Learning dynamic form data

Libo Chen

June 5, 2025

1 Introduction

We consider a system characterized by the differential equation

$$y' = Ay,$$

where A is a known matrix and y is the state vector of the system. We can write it as

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix},$$

$$y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix},$$

A is a 2x2 matrix, and y is a 2-dimensional vector. The elements $a_{11}, a_{12}, a_{21}, a_{22}$ represent the entries of matrix A, while y_1 and y_2 are the components of the vector y. The initial condition of the system is known and given by

$$y(0) = y_0 = (c_1, c_2),$$

in which c_1 and c_2 are known constants. Our objective is to employ machine learning algorithms to effectively predict the future states of y, leveraging the underlying dynamics encapsulated by the differential equation.

2 Regular Time Series Methods

2.1 Neural Network Approximation

To predict the state of the system in the next step, we consider the t ranges from 0 to 4, with a training set comprising eight data points. Consequently, this yields $\Delta t = 0.5714$. we can use a neural network with parameters including weight matrices W_1, W_2 and bias vectors b_1, b_2. We denote the prediction by $\hat{y}$ which can be calculated as

$$\begin{aligned} \hat{y}(\Delta t) &= W_2 \cdot (\tanh(W_1 y_0 + b_1)) + b_2 \\ \hat{y}(2\Delta t) &= W_2 \cdot (\tanh(W_1 \hat{y}(\Delta t) + b_1)) + b_2 \\ &\vdots \\ \hat{y}(8\Delta t) &= W_2 \cdot (\tanh(W_1 \hat{y}(7\Delta t) + b_1)) + b_2, \end{aligned}$$

where n can be any integer, for example, n=50
- W_1 is a $n \times 2$ weight matrix for the transformation from the input to the hidden layer,
- b_1 is a $n \times 1$ bias vector for the hidden layer,
- W_2 is a $2 \times n$ weight matrix for the transformation from the hidden layer to the output,
- b_2 is a 2×1 bias vector for the output layer.

$W_1 y_0 + b_1$ represents the weighted input to the activation function in the first layer, tanh is the hyperbolic tangent function applied element-wise to the result, and W_2 and b_2 are the weight matrix and bias vector for the output layer.

To express this in a more general form, we express this as follows

$$\hat{y}(n\Delta t) = f_{\text{out}}(f_{\text{hidden}}(\hat{y}((n-1)\Delta t; \Theta_{\text{hidden}}); \Theta_{\text{out}}).$$

Here, $\hat{y}(n\Delta t)$ is the predicted output at time $n\Delta t$, and f_{hidden} and f_{out} represent the transformation functions of the hidden and output layers, respectively. The parameters $\Theta_{\text{hidden}} = \{W_1, b_1\}$ and $\Theta_{\text{out}} = \{W_2, b_2\}$ include the weights and biases for the hidden and output layers, encapsulating the network's parameters. This implies that we want to consider a loss function which we optimise with respect to $\Theta = \{\Theta_{\text{hidden}}, \Theta_{\text{out}}\}$.

After building this model, we construct a loss function $L(\Theta)$, as

$$L(\Theta) \equiv \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}(i)|^2,$$

where y_i is the actual data in the time i, $\hat{y}(i)$ is the predicted value, and N is the number of samples. Now, we learn the parameters W^* such that it satisfies

$$\Theta^* = \min_{\Theta} L(\Theta).$$

This optimization can be done by using a RMSprop algorithm discussed below.

Algorithm 1 RMSprop Algorithm

Require: Learning rate $\eta = 1 \times 10^{-3}$, decay rate $\gamma = 0.99$, small constant $\epsilon = 1 \times 10^{-8}$

1: Initialize weight parameters W for each module in the network:
2: **for** each module m in $self.net$ **do**
3: **if** m is an instance of nn.Linear **then**
4: $m.weight \leftarrow$ Drawn from $\mathcal{N}(0, 0.01)$
5: $m.bias \leftarrow$ Set to 0
6: **end if**
7: **end for**
8: Initialize running average G=$G_{W_1}, G_{b_1}, G_{W_2}, G_{b_2}$ to zero for each weight, This will store the moving average of the squared gradients
9: **while** not converged **do**
10: Compute gradient $\nabla L(\Theta)$(Compute the gradient of the loss with respect to each parameter: $\nabla_{W_1} L, \nabla_{b_1} L, \nabla_{W_2} L, \nabla_{b_2} L$) with respect to Θ
11:

$$\begin{aligned} G_{W_1} &\leftarrow \gamma G_{W_1} + (1-\gamma)\left(\nabla_{W_1} L\right)^2 \\ G_{b_1} &\leftarrow \gamma G_{b_1} + (1-\gamma)\left(\nabla_{b_1} L\right)^2 \\ G_{W_2} &\leftarrow \gamma G_{W_2} + (1-\gamma)\left(\nabla_{W_2} L\right)^2 \\ G_{b_2} &\leftarrow \gamma G_{b_2} + (1-\gamma)\left(\nabla_{b_2} L\right)^2 \end{aligned}$$

