore, regularization terms can be used to penalize large weights or to impose physical principles like energy conservation [12]. Subsequently, the total loss function is expressed as:

$$\begin{aligned} L &= L_{\text{ODE}} + L_{\text{reg}} \\ &= \sum_{n=0}^{K} \Big(D\left(t_n, y_n, \dot{y}_n\right) - f(t_n) \Big)^2 + L_{\text{reg}}. \end{aligned} \tag{4}$$

The NN solution y of Eq. (2) is a closed-form solution, meaning that it can be evaluated at every time point, differentiated, and inverted. These are unique properties of NN solvers that are not shared by standard integrators. Despite the advantages shown by feed-forward NN solvers, an RNN solver is missing from the literature. In this work, we present a novel RNN solver, specifically an echo-state RC, capable of solving ODEs in a given training and IC range.

3 Continuous-time recurrent neural networks

Learning from sequential data is a challenging task for machine learning because of the underlying time correlation. RNNs share parameters across the hidden layers giving them an intrinsic memory and subsequently, they are well suited to handle sequential data. Standard RNNs require discrete input data at discrete time points. In the continuous-time limit, when the discrete time points are close to each other, the dynamics of the hidden RNN layers can be approximated by continuously defined dynamics through ODEs [41–43]. This approximation has been adopted by residual networks [41] and continuous depth models [42, 43]. This is a core idea in the present study because it allows, in a backpropagation-free process, the calculation of the time derivatives of the outputs of an RNN.

Consider an RNN unit with P input time series $\mathbf{u}(t) = (u_1(t), \ldots u_P(t)) \in \mathbb{R}^P$, and M hidden recurrent neurons described by a temporal state vector $\mathbf{h}(t) = (h_1(t), \ldots h_M(t)) \in \mathbb{R}^{1\times M}$, where the time variable consists of K points as $t = (t_1, t_2, \ldots, t_K)$. We show that in the continuous-time limit where the time step Δt between two sequential data points is very small ($\Delta t \ll 1$), the dynamics of a leaky RNN unit can be approximated by a system of first order nonlinear ODEs of the form,

$$\dot{\mathbf{h}} = -\tilde{\alpha}\mathbf{h} + \tilde{\alpha} f\left(\mathbf{W}_{\text{res}} \cdot \mathbf{h} + \mathbf{W}_{\text{in}} \cdot \mathbf{u} + \mathbf{b}\right), \tag{5}$$

where $\dot{\mathbf{h}} \equiv d\mathbf{h}/dt$ and the dot denotes the inner product. The input weights and biases are represented, respectively, by $\mathbf{W}_{\text{in}} \in \mathbb{R}^{M\times P}$ and $\mathbf{b} \in \mathbb{R}^M$, $\mathbf{W}_{\text{res}} \in \mathbb{R}^{M\times M}$ describes the recurrent weights, $\tilde{\alpha}$ is the leakage rate, and $f(\cdot)$ denotes a nonlinear activation function [21]. Applying a Euler discretization for the first derivatives, $\dot{\mathbf{h}} = \frac{\mathbf{h}_{n+1} - \mathbf{h}_n}{\Delta t}$, the system of Eq. (5) takes the discrete form

$$\mathbf{h}_{n+1} = (1 - \alpha)\,\mathbf{h}_n + \alpha f\left(\mathbf{W}_{\text{res}} \cdot \mathbf{h}_n + \mathbf{W}_{\text{in}} \cdot \mathbf{u}_n + \mathbf{b}\right), \tag{6}$$

with $\alpha = \tilde{\alpha}\Delta t$. Equation (6) describes the update of a leaky RNN unit and subsequently, it is a first order approximation of the continuous model described by Eq. (5). Since Eq. (6) determines all $\mathbf{h}_n$ of an RNN, Eq. (5) provides, without any computational cost, the first time derivatives as:

$$\dot{\mathbf{h}}_n = -\frac{\alpha}{\Delta t}\mathbf{h}_n + \frac{\alpha}{\Delta t} f\left(\mathbf{W}_{\text{res}} \cdot \mathbf{h}_n + \mathbf{W}_{\text{in}} \cdot \mathbf{u}_n + \mathbf{b}\right). \tag{7}$$

Higher order derivatives can be calculated by taking time derivatives of Eq. (7) and applying the chain rule to the $f(\cdot)$ term. The only numerical error in the first derivative of Eq. (7) is introduced through the assumption that Eq. (6) is derived from Eq. (5) by applying a Euler discretization. This error can be arbitrary small by appropriately choosing a small Δt. Consequently, the time derivatives of the hidden states can be estimated without using backpropagation.

Considering the general case of an RNN that returns R outputs $N(t) \in \mathbb{R}^R$, we read

$$N(t_n) = \mathbf{W_o} \cdot \mathbf{h}_n + \mathbf{b}_{\text{o}}, \tag{8}$$

where $\mathbf{W_o} \in \mathbb{R}^{R\times M}$ and $\mathbf{b}_{\text{o}} \in \mathbb{R}^R$ are the weights matrix and biases of an output linear layer. The time derivative of N can be calculated in a backpropagation-free mode using the result in Eq. (7) as:

$$\dot{N}(t_n) = \mathbf{W_o} \cdot \dot{\mathbf{h}}_n. \tag{9}$$

Backpropagating in RNNs is computationally expensive and can be impractical for large K. On the other hand, through Eq. (9) we can compute time derivatives without computational cost. Although throughout this study we apply Eqs. (7) and (9) for the RC architecture, the approach has broader implications since it holds for any RNN. Subsequently, it opens the door to a wide range of potential applications including general differential equation NN solvers and physics-informed RNNs.

4 Reservoir computing forms an ordinary differential equation solver

In this section, we introduce an unsupervised RC model that takes t as an input sequence, namely $\mathbf{u}_n = t_n$, and is trained to solve ODEs within the range of t. First, we examine single linear and nonlinear first order ODEs. Later, we modify the proposed RC to solve systems of first order ODEs. Similar to feed-forward NN solvers, the objective of the proposed machine learning method is to minimize the loss function of Eq. (4) for given ODE and ICs.

We employ an RC that returns one output sequence N, hence Eqs. (8) and (9) yield

$$N(t_n, \mathbf{W}_{\text{out}}) = \mathbf{W}_{\text{out}} \cdot \tilde{\mathbf{h}}(t_n), \tag{10}$$

$$\dot{N}(t_n, \mathbf{W}_{\text{out}}) = \mathbf{W}_{\text{out}} \cdot \dot{\tilde{\mathbf{h}}}(t_n), \tag{11}$$

where $\tilde{\mathbf{h}} = [1, \mathbf{h}]$ contains a column of ones accounting for the bias of the output layer of the RC, $\dot{\tilde{\mathbf{h}}} = [\mathbf{0}, \dot{\mathbf{h}}]$ since the constant bias vanishes after operating the derivative, and the readout (output) layer $\mathbf{W}_{\text{out}} \in \mathbb{R}^{R\times(M+1)}$ accounts for the only trainable (weights and bias) parameters of the RC. Using the parameterization in Eq. (2), we construct the RC solution

$$y(t, \mathbf{W}_{\text{out}}) = \psi_0 + g(t)\mathbf{W}_{\text{out}} \cdot \tilde{\mathbf{h}}(t), \tag{12}$$

with $g(0) = 0$ and thus, $y(0, \mathbf{W}_{\text{out}}) = \psi_0$. Having the RC parametric solution of Eq. (12) we train $\mathbf{W}_{\text{out}}$ such that Eq. (3) is minimized and subsequently, we obtain y that approximately satisfies the general ODE (1) and identically satisfies the given ICs.

The L of Eq. (4) can be minimized using gradient descent and backpropagation through a linear layer. Interestingly, for linear non-autonomous first order ODEs, a closed form solution of $\mathbf{W}_{\text{out}}$ that minimize Eq. (4) is derived and thus, numerical optimization is not required. Consequently, solving linear ODEs with RC is a backpropagation-free training method.

Linear differential equations, backpropagation-free learning method: In the RC architecture, the only adjustable parameters appear in the output layer, giving the opportunity to get an analytical closed-form solution for the optimal $\mathbf{W}_{\text{out}}$. We exploit this potential by studying first order linear non-homogeneous ODEs. These equations often appear in diffusion processes like fluid dynamics, and are described by the general linear differential equation:

$$a_1(t)\dot{\psi} + a_0(t)\psi - f(t) = 0, \tag{13}$$

where the coefficients $a_0(t)$, $a_1(t)$ and the force $f(t)$ are continuous functions of t. Minimizing Eq. (4) for the ODE Eq. (13) when y is used instead of ψ, a closed-form solution for $\mathbf{W}_{\text{out}}$ is derived to produce y that approximately solves the ODE (13). Substituting Eq. (12), Eq. (13) can be elegantly re-expressed in matrix notation as

$$A_1\dot{Y} + A_0 Y - F = 0, \tag{14}$$

with $F = \big(f(t_0), \ldots, f(t_K)\big)^T$, $A_i = \big(a_i(t_0), \ldots, a_i(t_K)\big)^T$ $(i = 0, 1)$, and the RC solution $Y = \big(y(t_0, \mathbf{W}_{\text{out}}), \ldots, y(t_K, \mathbf{W}_{\text{out}})\big)^T$ which is written according to Eq. (12) as:

$$\begin{aligned} Y &= \Psi_0 + (\mathbf{G} \circ \mathbf{H})\, \mathbf{W}_{\text{out}} \\ &= \Psi_0 + \mathbf{S}\mathbf{W}_{\text{out}}, \end{aligned} \tag{15}$$

and the associate time derivative reads:

$$\dot{Y} = \dot{\mathbf{S}}\mathbf{W}_{\text{out}}, \tag{16}$$

where $\circ$ denotes the Hadamard product, $\Psi_0 \in \mathbb{R}^K$ is the constant vector $\Psi_0 = (\psi_0, \ldots, \psi_0)^T$, $\mathbf{G} \in \mathbb{R}^{K\times M}$ is a matrix with repeating rows of $(g(t_0), \ldots, g(t_K))^T$, $\mathbf{H} \in \mathbb{R}^{K\times M}$ is the state matrix $\mathbf{H} = \Big(\tilde{\mathbf{h}}(t_0), \ldots, \tilde{\mathbf{h}}(t_K),\Big)^T$, and $\mathbf{S} = \mathbf{G} \circ \mathbf{H} \in \mathbb{R}^{K\times M}$. To derive a close-form solution of $\mathbf{W}_{\text{out}}$ we consider L_2 regularization, $L_{\text{reg}} = \lambda \mathbf{W}_{\text{out}}^T \mathbf{W}_{\text{out}}$, where λ is the regularization parameter [21]. Minimizing L of Eq. (4) for the ODE of Eq. (14) and with L_2 regularization, we get:

$$\begin{aligned} \frac{\partial}{\partial \mathbf{W}_{\text{out}}} \Big[\Big(A_1\dot{Y} + A_0 Y - F\Big)^T \Big(A_1\dot{Y} + A_0 Y - F\Big) + \lambda \mathbf{W}_{\text{out}}^T \mathbf{W}_{\text{out}} \Big] &= 0 \\ \Big(A_1\dot{\mathbf{S}}\mathbf{W}_{\text{out}} + A_0\mathbf{S}\mathbf{W}_{\text{out}} + A_0\Psi_0 - F\Big)^T \Big(A_1\dot{\mathbf{S}} + A_0\mathbf{S}\Big) + \lambda \mathbf{W}_{\text{out}}^T &= 0 \\ \big(\mathbf{W}_{\text{out}}^T D_{\mathbf{H}}^T + D_0^T\big)\, D_{\mathbf{H}} + \lambda \mathbf{W}_{\text{out}}^T &= 0, \end{aligned} \tag{17}$$

where we define the matrices

$$D_{\mathbf{H}} = A_1\dot{\mathbf{S}} + A_0\mathbf{S}, \tag{18}$$

$$D_0 = A_0\Psi_0 - F. \tag{19}$$

Solving Eq. (17) for $\mathbf{W}_{\text{out}}$ yields a closed-form equation of $\mathbf{W}_{\text{out}}$ that constructs an RC solution of Eq. (15) which approximately solves any linear non-homogeneous first order ODE, hence

$$\mathbf{W}_{\text{out}} = -\big(D_{\mathbf{H}}^T D_{\mathbf{H}} + \lambda \mathbf{1}\big)^{-1} D_{\mathbf{H}}^T D_0, \tag{20}$$

where $\mathbf{1}$ is a $1 \times M$ vector of ones. We read that Eq. (20) consists of two characteristic matrices, $D_{\mathbf{H}}$ and D_0 given by Eqs. (18) and (19), respectively. The former ($D_{\mathbf{H}}$) contains information of the RC hidden states $\mathbf{H}$, while the last (D_0) includes the ICs and the force function. Both characteristic matrices are informed about the differential equation since the coefficients $A_0,\ A_1$ appear in both places. We are able to obtain the closed-form solution $\mathbf{W}_{\text{out}}$ because Y and $\dot{Y}$ of Eqs. (15), (16) are both linear to $\mathbf{W}_{\text{out}}$. Thus, their first derivative with respect to $\mathbf{W}_{\text{out}}$ is independent of $\mathbf{W}_{\text{out}}$, and therefore, a linear system for $\mathbf{W}_{\text{out}}$ is derived in Eq. (17). Such a closed-form is not possible for nonlinear ODEs since a linear system of $\mathbf{W}_{\text{out}}$ cannot be derived with the parametrization of Eq. (15).

Equation (20) states that the RC solver can be trained to solve linear first order ODEs without using numerical optimization, such as gradient descent and backpropagation. The computationally costly part of the proposed RC solver is the hyperparameter optimization and this is the reason that an efficient method such as BO [34] is crucial. Through numerical experiments, we demonstrate that one set of hyper-parameters is sufficient for a wide range of ICs. Using the same set means that all the RC solutions for a particular ODE share the same states. Subsequently, we construct $\mathbf{H}$ for one IC and reuse them for additional ICs allowing a computationally efficient exploration of many ICs.

We evaluate the closed-form solution of Eq. (20) by solving two linear ODEs for different ICs. During experimental evaluation we adopt the efficient parametric function used in Refs. [10, 12]:

$$g(t) = 1 - e^{-t}. \tag{21}$$

We assess the RC performance by calculating the root-mean-square-residuals (RMSR) of the RC solutions, namely

$$\text{RMSR}(t) = \sqrt{\frac{1}{L}\sum_{\text{ICs}}\Big(a_1(t)\dot{y} + a_0(t)y - f(t)\Big)^2}, \tag{22}$$

where the sum denotes averaging along L different ψ_0. For the first experiment we consider the ODE

$$\dot{\psi} + \psi = \sin(t), \tag{23}$$

which has the exact solution:

$$\psi(t) = e^{-t}\left(y_0 + \frac{1}{2}\right) + \frac{1}{2}\left(\sin(t) - \cos(t)\right). \tag{24}$$

From Eq. (23) we note that $a_0 = a_1 = 1$ and $f(t) = \sin(t)$. We get the optimal $\mathbf{W}_{\text{out}}$ by calculating the characteristic matrices of Eqs. (19), (18) and substituting them into Eq. (20). Then, we construct the RC solution by employing Eq. (12). The RC solutions of Eq. (23) along with the exact solutions (24) are demonstrated in the left side of Fig. 1 for several ICs in the range $\psi_0 = [-5.5, 5.5]$. Upper graph: the solid blue lines account for the RC solutions while the dashed red curves indicate the exact solutions; each pair of solid-dashed lines corresponds to a solution with different ψ_0. The lower image shows the RMSR. The blue color indicates the ICs used in the BO to obtain the optimal hyper-parameters, while for the solutions indicated by green lines we apply only the exact $\mathbf{W}_{\text{out}}$.

The second numerical experiment is an ODE with time dependent coefficients, defined by:

$$\dot{\psi} + t^2\psi = \sin(t), \tag{25}$$

where $a_0 = t^2$, $a_1 = 1$, and $f = \sin(t)$. We calculate the optimal $\mathbf{W}_{\text{out}}$ with Eq. (20) and construct RC solutions in the range of ICs $[-10, 10]$. The RC predictions are shown in the right panel of Fig. 1. The upper graph demonstrates the predicted trajectories, while the lower image outlines the RMSR. There is no exact solution for the Eq. (25), hence only the RC predictions are shown in the upper graph. We employ BO only for a few ICs shown in blue.

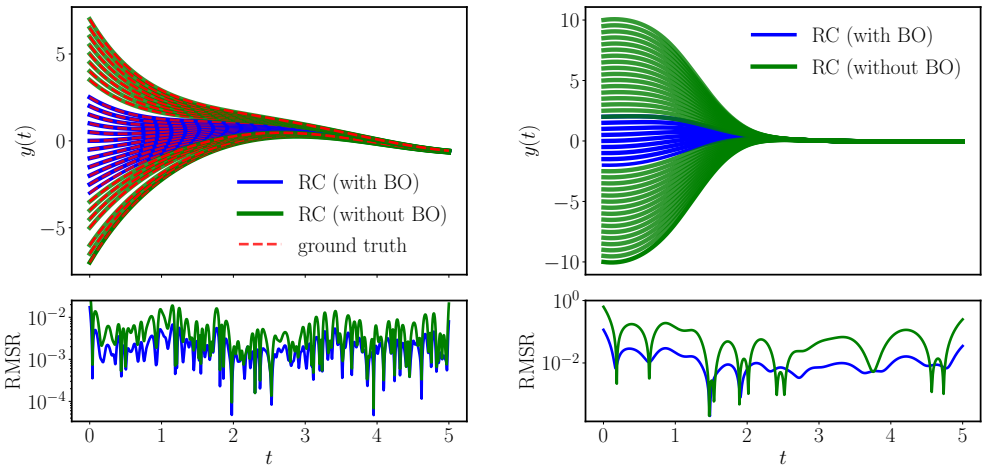

Figure 1: Backpropagation-free training for solving the linear ODEs of Eq. (23) (right) and Eq. (25) (left). Upper graphs outline the RC solutions for different ICs. Lower panels show the RMSR. Blue and green lines represent ICs used and not used in BO.