12:

$$W_1 \leftarrow W_1 - \frac{\eta}{\sqrt{G_{W_1} + \epsilon}} \nabla_{W_1} L, b_1 \leftarrow b_1 - \frac{\eta}{\sqrt{G_{b_1+\epsilon}}} \nabla_{b_1} L$$
$$W_2 \leftarrow W_2 - \frac{\eta}{\sqrt{G_{W_2} + \epsilon}} \nabla_{W_2} L, b_2 \leftarrow b_2 - \frac{\eta}{\sqrt{G_{b_2+\epsilon}}} \nabla_{b_2} L$$

13: **end while**

2.2 Neural Network Matrix Approximation

The objective of Neural Network Matrix Approximation is clear: to develop a Neural Network. This neural network is tasked with approximating the transformation matrix. This matrix governs the evolution of a system. This system is described by a linear Ordinary Differential Equation (ODE).

Consider the linear ODE provided by Introduction:

$$y' = Ay,$$

where y represents the system's state and A is a constant matrix. The exact solution over a time interval Δt can be expressed using the matrix exponential

$$y(n + \Delta t) = By(n),$$

where B is defined as

$$B = \exp(A\Delta t) = I + A\Delta t + \frac{A^2\Delta t^2}{2!} + \frac{A^3\Delta t^3}{3!} + \dots.$$

A Neural Operator is proposed to approximate this solution, structured to model the system's state at discrete time steps, for $\Delta t = 0.5714$, we conduct training on eight data points.

$$\begin{aligned}
&\hat{y}(\Delta t) = \hat{B}y(0) \\
&\hat{y}(2\Delta t) = \hat{B}\hat{y}(\Delta t) \\
&\vdots \\
&\hat{y}(8\Delta t) = \hat{B}\hat{y}(7\Delta t),
\end{aligned}$$

where $\hat{B}$ is a learnable 2×2 matrix approximating B, and $\hat{y}(t)$ represents the neural network's prediction of the system's state at time t.

Once the Neural Operator model is established, we define the parameters of the matrix $\hat{B}$ to be learned. The parameter matrix Θ, representing $\hat{B}$, is defined as:

$$\Theta = \hat{B} = \begin{bmatrix} W_1 & W_2 \\ W_3 & W_4 \end{bmatrix},$$

where each w_i is a parameter to be optimised.

We construct the loss function $L(\Theta)$ as the Mean Squared Error (MSE) between the actual system state and the predicted state at each time step:

$$L(\Theta) = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}(i)|^2,$$

where y_i is the actual state of the system at time i, $\hat{y}(i)$ is the predicted state by the Neural Operator at the same time, and N is the total number of sampled time steps.

To find the optimal parameters Θ^* that minimize the loss function $L(\Theta)$. This is formulated as an optimization problem:

$$\Theta^* = \min_{\Theta} L(\Theta).$$

In practice, this optimization is carried out using algorithms like RMSprop algorithm or gradient descent, where the parameters Θ are iteratively adjusted to reduce the loss.

After training, we can compare the learned matrix $\hat{B}$ with the analytical matrix B. This comparison validates how accurately the neural network has learned to approximate the dynamics dictated by the ODE.

2.3 NeuralODE Approximation

In this subsection, we consider a Neural Ordinary Differential Equation (Neural ODE), where we have an initial value problem characterized by a differential equation $y' = Ay$, and a known initial state $y(0) = y_0 = (c_1, c_2)$, the Neural ODE architecture could be expressed as follows.

Let $y(t)$ be the state of our system at time t, and let $f_\theta(y(t), t)$ be a neural network parameterized by θ that approximates the derivative of y with respect to t. The Neural ODE defines the continuous transformation of y over time as

$$y'(t) = f_\theta(y(t), t).$$

Consider a discrete approximation. Given $y(0) = y_0$, we need to compute $\tilde{y}(1)$ using an ODE solver that integrates this equation from $t = 0$ to $t = 1$

$$\tilde{y}(\Delta t) = y(0) + \int_0^{\Delta t} f_\theta(y(\tau), \tau)\, d\tau;$$

similarly,

$$\tilde{y}(8\Delta t) = \tilde{y}(7\Delta t) + \int_{7*\Delta t}^{8*\Delta t} f_\theta(y(\tau), \tau)\, d\tau,$$

where $y(\tau)$ is the solution of the differential equation at each point τ within the integration interval.

In practice, f_θ could be a neural network

$$f_\theta(y(\tau), \tau) = W_2 \cdot \tanh\left(W_1 \cdot y(\tau) + b_1\right) + b_2.$$

The parameters θ consist of W_1, W_2, b_1, and b_2, and they are learned during training. The integral can be computed using numerical ODE solvers, such as the Runge-Kutta methods, which are often used in the implementation of Neural ODE.

The loss function and the optimal method is the same as Neural Operator Approximation.