In both experiments, BO was applied to a bundle of ICs implying that a single set of hyper-parameters is sufficient to construct different RC solutions of the same ODE. In particular, applying BO and using the exact $\mathbf{W}_{\text{out}}$ of Eq. (20), we get the optimal hyperparameters that yielding the RC solutions shown with blue in Fig. 1. Then, using the same hyperparameters we exactly solve (without BO) for $\mathbf{W}_{\text{out}}$ to construct RC solutions for different ICs as they are indicated in green in Fig. 1. Moreover, the results shown in Fig. 1 validate the closed-form solution of Eq. (20) verifying that the RC solver is a backpropagation-free unsupervised machine learning method for solving linear first order ODEs.

Nonlinear differential equations, training with gradient descent: It is not possible to derive a closed-form solution for $\mathbf{W}_{\text{out}}$ for nonlinear ODEs. Nevertheless, RC can solve nonlinear ODEs by training $\mathbf{W}_{\text{out}}$ using gradient descent (GD). We demonstrate the capacity of RC in solving nonlinear equations by studying Bernoulli type nonlinear equations of the form:

$$a_1(t)\dot{y} + a_0(t)y + q(t)y^2 = f(t). \tag{26}$$

Although it is not possible to derive an exact solution for the optimal $\mathbf{W}_{\text{out}}$, an approximate closed formula is obtained through a linearization procedure. Then, we use the linearized $\mathbf{W}_{\text{out}}$ instead of random weights to start the GD. This is a transfer learning approach that drastically accelerates and improves the training (see SM for more details). In the context of linearization approximation any nonlinear term of $\mathbf{S}\mathbf{W}_{\text{out}}$ is dropped. This is a valid approximation since $\mathbf{W}_{\text{out}} \ll 1$ due to the L_2 regression and $\mathbf{S}$ varies in the range $[-1, 1]$ due to the parametric function of Eq. (21) and the activation functions $\tanh()$ or $\sin()$, which are all bounded within $[-1, 1]$. Consequently, we read $\mathbf{W}_{\text{out}}\mathbf{S} \ll 1$ and thus, higher orders can be neglected, namely $(\mathbf{W}_{\text{out}}\mathbf{S})^\nu \simeq 0$ for any integer $\nu > 1$.

Minimizing the loss function of Eq. (4) for the nonlinear ODE (26) yields

$$\Big(A_1\dot{Y} + A_0 Y + QY^TY - F\Big)^T \frac{\partial}{\partial \mathbf{W}_{\text{out}}}\Big(A_1\dot{Y} + A_0 Y + QY^TY - F\Big) + \lambda \mathbf{W}_{\text{out}}^T = 0$$

$$\Big(\mathbf{W}_{\text{out}}^T D_{\mathbf{H}}^T + D_0^T + Q\left(\Psi_0^2 + 2\Psi_0 \mathbf{W}_{\text{out}}^T\right)\Big)\Big(D_{\mathbf{H}} + 2Q\left(\Psi_0 + \mathbf{S}\mathbf{W}_{\text{out}}\right)\Big) + \lambda \mathbf{W}_{\text{out}}^T = 0 \tag{27}$$

$$\mathbf{W}_{\text{out}}^T\Big(\tilde{D}_{\mathbf{H}}^T \tilde{D}_{\mathbf{H}} + \tilde{2}QD_0^T\mathbf{S} + \lambda\mathbf{1}\Big) + \tilde{D}_0^T \tilde{D}_{\mathbf{H}} = 0, \tag{28}$$

where in Eq. (27) nonlinear terms of $\mathbf{S}\mathbf{W}_{\text{out}}$ are dropped. $Q = (q(t_0), \ldots, q(t_K))$, and the modified characteristic matrices are defined as:

$$\tilde{D}_{\mathbf{H}} = D_{\mathbf{H}} + 2Q\Psi_0, \tag{29}$$

$$\tilde{D}_0 = D_0 + Q\Psi_0^2. \tag{30}$$

The linear algebraic system of Eq. (28) can be inverted to give the linearized RC weights as:

$$\mathbf{W}_{\text{out}} = -\Big(\tilde{D}_{\mathbf{H}}^T \tilde{D}_{\mathbf{H}} + \tilde{2}QD_0^T\mathbf{S} + \lambda\mathbf{1}\Big)^{-1} \tilde{D}_{\mathbf{H}}^T \tilde{D}_0. \tag{31}$$

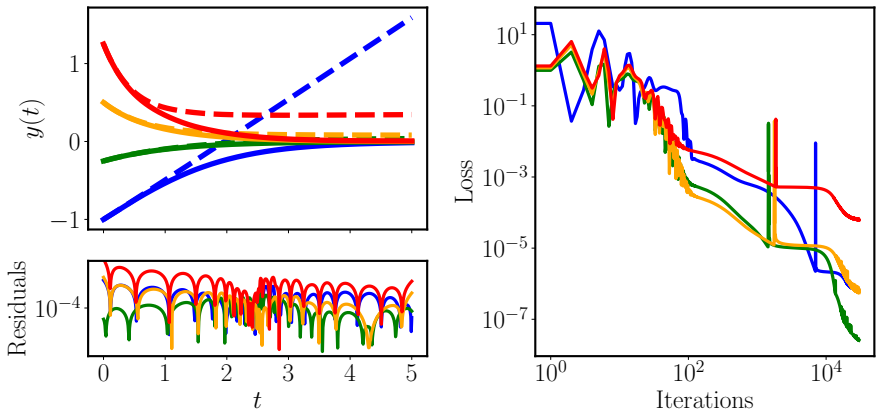

Figure 2: Nonlinear ODE. The upper-left image shows RC solutions obtained by linearized weights (dashed lines) and the solutions obtained after gradient descent optimization (solid lines). The lower-left panel outlines the residuals for each RC solution after applying GD. The graph on the right shows the loss function during the GD iterations. Each color represents a different IC.

We assess the performance of the RC by solving the ODE (26) for $a_0 = a_1 = 1$, $f = 0$, and $q = 0.5$. Starting with the linearized $\mathbf{W}_{\text{out}}$ of Eq. (31), we employ GD to train the parameters. This is efficient since we only optimize a single linear layer. Figure 2 presents the RC solutions (top-left graph) and the associated residuals (bottom-left) for different ICs indicated by different colors. Upper plot: the

dashed lines indicate RC predictions obtained by solely applying the linearized $\mathbf{W}_{\text{out}}$ of Eq. (31), before applying GD; solid lines are the RC predictions after GD. The right side of Fig. 2 outlines the loss function during the GD iterations where each colored loss trace corresponds to the associated colored line in the left plots.

Systems of ordinary differential equations, Hamiltonian systems: In this section, we employ RC to solve systems of ODEs. To apply RC to systems, the network architecture needs to be modified to return multiple outputs N_j, where j indicates a different output. The number of the N_j needs to be the same with the number of the equations in the system. Each N_j has a different set of weights $\mathbf{W}_{\text{out}}^{(j)}$, while all N_j share the same hidden states, namely:

$$N_j = \mathbf{W}_{\text{out}}^{(j)} \cdot \tilde{\mathbf{h}}. \tag{32}$$

Moreover, the loss function (4) includes all the ODEs included in the system. We exploit the RC solver in solving the equations of motion for a nonlinear Hamiltonian system, the nonlinear oscillator. In this system the energy is conserved and thus, we adopt the energy regularization introduced in Ref. [12] that drastically accelerates the training and improves the fidelity of the predicted solutions.

Hamiltonian systems obey the energy conservation law. Specifically, these systems are characterized by a time-invariant hamiltonian function that represents the total energy. The hamiltonian of a nonlinear oscillator with unity mass and frequency is given by:

$$\mathcal{H}(x,p) = \frac{p^2}{2} + \frac{x^2}{2} + \frac{x^4}{4}, \tag{33}$$

and the associated equations of motion read:

$$\dot{x} = p \tag{34}$$

$$\dot{p} = -x - x^3 \tag{35}$$

where $x,\ p$ are the position and momentum variables [12]. The loss function consists of three parts: L_{ODE} for the ODEs (34), (35); a hamiltonian penalty $L_{\mathcal{H}}$ that penalizes violations in the energy conservation and is defined by Eq. (33); and a regularization term. Subsequently, the total L is:

$$\begin{aligned} L &= L_{\text{ODE}} + L_{\mathcal{H}} + L_{\text{reg}} \\ &= \sum_{n=0}^{K} \Big[\left(\dot{x}_n - p_n\right)^2 + \left(\dot{p}_n + x_n + x_n^3\right)^2 + \left(E - \mathcal{H}(x_n, p_n)\right)^2 \Big] + L_{\text{reg}}. \end{aligned} \tag{36}$$

$E = \mathcal{H}(x_0, p_0)$ represents the total energy defined by the ICs $x(0) = x_0$, $p(0) = p_0$. Earlier we choose L_2 regularization because we derived exact solutions for the $\mathbf{W}_{\text{out}}$; this was possible with L_2. For systems of ODEs we do not derive exact $\mathbf{W}_{\text{out}}$ and thus, we can apply any L_{reg}. We use the elastic net regularization which has been shown to be a dominant generalization of L_1 and L_2 (see the SM) [44]. The RC solutions are defined through Eqs. (12) and (32) as:

$$x_n = x_0 + \mathbf{W}_{\text{out}}^{(x)} \cdot \Big(g(t_n)\tilde{\mathbf{h}}(t_n) \Big), \tag{37}$$

$$p_n = p_0 + \mathbf{W}_{\text{out}}^{(p)} \cdot \Big(g(t_n)\tilde{\mathbf{h}}(t_n) \Big). \tag{38}$$

We employ the RC solver from the `rcTorch` library to solve the Eqs. (34) and (35). Specifically, we minimize Eq. (36) by applying GD and use $g(t)$ of Eq. (21). The results are outlined in Fig. 3. First, we consider a singe set of ICs, $(x_0, p_0) = (1.3, 1)$, and use the hybrid mode consisting of GD and BO to find the optimal hyper-parameters. This optimization is performed in the time range $t = [0, 6\pi]$. Then, using the obtained hyper-parameters we expand the time range to $t = [0, 10\pi]$ and generate RC solutions solely using GD. The RC solution for x is presented in the upper left graph, while the lower plot shows the residuals. In both images, the dashed red line indicates the end of the BO. Using the same hyper-parameter set, we apply the RC solver in a range of ICs. The right panel we shows the phase-space diagram ($x - p$ plot) for the IC used in the BO (blue line) and for the ICs where only GD is applied (green lines). In the SM we report the residuals and the loss traces during the GD for all the investigated ICs. Figure 3 is evidence that a single hyper-parameter set can be used to solve an ODE system for different ICs and time ranges.

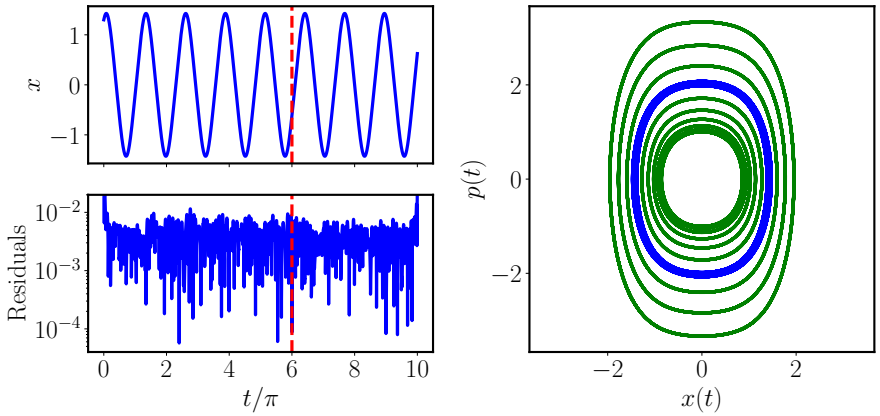

Figure 3: Nonlinear oscillator. Left panel represents a single IC. BO is applied up to red dahsed line. Right graph demonstrate the RC solutions for different ICs not used in BO optimization (green), while blue line is associated to the solution shown in the left plot.

5 Conclusion

Recently, NN differential equation solvers have attracted a lot of interest. These solvers present some crucial advantages over traditional integrators such as they provide analytical and differentiable solutions that can be inverted, suffer less from the "curse of dimensionality", and learn general solutions. While many methods and scientific libraries for NN solvers have been reported an RNN solver is still missing from the literature.

Novelty: We presented a novel RNN ODE solver in the context of an unsupervised RC and assessed the performance of the RC solver by solving linear and nonlinear ODEs. We showed that a closed-form solution for the RC weights is possible for solving linear ODEs with explicit time dependence, leading to a backpropagation-free optimization method. For nonlinear system, we applied GD which is very efficient since in the RC architecture we train only a linear output layer. For the hyper-parameter optimization, we employed BO integrated with GD. We found that a single set of hyper-parameters can be shared for solving ODEs for different initial conditions and time intervals.

Limitations: For BO we used TURBO-1, however, dominant methods such as TURBO-m have been shown to be more robust and get stuck less often in local minima. The efficacy of the proposed RC solver has not been corroborated for very demanding ODEs due to the limited capacity in finding optimal hyperparameters. Although we derived a closed-form solution for a single linear ODE, such a closed formula has not been derived for linear systems of ODEs. Hence, GD is required for solving systems. The proposed RC architecture takes an input t restricting the RC to solve only ODEs. With more variables as inputs, the RC will be able to solve partial differential equations.Currently a uniformly spaced input t has been used, so the efficacy of RC in solving stiff ODEs is not yet determined. A thorough investigation of the optimal reservoir dynamics has not been performed in this study. In this work, we demonstrated that unsupervised RC can solve ODEs by solving a few specific problems using the `rcTorch` library. Future updates of `rcTorch` will remove the weakness discussed above, significantly expanding the potential applications of the library. A more general and more powerful `rcTorch` version is slated to be released in the coming months.

Training an RC is extremely fast (see SM for runtime details and coding demonstrations). Although more investigation is warranted, RC has the potential to make NN solvers dramatically faster and even potentially competitive with integrators. Furthermore, RC has widely been adopted to form neuromorphic devices. Since a single set of hyper-parameters can be used to solve a system of ODEs for different ranges of ICs and time ranges, a physical reservoir can be designed to respect these hyper-parameters. Then a readout output layer will be efficiently trained to solve a system of ODEs for different ICs and times. Subsequently, the proposed machine learning method can be potentially implemented to form a neuromorphic computing device for solving ODEs.

6 Broader Impact

Solving differential equations is substantial in every scientific field including engineering, applied physics, quantum chemistry, finance, and biology. Solving these equations can be extremely demanding and frequently prohibitive due to the limitations of existing numerical methods. Subsequently, new technologies and more efficient methods for solving differential equations are crucial to accelerate progress in scientific research. In this work, we introduced a general framework for solving differential equations with recurrent neural networks. We demonstrate the method by solving systems of ordinary differential equations. Yet, this method can be expanded to systems of partial differential equations as well as eigenvalue problems. Moreover, we suggest a computationally efficient method to calculate time derivatives of the outputs of recurrent networks, making possible the development of recurrent data-driven physics-informed neural networks.

Societal and Environmental Impact: Solving differential equations with RC may have negative social impacts depending on what the user employs them for, but they are not immediately obvious and are likely indirect. As far as the environment is concerned, Bayesian optimization can be computationally expensive, but it is not expensive enough to warrant concerns about environmental impacts when compared to heavier models like transformers, feed forward networks or RNNs such as LSTMs. There is an upfront cost with RC, but even after taking this into consideration, our models are likely faster, more efficient, and much less energy intensive (having a smaller carbon footprint) than comparable feed forward neural network differential equation solvers. Moreover, the proposed method can potentially be used to build neuromorphic devices, drastically accelerating computations with extremely low energy consumption.

Acknowledgments

The authors would like to thank Shaan Desai, Shivam Raval, Hargun Singh Oberoi for their comments on the manuscript and numerical experiments. In addition, we would like to thank Reinier Maat for advising us on the development of the `rcTorch` library.

References

[1] Lu Lu et al. "Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators involving nonlinear partial differential equations". In: *Nature machine intelligence* 3 (2021), pp. 218–229. DOI: https://doi.org/10.1038/s42256-021-00302-5.

[2] Samuel H. Rudy et al. "Data-driven discovery of partial differential equations". In: *Science Advances* 3.4 (2017). DOI: 10.1126/sciadv.1602614.

[3] J. Nathan Kutz et al. "Data-Driven discovery of governing physical laws and their parametric dependencies in engineering, physics and biology". In: *2017 IEEE 7th International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP)* (2017), pp. 1–5. DOI: 10.1109/CAMSAP.2017.8313100.

[4] Yohai Bar-Sinai et al. "Learning data-driven discretizations for partial differential equations". In: *Proceedings of the National Academy of Sciences of the United States of America* 116.31 (2019), 15344–15349. DOI: 10.1073/pnas.1814058116.