2.4 NeuralODE Matrix Approximation

The NeuralODE Matrix Approximation method aims to model and predict the evolution of a system described by a linear ordinary differential equation using a simplified Neural ODE framework. The NeuralODE Matrix Approximation method adopts an architectural approach similar to the standard Neural ODE Approximation. However, it distinguishes itself by employing a linear transformation model.

In this approach, the function $f_\theta(y(t), t)$, which previously represents a more complex neural network in standard Neural ODEs, is simplified to a linear transformation

$$f_\theta(y(t), t) = My(t),$$

where M is a learnable 2×2 matrix.

3 Results and Discussions

In this section, we present numerical examples to illustrate the application of the methods proposed. The training data are generated by sampling points from a fixed matrix operator. Each point serves as an initial state. Then, using a fixed matrix operator, we advance each initial state forward in time by a fixed time step. This process yields pairs of initial and evolved states, constituting our training data. The generated trajectories provide the behavior of the system.

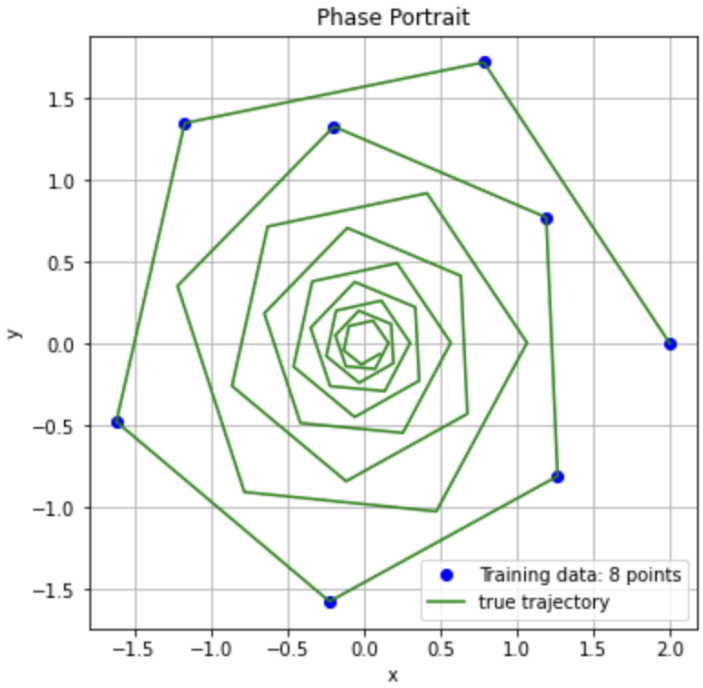

Figure 1: Train dataset consists of eight data points, with an initial point of $(2, 0)$. The time step is fixed as dt, where $t = \text{torch.linspace}(0., 4, 8)$.

3.1 Example 1 Spiral Source

Training results of four methods: All of our network models, including Neural Operator Approximation, Neural Operator Matrix Approximation, NeuralODE Approximation, and NeuralODE Matrix Approximation, are trained using the loss function defined in Section 2 and using the open-source PyTorch library. We typically train the model for 1000 epochs. For the network, All the weights are initialized randomly from Gaussian distributions and all the biases(including Matrix parameters) are initialized to be zeros.

Table 1: Training results

Name	Loss	Time
Neural Network Approximation	0.002339	1.7562
Neural Network Matrix Approximation	6.239861249923706e-06	0.8321
NeuralODE Approximation	0.0019891	2.4133
NeuralODE Matrix Approximation	2.094358205795288e-05	1.4066

Neural Network Matrix Approximation and NeuralODE Matrix Approximation achieved smaller losses compared to the other two methods, with each having only four parameter values. Below, we present the predicted matrix values with the true matrix values.

NeuralODE Matrix Approximation:

$$\text{Predicted Matrix: } \begin{bmatrix} -0.0998 & 2.0000 \\ -1.9999 & -0.0998 \end{bmatrix} \quad \rightarrow \quad \text{True Matrix: } \begin{bmatrix} -0.1 & 2 \\ -2 & -0.1 \end{bmatrix}$$

Neural Operator Matrix Approximation:

$$\text{Predicted Matrix: } \begin{bmatrix} 0.3919 & 0.8593 \\ -0.8593 & 0.3919 \end{bmatrix} \quad \rightarrow \quad \text{True Matrix: } \begin{bmatrix} 0.3919 & 0.8593 \\ -0.8593 & 0.3919 \end{bmatrix}$$

These two methods perform well, primarily due to their minimal number of parameters. Furthermore, when comparing the predicted parameter values with the true values, the differences are small. In the following examples, we will explore the reasons behind the good performance of the other two methods(Neural Network Approximation, NeuralODE Approximation) on the training set and analyze under what testing conditions they can achieve get better predictions.

Test on the different initial data: We first consider computing losses corresponding to varying initial values of a system's state variables. We iterate through a range of initial values only consider one step , sequentially computing the system trajectory using the trained models. We then compare this predicted trajectory with the truth trajectory and derive the loss metric.