[5] Sam Greydanus, Misko Dzamba, and Jason Yosinski. "Hamiltonian Neural Networks". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 15379–15389. URL: http://papers.nips.cc/paper/9672-hamiltonian-neural-networks.pdf.

[6] Tom Bertalan et al. "On learning Hamiltonian systems from data". In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 29.12 (2019), p. 121107. DOI: 10.1063/1.5128231.

[7] Anshul Choudhary et al. "Physics-enhanced neural networks learn order and chaos". In: *Phys. Rev. E* 101 (6 2020), p. 062207. DOI: 10.1103/PhysRevE.101.062207.

[8] Yuyao Chen et al. "Physics-informed neural networks for inverse problems in nano-optics and metamaterials". In: *Opt. Express* 28.8 (2020), pp. 11618–11633. DOI: 10.1364/OE.384875.

[9] Isaac E. Lagaris, Aristidis Likas, and Dimitrios I. Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations". In: *IEEE transactions on neural networks* 9 (5 1998), pp. 987–1000. DOI: 10.1109/72.712178.

[10] Pola Lydia Lagari et al. “Systematic Construction of Neural Forms for Solving Partial Differential Equations Inside Rectangular Domains, Subject to Initial, Boundary and Interface Conditions”. In: *International Journal on Artificial Intelligence Tools* 29 (5 2020), p. 2050009. DOI: 10.1142/S0218213020500098.

[11] Cedric Flamant, Pavlos Protopapas, and David Sondak. *Solving Differential Equations Using Neural Network Solution Bundles*. 2020. arXiv: 2006.14372 [cs.LG].

[12] Marios Mattheakis et al. “Hamiltonian Neural Networks for solving differential equations”. In: *arXiv:2001.11107 [physics]* (2020). URL: http://arxiv.org/abs/2001.11107.

[13] Jiequn Han, Arnulf Jentzen, and E Weinan. “Solving high-dimensional partial differential equations using deep learning”. In: *Proceedings of the National Academy of Sciences of the United States of America* 115 34 (2017), pp. 8505–8510. DOI: 10.1073/pnas.1718942115.

[14] Justin A. Sirignano and Konstantinos Spiliopoulos. “DGM: A deep learning algorithm for solving partial differential equations”. In: *Journal of Computational Physics* 375 (2018), pp. 1339–1364. DOI: 10.1016/j.jcp.2018.08.029.

[15] Feiyu Chen et al. “NeuroDiffEq: A Python package for solving differential equations with neural networks”. In: *Journal of Open Source Software* 5.46 (2020), p. 1931. DOI: 10.21105/joss.01931.

[16] Henry Jin, Marios Mattheakis, and Pavlos Protopapas. “Unsupervised Neural Networks for Quantum Eigenvalue Problems”. In: *2020 NeurIPS Workshop on Machine Learning and the Physical Sciences*. NeurIPS, 2020. URL: https://ml4physicalsciences.github.io/2020/files/NeurIPS_ML4PS_2020_16.pdf.

[17] Alessandro Paticchio et al. “Semi-supervised Neural Networks solve an inverse problem for modeling Covid-19 spread”. In: *2020 NeurIPS Workshop on Machine Learning and the Physical Sciences*. NeurIPS, 2020. URL: https://ml4physicalsciences.github.io/2020/files/NeurIPS_ML4PS_2020_29.pdf.

[18] P. Grohs et al. “A proof that artificial neural networks overcome the curse of dimensionality in the numerical approximation of Black-Scholes partial differential equations”. In: *arXiv:1809.02362 [math.NA]* (2018). URL: https://arxiv.org/abs/1809.02362.

[19] M. Raissi, P. Perdikaris, and G.E. Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational Physics* 378 (2019), pp. 686–707. DOI: https://doi.org/10.1016/j.jcp.2018.10.045.

[20] Herbert Jaeger and Harald Haas. “Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication”. In: *Science* 304.5667 (2004), pp. 78–80. DOI: 10.1126/science.1091277.

[21] Zhixin Lu et al. “Reservoir observers: Model-free inference of unmeasured variables in chaotic systems”. In: *Chaos* 27.4 (2017). DOI: 10.1063/1.4979665.

[22] George Neofotistos et al. “Machine Learning With Observers Predicts Complex Spatiotemporal Behavior”. In: *Frontiers in Physics* 7 (2019), p. 24. DOI: 10.3389/fphy.2019.00024.

[23] Jaideep Pathak et al. “Model-Free Prediction of Large Spatiotemporally Chaotic Systems from Data: A Reservoir Computing Approach”. In: *Phys. Rev. Lett.* 120 (2 2018), p. 024102. DOI: 10.1103/PhysRevLett.120.024102.

[24] Jonathan Dong et al. “Reservoir Computing meets Recurrent Kernels and Structured Transforms”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 16785–16796. URL: https://proceedings.neurips.cc/paper/2020/file/c348616cd8a86ee661c7c98800678fad-Paper.pdf.

[25] Sandra Nestler et al. “Unfolding recurrence by Green’s functions for optimized reservoir computing”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 17380–17390. URL: https://proceedings.neurips.cc/paper/2020/file/c94a589bdd47870b1d74b258d1ce3b33-Paper.pdf.

[26] Anteo Smerieri et al. “Analog readout for optical reservoir computers”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/250cf8b51c773f3f8dc8b4be867a9a02-Paper.pdf.

[27] Andrew Katumba et al. “Low-loss photonic reservoir computing with multimode photonic integrated circuits”. In: *SCIENTIFIC REPORTS* 8.1 (2018), 2653:1–2653:10. DOI: 10.1038/s41598-018-21011-x.

[28] G. Tanaka et al. “Recent Advances in Physical Reservoir Computing: A Review”. In: *Neural networks : the official journal of the International Neural Network Society* 115 (2019), pp. 100–123. DOI: /10.1016/j.neunet.2019.03.005.

[29] Satoshi Sunada and Atsushi Uchida. “Photonic reservoir computing based on nonlinear wave dynamics at microscale”. In: *SCIENTIFIC REPORTS* 9.1 (2019), 19078:1–19078:10. DOI: 10.1038/s41598-019-55247-y.

[30] Laurent Larger et al. “High-Speed Photonic Reservoir Computing Using a Time-Delay-Based Architecture: Million Words per Second Classification”. In: *Phys. Rev. X* 7 (1 2017), p. 011015. DOI: 10.1103/PhysRevX.7.011015.

[31] Giulia Marcucci, Davide Pierangeli, and Claudio Conti. “Theory of Neuromorphic Computing by Waves: Machine Learning by Rogue Waves, Dispersive Shocks, and Solitons”. In: *Phys. Rev. Lett.* 125 (9 2020), p. 093901. DOI: 10.1103/PhysRevLett.125.093901.

[32] Huong Ha et al. *Bayesian Optimization with Unknown Search Space*. 2019. arXiv: 1910.13092 [stat.ML].

[33] David Eriksson et al. *Scalable Global Optimization via Local Bayesian Optimization*. 2020. arXiv: 1910.01739 [cs.LG].

[34] Jacob Reinier Maat, Nikos Gianniotis, and Pavlos Protopapas. “Efficient Optimization of Echo State Networks for Time Series Datasets”. In: *2018 International Joint Conference on Neural Networks (IJCNN)*. 2018, pp. 1–7. DOI: 10.1109/IJCNN.2018.8489094.

[35] Zhongxiang Dai, Kian Hsiang Low, and Patrick Jaillet. *Federated Bayesian Optimization via Thompson Sampling*. 2020. arXiv: 2010.10154 [cs.LG].

[36] Maximilian Balandat et al. “BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 21524–21538. URL: https://proceedings.neurips.cc/paper/2020/file/f5b1b89d98b7286673128a5fb112cb9a-Paper.pdf.

[37] Lu Lu et al. “A deep learning library for solving differential equations”. In: *SIAM Review* 63 (2021), pp. 208–228. DOI: 10.1137/19M1274067.

[38] Oliver Hennigh et al. “NVIDIA SimNet: an AI-accelerated multi-physics simulation framework”. In: *arXiv:2012.07938 [physics.flu-dyn]* (2020). URL: https://arxiv.org/abs/2012.07938.

[39] George Em Karniadakis et al. “Physics-informed machine learning”. In: *Nature Review Physics* (2021). DOI: https://doi.org/10.1038/s42254-021-00314-5.

[40] Kurt Hornik. “Approximation capabilities of multilayer feedforward networks”. In: *Neural Networks* 4 (1991), pp. 251–257. DOI: 10.1016/0893-6080(91)90009-T.

[41] Bo Chang et al. *Reversible Architectures for Arbitrarily Deep Residual Neural Networks*. 2018. URL: https://aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16517.

[42] Ricky T. Q. Chen et al. “Neural Ordinary Differential Equations”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. 2018. URL: https://proceedings.neurips.cc/paper/2018/file/69386f6bb1dfed68692a24c8686939b9-Paper.pdf.

[43] Michael Poli et al. “Hypersolvers: Toward Fast Continuous-Depth Models”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 21105–21117. URL: https://proceedings.neurips.cc/paper/2020/file/f1686b4badcf28d33ed632036c7ab0b8-Paper.pdf.

[44] Hui Zou and Trevor Hastie. “Regularization and variable selection via the Elastic Net”. In: *Journal of the Royal Statistical Society, Series B* 67 (2005), pp. 301–320.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] . See section: Conclusion.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] . See section: Broader Impact.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] .
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] . Sections 3 and 4 have complete discussions.
   (b) Did you include complete proofs of all theoretical results? [Yes] . Section 3 and 4 contain all the proofs.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] . Details can be found in the supplementary material (SM). We also provide the URL for a github public repository that contains the library and notebooks used in this study.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] . They are stated in section 4 and more details can be found in the SM.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] . In section 4 and in the SM.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] . All the details are reported in the SM.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] .
   (b) Did you mention the license of the assets? [Yes] .
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] . Yes, a new asset for the library used in this study is included.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] . There was no need for consent.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] . In this neural network method we do not use data for the training.
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A] . We did not use crowdsourcing or conducted research with human subjects.
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A] . We did not use crowdsourcing or conducted research with human subjects.
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A] . We did not use crowdsourcing or conducted research with human subjects.

Supplementary Material: Unsupervised Reservoir Computing for Solving Ordinary Differential Equations

Marios Mattheakis,* **Hayden Joy, Pavlos Protopapas**
John A. Paulson School of Engineering and Applied Sciences, Harvard University
Cambridge, Massachusetts 02138, United States
mariosmat@seas.harvard.edu, hjoy@college.harvard.edu, pavlos@seas.harvard.edu

Supplementary python notebook overview

We provide 6 python notebooks showing our results presented in the main manuscript and in this supplementary material (SM). They break into two categories: Bayesian optimization (BO) for hyper-parameters used in the reservoir computing (RC) and for RC training. The training notebooks demonstrate the differential equation solutions which appeared in the main manuscript and in the SM. The BO notebooks show how we optimized for hyper-parameters for the various models. We note that BO gives different results after every optimization due to random initialization. Although they can vary in quality, the obtained hyper-parameters are always valid and thus, the BO is robust. All the notebooks also have timed cells so as to approximately reproduce the table 1 which shows the computational time taken for the various experiments.

Below is a list of supplementary jupyter notebooks and a brief description of each. All the notebooks reproducing the results can be found at the following link:
https://github.com/blindedjoy/RcTorch/tree/master/final_notebooks

- **Linear_solutions.ipynb**
  Demonstrates the solutions to the linear differential equations (which did not need gradient descent).
- **Bernoulli_solutions.ipynb**
  Demonstrates the solutions to the bernoulli equation with three solutions types: linearization only, backpropagation only, and a combination of linearization and backpropagation.
- **Systems_solutions.ipynb**
  Demonstrates the solutions to the nonlinear oscillator.
- **Linear_BO.ipynb**
  A notebook showing BO similar to what was used to find the hyper-parameter set for the linear equations. Does not need GPUs.
- **Bernoulli_BO.ipynb**
  A notebook showing BO similar to what was used to find the hyper-parameter set for the bernoulli equation. Should be run with GPUs.
- **systems_BO.ipynb**
  A notebook showing BO similar to what was used to find the hyper-parameter set for the nonlinear oscillator equation. Should be run with GPUs.

*https://scholar.harvard.edu/marios_matthaiakis/home

Preprint. Under review.

Experiment Runtimes

In Table 1 we report the experiment times needed for different experiments presented in the main manuscript and in the SM. All the notebooks are accessible through the link in the above section, hence the experiments are reproducible. The CPU experiments were run on a MacBook Pro (16-inch, 2019) with a 2.4 GHz 8-Core Intel Core i9 processor and 64 GB 2667 MHz DDR4 of memory. For users with a less powerful machine, a smaller batch size should be used since the batch size corresponds to the number of RCs being trained in parallel via the multi-processing package; however, this will slow down the computations. On a mac machine, activity monitor should be used to make sure that we are not overloading the memory which will cause data to be read in and out from the hard-drive which dramatically slows down performance.

The GPU experiments were run on google colab (with 12 GB of RAM), therefore a batch size of 1 was used due to this constraint. We note that GD (backpropagation) for different initial conditions (ICs) could be run in parallel, however, this is not yet implemented in the current version of the `rcTorch` . In addition, the library has not been fully optimized implying large further optimization gains.

Differential equations	# nodes	# ICs	solve time	BO time	epochs	batch size
Linear ODEs (CPUs)		20				
simple population	250	20	0.795	184	–	10
driven population	500	20	1.62	254	–	10
$t-$dependent coefficients	500	20	0.895	197	–	10
Linear ODEs (CPUs)		100				
simple population	250	100	1.32	316	–	10
driven population	500	100	3.74	864	–	10
$t-$dependent coefficients	500	100	3.03	774	–	10
Nonlinear Bernoulli (GPU)						
Linearization	500	4	1.15	–	–	–
GD only	500	4	191	3075	5000	1
GD with linearized weights	500	4	184	3075	5000	1
System of ODEs (GPU)						
Nonlinear oscillator	500	2	129	4440	5000	1

Table 1: Computational time needed by the RC solver for the experiments presented in the notebooks

Hyper-parameters list

For all experiments a random state of 209 was used which was set with pytorch's `manual_seed` code. We did not optimize the random state.

```
Simple population equation (not presented in the main manuscript):
{'dt': 0.0031622776601683794,
 'n_nodes': 250,
 'connectivity': 0.7170604557008349,
 'spectral_radius': 1.5755887031555176,
 'regularization': 0.00034441529823729916,
 'leaking_rate': 0.9272222518920898,
 'bias': 0.1780446171760559}

Driven population equation (Eq. (23) in the main manuscript):
{'dt': 0.0031622776601683794,
 'n_nodes': 500,
 'connectivity': 0.7875262340500385,
 'spectral_radius': 9.97140121459961,
 'regularization': 8.656278081920211,
```

```
 'leaking_rate': 0.007868987508118153,
 'bias': -0.2435922622680664}

First order ODE with time dependent coefficients (Eq. (25) in the main manuscript):
{'n_nodes': 500,
 'connectivity': 0.09905712745750006,
 'spectral_radius': 1.8904799222946167,
 'regularization': 714.156090350679,
 'leaking_rate': 0.031645022332668304,
 'bias': -0.24167031049728394,
 'dt' : 0.005}

Bernoulli type nonlinear ODE (Eq. (26)  in the main manuscript):
{'dt': 0.007943282347242814,
 'n_nodes': 500,
 'connectivity': 0.0003179179463749722,
 'spectral_radius': 7.975825786590576,
 'regularization': 0.3332787303378571,
 'leaking_rate': 0.07119506597518921,
 'bias': -0.9424528479576111}
epochs: 30000
learning_rate: 0.01
spikethreshold: 0.25


Nonlinear Oscillator system (Eqs. (34) and (35)
in the main manuscript):
{'dt': 0.001,
 'regularization': 48.97788193684461,
 'n_nodes': 500,
 'connectivity': 0.017714821964432213,
 'spectral_radius': 2.3660330772399902,
 'leaking_rate': 0.0024312976747751236,
 'bias': 0.37677669525146484,
 'enet_alpha': 0.2082211971282959,
 'enet_strength': 0.118459548397668,
 'spikethreshold': 0.43705281615257263,
 'gamma': 0.09469877928495407,
 'gamma_cyclic': 0.999860422666841}
epochs: 50000
learning_rate: 0.01
```

Numerical implementation

RC architecture

The `rcTorch` library presented and used in the study was written in pytorch and was adapted from the library developed in Ref. [**Maat2018**]. We used RC reservoirs consisting of 500 hidden nodes. This architecture was found to be sufficiently expressive to solve the ODEs discussed in the main manuscript of this study. We note that the computational complexity and memory cost of an RC grow quadratically with respect to the number of nodes [**dong2020reservoir**]. In `rcTorch` we are allowed to choose any activation function we want as long as it is bounded; unbounded activation functions such as *ReLU* have been shown not to converge in the context of RC [**dong2020reservoir**]. For the first order ODEs we use $\tanh(\cdot)$ activation, whereas, for the system of ODEs we apply a $\sin(\cdot)$ activation since it has been shown that in dynamical systems this type of activation is very effective [**mariosHNN**].

Bayesian optimization

Although RNNs are very powerful models in the handling of sequential data, they suffer from the vanishing and exploding gradient problem. Echo-state RNNs like RCs eliminate this issue by fixing their input, hidden weights, and hidden and biases. Moreover, training RNNs over long sequences (time series) is a computationally demanding task. Echo state networks, by contrast, are very efficient and extremely fast due to their special architecture. The computationally expensive aspect of building an effective RC model is to find the optimal hyper-parameters that define the network architecture. Here, we review a BO method that is introduced in Ref. [**Maat2018**] and is adopted in the library (`rcTorch`) used in this study.