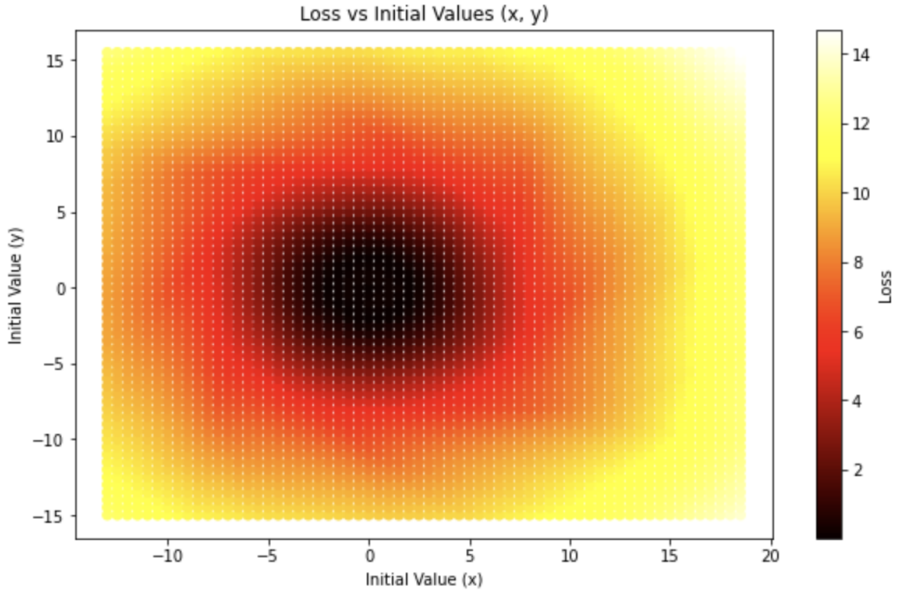

Figure 2: NeuralODE Approximation

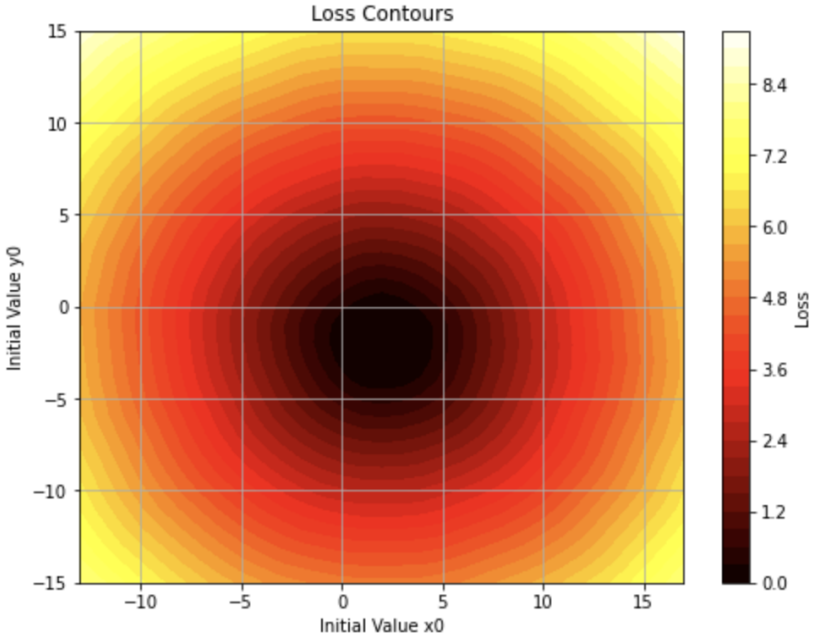

Figure 3: NeuralOperator Approximation

Remark: The loss at different initial values looks like circle, and we attempt to interpret this phenomenon from the eigenvalues and eigenvectors of the equation itself.

Complex eigenvalues: $\lambda = p + iq, \quad \bar{\lambda} = p - iq \quad (q \neq 0)$ If the eigenvector $\mathbf{v} = \mathbf{p} + i\mathbf{q}$ correspoinds to λ, then $\overline{\mathbf{v}} = \mathbf{p} - i\mathbf{q}$ is the eignevector of $\bar{\lambda}$. The general solution is $\mathbf{x}(t) = c_1 \Re\left(e^{\lambda t}\mathbf{v}\right) + c_2 \Im\left(e^{\lambda t}\mathbf{v}\right)$. Applying Euler's formula and some trigonometric identities we may write the general solution as

$$\mathbf{x}(t) = Ce^{pt}(\sin(qt - \gamma)\mathbf{p} + \cos(qt - \gamma)\mathbf{q}).$$

Through computation, we obtain:
Eigenvalues: $(-0.1 + 2i, -0.1 - 2j)$
Eigenvectors:

$$\begin{bmatrix} -0.70710678i & 0.70710678i \\ 0.70710678 & 0.70710678 \end{bmatrix}$$

The image of the eigenvectors is as shown below:

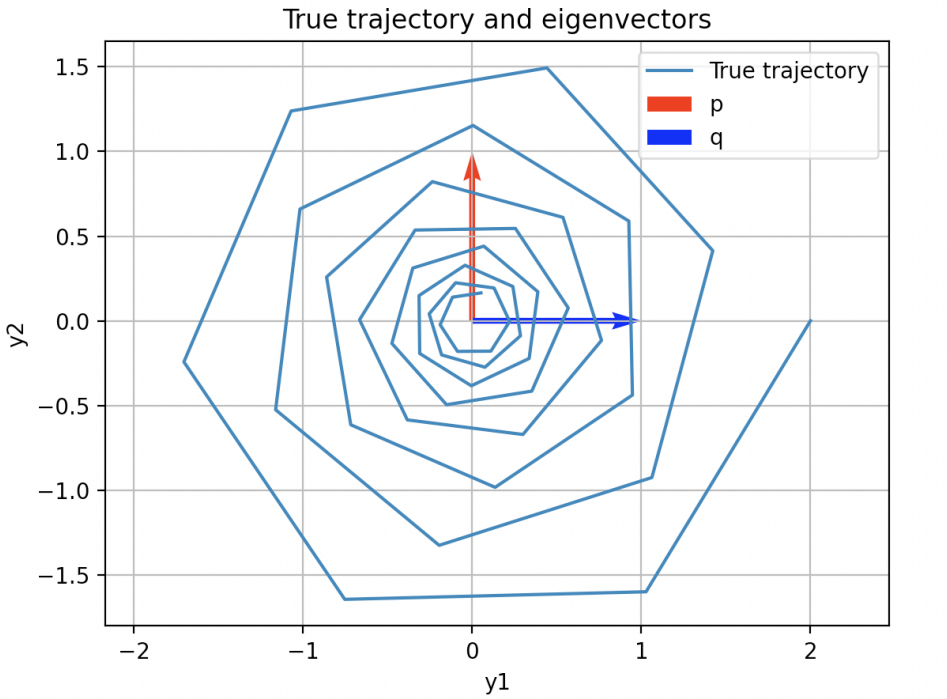

Figure 4: True trajectory and eigenvectors

Relationship y(0) with y(1): The true and predicted values are plotted against the initial values, showcasing the relationship between the initial values and the system's first state variables at the specified time step.

Remark: In our model, we utilize the tanh function. This explains why employing nonlinear methods get good results for linear equations, as the tanh function partially overlaps with the true relationship.

$$\tanh(x) \doteq \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

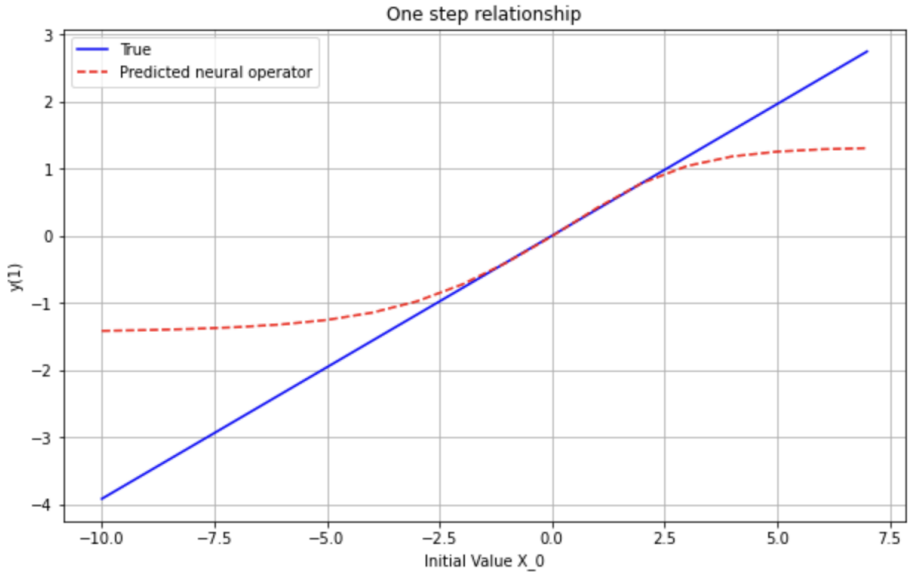

Figure 5: Neural Network Approximation Relationship : The time step is fixed as dt, where $t = \text{torch.linspace}(0., 4, 8)$.