Turbo-1: BO is a powerful method for analyzing expensive acquisition functions like the training of neural networks. It is a likelihood maximization problem where the surrogate model is typically a gaussian process model (GP). We use the Turbo-1 algorithm outlined in a BoTorch tutorial [**TURBO2020**, **Botorch2020**] which, like global BO methods, is robust to noisy observations and has rigorous uncertainty estimates. However, an advantage of Turbo over other BO algorithms is that it does not suffer from "the overemphasized exploration that results from global optimization" and does not "scale poorly to high dimensions", problems common observed problem with GP models [**TURBO2020**]. That is, GP tends to be good at exploring the hyper-parameter space but it does not perform well at optimizing in local regions. Turbo performs well by breaking down the global optimization problem into many smaller local optimization problems. Moreover, Turbo has been shown to be faster than other state-of-the-art black box evaluation methods (BO as well as other types of methods) and has been shown to find better global maxima. Currently, `rcTorch` library embeds Turbo-1, however, an upgrade to Turbo-m is scheduled and will make the hyper-parameter search more robust and less likely to get stuck in local maxima than Turbo-1.

Thompson Sampling: Furthermore, we use a Thompson sampling (TS) acquisition function since it has been shown (unlike many other common acquisition functions) to scale linearly in terms of complexity with increased batch size [**TURBO2020**]. `rcTorch` trains multiple RCs in parallel for each batch and larger batch sizes have shown to have better performance in combination with Turbo and TS. In particular, Turbo has been shown to work well with large batch sizes. Larger batch sizes can result in better solutions while not causing a dramatic increase overall TURBO runtime (which is a huge problem for other BO methods). In addition, TS is an efficient acquisition function for exploration which complements Turbo's relative exploitation strength. Thus, the BO used in `rcTorch` is a modern and balanced technique. The combination of TURBO and TS are excellent algorithmic choices which make `rcTorch` very efficient and hold promise for the application of `rcTorch` on more complex, higher dimensional, and potentially chaotic problems in the future. In addition, these algorithms are well suited for large numbers of hyper-parameters and more robust solutions making it likely that `rcTorch` will be able to efficiently solve problems with dozens or even hundreds of hyper-parameters in the near future.

BO cross validation: During every iteration of BO with `rcTorch` we evaluate our model based on a specified number of cross validation samples n. This number plays a role in our BO loss function in Eq. (1). Each cross validation sample corresponds to the training of one RC network with one set of hyper-parameters. First BO randomly selects the start of a subset of the training range with the `subsequence_length` argument which determines how many time points make up the cv_sample. Every `cv_sample` has a training set and a validation set immediately following the training set in sequence. This step of BO is performed to make BO more robust which has been demonstrated in [**Maat2018**]. Next the sub-sequence is split into training and validation sets with proportions determined by the `val_split` argument. If the `val_split` $\simeq 0.3$ then 70% of the sub-sequence length would be the training set and 30% would be the validation set. The weights of the output layer are optimized based on the training set and then we evaluate the RC on both the training set and the validation set. The validation set corresponds to an evaluation of the models ability to extrapolate, though this requires further study and improvement.

BO loss function: In this study we introduce an unsupervised RC where the only input is evenly spaced time points. In general performing BO for bundles of ICs was more robust than simply using one IC. The second sum of the loss function in Eq. (1) corresponds to the ICs considered to solve an ODE. It is worth noting that solving an ODE for different ICs requires one set of the hidden states, since one set of hyper-parameters is sufficient. This property reduces the computation cost since we do not need to construct the hidden states for every different IC. This allows `rcTorch` to optimize

for hyper-parameters which are robust over a range of ICs. These can be randomly sampled at every BO iteration or alternatively can a consistent list of ICs can be evaluated at every step. We selected the determinstic IC bundle method though the sampling IC method deserves further study. We can use up to m initial conditions yielding the loss function:

$$\mathcal{L}(t, N, \dot{N}) = \frac{1}{n\,m} \sum_{\text{cv}=1}^{n} \sum_{\text{IC}=1}^{m} \left(\beta \log(\mathcal{L}_{\text{train}}(t, N, \dot{N})) + (1-\beta) \log(\mathcal{L}_{\text{val}}(t, N, \dot{N})) \right), \quad (1)$$

where the BO level hyper-parameter is set to be $\beta = 0.5$, N is the RC output, and $\dot{N}$ the time derivative of the RC output.

Because we are using an unsupervised RC without having labeled data, we can evaluate it on both the training and validation sets. By contrast, when using `rcTorch` for regression (where we have labeled-data) we cannot evaluate the RC on the training set as this would lead to over-fitting.

Bayesian optimization hyper-parameter descriptions: Here, we describe the key hyper-parameters used in the BO of `rcTorch` .

`Spectral Radius`: The largest eigenvalue of the adjacency matrix governing the reservoir connections [**Ott2017**].

`Connectivity`: The proportion of reservoir connections (weights) that are non-zero. This hyper-parameter determines the sparsity of the reservoir, which is a key property which leads to the echo-state property [**Jaeger2004**].

`dt`: The discrete time step determining the step between two sequential input data-points. This directly determines the behaviour of the reservoir states. Equally-spaced input points are considered through this study.

`leaking_rate`: This hyper-parameter determines the memory of the reservoir and analogous to similar parameters observed in other leaky recurrent units such as GRUs.

`regularization`: The ridge regularization strength used when solving the linear equation for the exact solutions of the RC trainable parameters.

`n_nodes`: The number of reservoir nodes (denoted by M in the main manuscript). This hyper-parameter determines the dimensions of the adjacency matrix and the hidden states. More nodes means more expressivity and also more memory usage yielding a slower model.

`bias`: The bias term of the hidden states update.

RC Training

Gradient descent: The only layer that is trained in the RC is a linear output layer characterized by $\mathbf{W}_{\text{out}}$. Considering a loss function that consists of a mean-square-error (MSE) part and a ridge regularization, we have to minimize a convex loss function L. In the linear ODEs we showed that an exact solution for the $\mathbf{W}_{\text{out}}$ that minimizes the loss function is possible. This is not possible for nonlinear problems. However, we can use gradient descent (GD) to minimize the loss function. Since we are dealing with a convex loss function, the GD is expected to be very fast. This study provides evidence of such speed. For nonlinear ODEs where we lack an exact solution we employ `StepLR`, a pytorch learning rate scheduler to improve GD. We start with high learning rates in the range $[0.04, 0.01]$, and allow the RC to monitor the loss functions for "spikes" that exceeded some threshold ψ with typical value around 0.1. If $\psi > L_n - L_{n-1}$, where n is the iteration index in the GD and L denotes the loss function, we allow the optimizer to reduce the learning rate by a factor $\gamma = 0.1$. Because of these spikes, we keep the best weights observed over the loss period, not the weights observed at the final iteration. In addition, for the second order ODEs we introduce pytorch's `CyclicLR` with option `exp_range` with a corresponding hyper-parameter $\gamma_{\text{cyclic}} \in [0.99, 0.9999]$. When employed for 5000 iterations, the cyclic learning rate was found, based on empirical evidence, to dramatically accelerates GD and improves the quality of the final solutions. In particular, it was found to rapidly move to the convex part of the loss and help the GD to avoid local minima. This was particularly true for the non-linear oscillator system where both γ and ψ were introduced as hyper-parameters to the nonlinear ODEs during BO. At every iteration (epoch) the cyclic learning rate scheduler decays the learning rate by γ_{cyclic}. However, if the cyclic learning rate were employed

for later GD iterations (>5000) we observed that loss may explode and not recover. That is, the loss might experience dramatic spikes (on the order of 10^6) thereby destroying GD. Further investigation into the cyclic learning rate is warranted.

Elastic Net: In this study we used elastic net regularization for the non-linear oscillator. In particular, we introduced two hyper-parameters associated with elastic net: `enet_alpha` and `enet_strength`. Elastic net regularization is a combination of L1 and L2 regularization and it is common practice to use the following formula. Let ρ represent `enet_alpha` and ω represent `enet_strength`, the the loss function reads

$$L = \omega \left(\rho L_1 + (1-\rho)\, L_2\right),$$

ρ can be seen as the L1 regularization proportion of the elastic net term while $(1-\rho)$ can be seen as the L2 regularization proportion. ω is the overall strength of this term. We searched for ρ between 0 and 1 and for ω on a logarithmic scale.