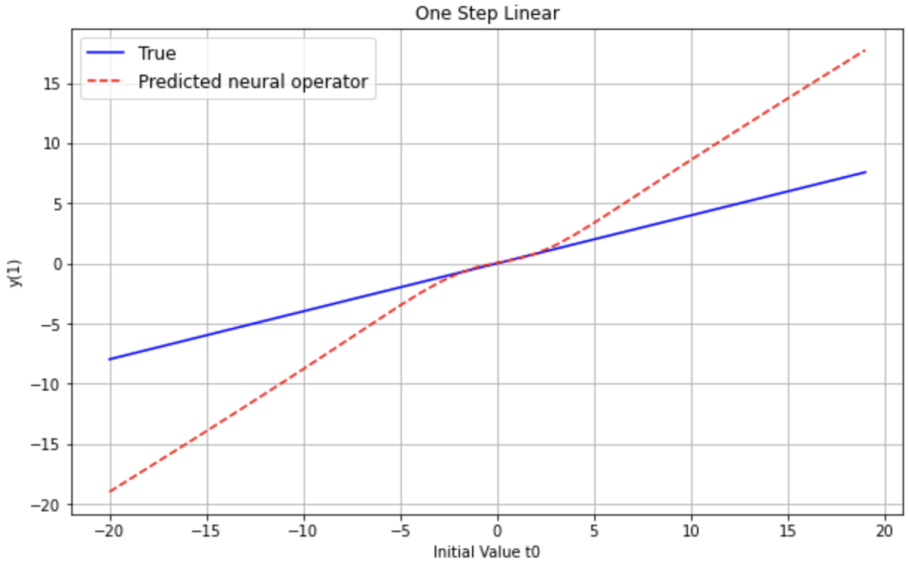

Figure 6: NeuralOde Approximation Relationship : The time step is fixed as dt, where $t = \text{torch.linspace}(0., 4, 8)$.

3.2 Example 2 Center

We now consider another linear ODE system(Tyepe: Center)

$$\begin{cases} \dot{y}_1 = -0.1y_1 + 10y_2, \\ \dot{y}_2 = -10y_1 + 0.1y_2. \end{cases}$$

The numerical results for the four trained network models are presented in the Table 2. We have set the maximum iterations is 2000, and we have also set a threshold loss. The iteration will automatically stop when the loss is less than 0.001.

Table 2: Training results

Name	Loss	Time	iterations
Neural Network (NN)	0.00081	0.89055	927
Neural Network Matrix (NNM)	0.000712	0.650988	1001
NeuralODE (NODE)	0.004210	5.34941	2000
NeuralODE Matrix (NODEM)	0.00097	2.1979	1016

Again, For the Neural Network Matrix Approximation and NeuralODE Matrix Approximation, we present the predicted matrix values with the true matrix values.

Neural Operator Matrix Approximation:

$$\text{Predicted Matrix: } \begin{bmatrix} 0.8475 & -0.5391 \\ 0.5390 & 0.8367 \end{bmatrix} \quad \rightarrow \quad \text{True Matrix: } \begin{bmatrix} 0.8477 & -0.5390 \\ 0.5390 & 0.8369 \end{bmatrix}$$

NeuralODE Matrix Approximation:

$$\text{Predicted Matrix: } \begin{bmatrix} -0.1322 & 10.0036 \\ -10.0028 & 0.0688 \end{bmatrix} \quad \rightarrow \quad \text{True Matrix: } \begin{bmatrix} -0.1 & 10 \\ -10 & 0.1 \end{bmatrix}$$

The difference between the NeuralODE Matrix Approximation's predicted outcomes and the actual results occurs because we use the RK4 method for matrix prediction. To maintain consistency, we generate the actual values using the same RK4 method. We can understand this error as the difference between the RK4 method and the real values, which can guide further exploration of RK4 method.

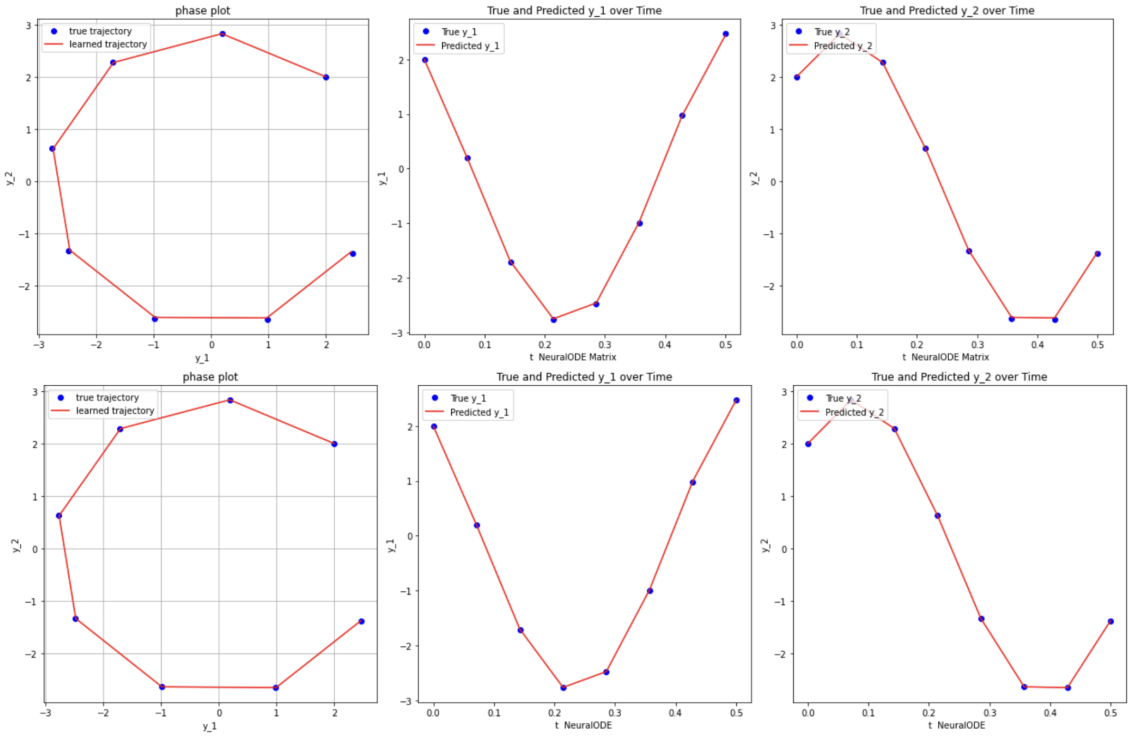

Figure 7: Trajectory and phase plots for the Example 2 with y0 = (2, 2). Top row: NueralODE Matrix; Bottom row: NueralODE.t = torch.linspace(0., 0.5, 8).

Relationship y(n) with y(n+1):

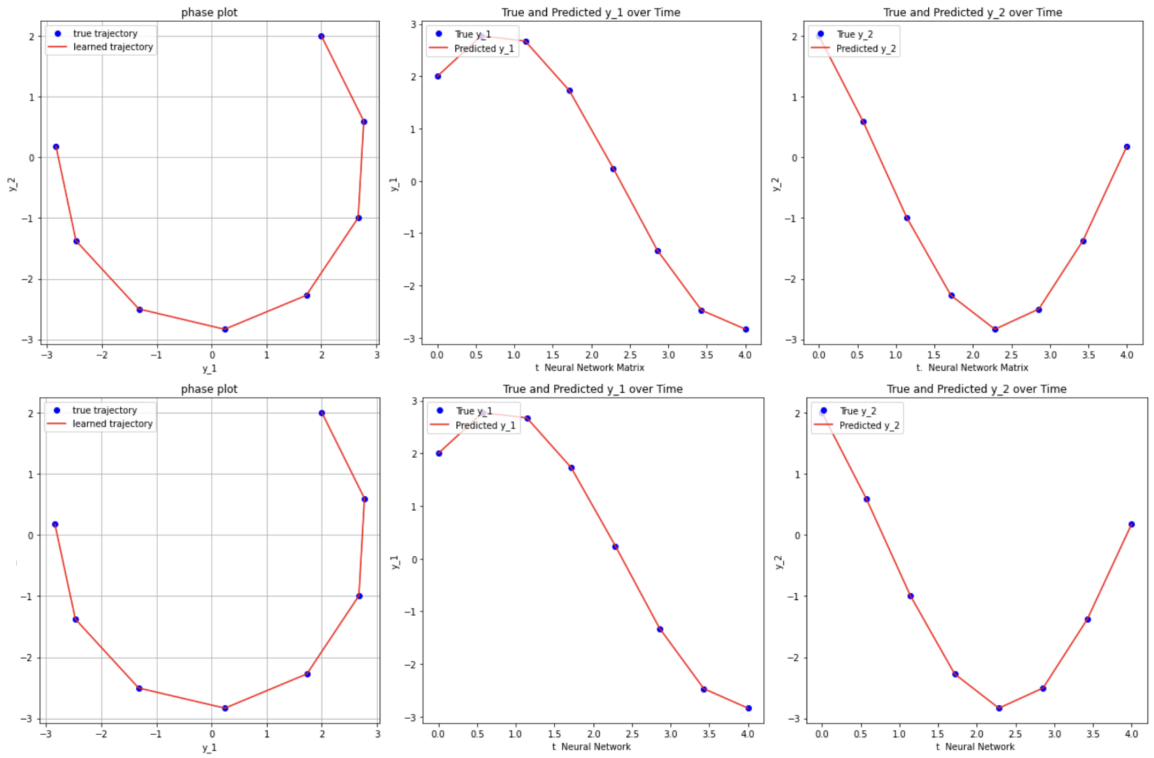

Figure 8: Trajectory and phase plots for the Example 2 with y0 = (2, 2). Top row: Nueral Network Matrix; Bottom row: Nueral Network.t = torch.linspace(0., 4, 8).

4 Irregular Time Series Methods

4.1 Recurrent neural networks

Consider a series of observations $\{y_i\}_{i=0}^{N}$ at times $\{t_i\}_{i=0}^{N}$. These points adhere to a specific ODE equation, noting that the time intervals between each point are not consistent.