Linearized RC output weights

We present a numerical exploration to show how the linearization approach discussed in the main manuscript improves the RC training during GD optimization. A linearized closed-form formula for the $\mathbf{W}_{\text{out}}$ is derived and given in the main manuscript by Eq. (31). Here, we apply the RC solver to the nonlinear bernoulli ODE (Eq. (26) in the main text). In Fig. 1 we report the loss function during GD iteration for the 4 initial conditions shown in different colors (one for each IC); we explore the same initial conditions investigated in the main manuscript. In particular, the solid lines in Fig. 1 show the loss trace when $\mathbf{W}_{\text{out}}$ are randomly initialized whereas, the dashed curves represent the GD training when the $\mathbf{W}_{\text{out}}$ are initialized by linearized formula (Eq. (31) in the main text). We observe in Fig. 1 that starting with the linearized $\mathbf{W}_{\text{out}}$ the loss functions converge faster and reach lower values (overall lower minima in the loss) than starting from random initial weights. To see evidence of the overall minima (as the Fig. 1 shows overlapping lines for many trajectories in the final iterations of training), see the bernoulli solutions notebook in the github repo.

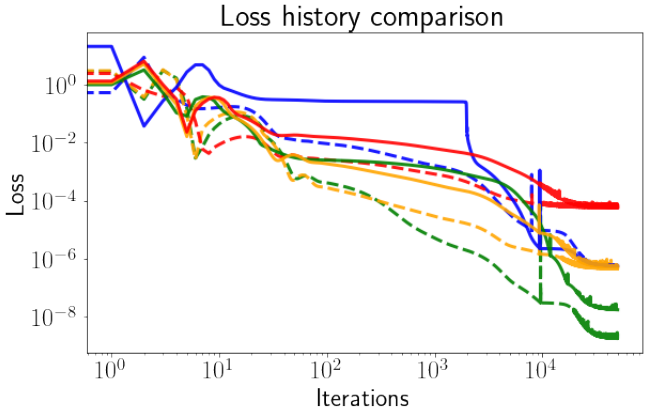

Figure 1: Loss trace during gradient descent for the nonlinear Bernoulli ODE (Eq. (26) in the main manuscript). The solid lines correspond to a randomly initialized output layer weight matrix $\mathbf{W}_{\text{out}}$; the dashed lines represent the training loss when linearized $\mathbf{W}_{\text{out}}$ are used as initial weights. Each color line corresponds to a different initial condition.

Nonlinear Oscillator

In this section, we present more details about the training and the performance of the RC solver when it is applied to the nonlinear oscillator. Figure 2 presents the results for different different ICs where each IC is represented by a different color; the thicker blue line corresponds to the only IC used in the BO. More specifically, in the left upper diagram we show the RC prediction for the phase space, namely an $x-p$ plot. The right graph represents the loss function during the GD iterations. The lower panel outlines the residuals for the RC solution solutions.

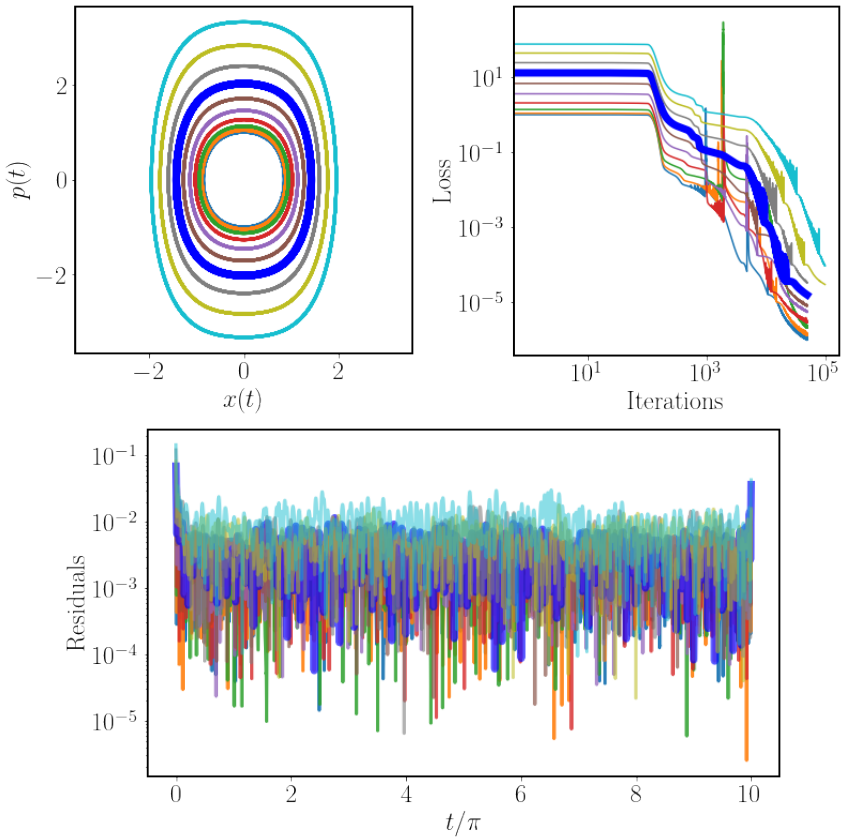

Figure 2: Caption

Comparison between RC ODE solver and forward Euler integrator

In this section we compare the numerical solution speed and accuracy obtained by the RC solver with the solutions obtained by a Euler integrator. For a fair comparison, in both solvers we used the same dt and computed 40 initial conditions in the range $[-10, 10]$; all the experiments were performed on the same machine. We explore and present the results for two first order linear ODEs. The simple population equation

$$\dot{y} + y = 0, \tag{2}$$

and the driven population equation (Eq. (23) in the main manuscript)

$$\dot{y} + y = \sin(t). \tag{3}$$

For the RC solver, we do not include the time needed for the BO, which is about 3 minutes as shown by Table 1. Moreover, since we solve linear ODEs, we do not apply GD since there is a closed-form formula for the RC training. In this context, we observe in Figs. 3 and 4 that RC solver is significantly faster than Euler integrator. "RC declare" refers to the time take to declare the esn object, build the hidden states, and solve for the first IC. "RC fit" is the time taken to solve for the remaining ICs.

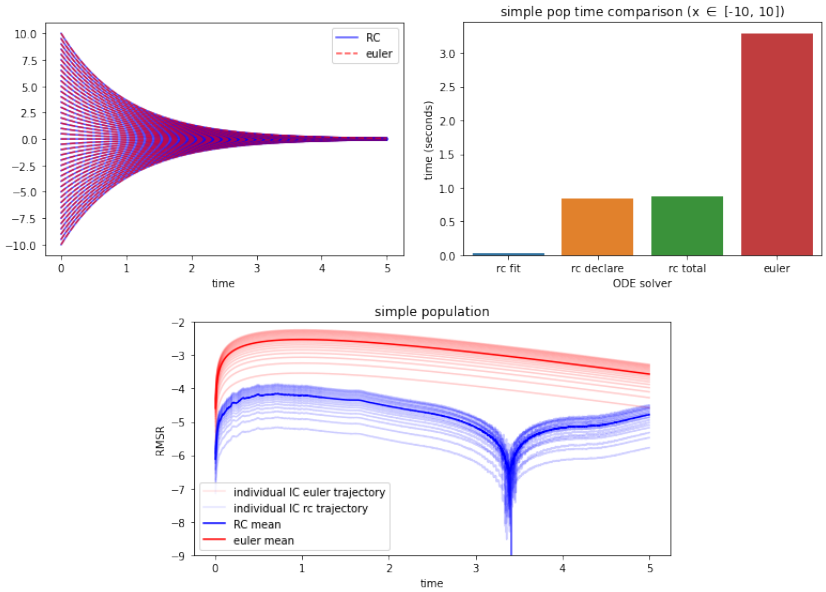

Figure 3: Comparison between RC and Euler ODE solvers one the population of Eq. (2). The top left plot outlines the predicted solutions and the right graph represents the total computational time used for each method. The lower image shows the root mean squared error of the predicted numerical solutions.

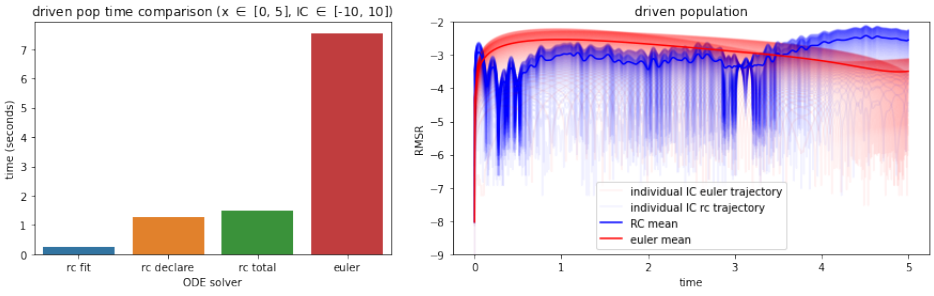

Figure 4: Comparison between RC and Euler ODE solvers one the driven population of Eq. (3). Left: The total computational time used for each method. Right: The residuals of the predicted numerical solutions.