To handle irregularly-timed samples with recurrent neural networks (RNN), A simple way is to integrate the time gap between observations $\Delta_t = t_i - t_{i-1}$ into the RNN's update function.

$$h_i = \text{RNNCell}\left(h_{i-1}, \Delta_t, y_{i-1}\right),$$

$$y_i = OutputNN\left(h_i\right).$$

RNNCell is the transition function of the RNN, which takes the previous hidden state h_{i-1} and the input representation y_{i-1} as input, and computes the current hidden state h_i. $OutputNN$ is an output function that maps the hidden state h_i to the output y_i at time step t_i.

4.2 Decay Recurrent neural networks

However, the Standard RNN approach raises the question of how to define the hidden state h between observations. A simple alternative introduces an exponential decay of the hidden state towards zero when no observations are made

$$h_i = \text{RNNCell}\left(h_{i-1} \cdot \exp\left\{-\tau \Delta_t\right\}, y_{i-1}\right),$$

$$y_i = OutputNN\left(h_i\right).$$

where τ is a decay rate parameter. However, experimental results indicate that the exponential decay dynamics did not enhance predictive performance compared to standard RNN approaches.

4.3 Neural Ordinary Differential Equations

We can parameterize the continuous dynamics of hidden units using an ordinary differential equation (ODE) specified by a neural network

$$\frac{dh(t)}{dt} = f(h(t), t, \theta),$$

in which the function f specifies the dynamics of the hidden state, using a neural network with parameters θ. The hidden state h(t) is defined at all times, and can be evaluated at any desired times using a numerical ODE solver:

$$h_0, \ldots, h_N = \text{ODESolve}\left(f(h(t), t, \theta), y_0, (t_0, \ldots, t_N)\right),$$

Starting from the input layer $h(0) = y(0) = y_0$, we can define the output layer h(T) to be the solution to this ODE initial value problem at some time T.

Unlike RNN, which require discretizing observation and emission intervals, continuously-defined dynamics can naturally incorporate data which arrives at arbitrary times. For irregular data, continuous Neural ODEs exhibit strong processing capabilities.

4.4 ODE-RNN

We note that an RNN with exponentially-decayed hidden state implicitly obeys the following ODE

$$\frac{dh(t)}{dt} = -\tau h.$$

With $h(t_0) = h_0$, where τ is a parameter of the model. Inspired by Neural ODEs, the hidden layer here can also be represented using a neural network. We can define the state between observations to be the solution to an ODE, and then at each observation, update the hidden state using a standard RNN update.

$$h_i' = \text{ODESolve}\left(f_\theta, h_{i-1}, (t_{i-1}, t_i)\right),$$

$$h_i = \text{RNNCell}\left(h_i', y_{i-1}\right),$$

$$y_i = OutputNN\left(h_i\right).$$

When updating the hidden statenthe Hybird model does not explicitly depend on t or Δt, but does depend on time implicitly through the resulting dynamical system. Compared to RNNs with exponential decay, The approach allows a more flexible parameterization of the dynamics.

4.5 Latent ODE-RNN

RNNs and the ODE-RNN presented above are easy to train and allow fast online predictions. However, autoregressive models can be hard to interpret, since their do not model a distribution of the data.

Latent ODE-RNN consists of three components: a Variational Autoencoder (VAE) architecture utilizing an ODE-RNN encoder and a Neural ODE decoder. VAEs consist of probabilistic versions of encoders and decoders.

Encoder: The encoder maps the input data to the parameters of a probability distribution over the latent space, typically a Gaussian distribution with mean μ and variance σ^2.

$$q\left(z_0 \mid \{y_i, t_i\}_{i=0}^N\right) = \mathcal{N}\left(\mu_{z_0}, \sigma_{z_0}\right),$$

$$\mu_{z_0}, \sigma_{z_0} = g(ODERNN_\phi\left\{y_i, t_i\right\}).$$

g is a neural network translating the final hidden state of the ODE-RNN encoder into the mean and variance of z0, ϕ represents the parameters of the encoder neural network.

Decoder: The decoder takes a latent space sample z and maps it back to the original data space. This generative process is governed by the ODE, where the initial latent state z_0 dictates the entire trajectory of the data generation process.

$$\begin{aligned} z_0 &\sim p\left(z_0\right) \\ z_0, z_1, \ldots, z_N &= \text{ODESolve}\left(f_\theta, z_0, \left(t_0, t_1, \ldots, t_N\right)\right) \\ \text{each} \quad y_i &\overset{\text{indep.}}{\sim} p\left(y_i \mid z_i\right) \quad i = 0, 1, \ldots, N. \end{aligned}$$

θ represents the parameters of the decoder neural network.

To make predictions in this model, the ODE-RNN encoder is run backwards in time to produce an approximate posterior over the initial state: $q\left(z_0 \mid \{y_i, t_i\}_{i=0}^N\right)$. Given a sample of z_0, we can find the latent state at any point of interest by solving an ODE initial-value problem. To get the approximate posterior at time point t_0, we run the ODE-RNN encoder backwards-in-time from t_N to t_0. We jointly train both the encoder and decoder by maximizing the evidence lower bound (ELBO):

$$ELBO(\theta, \phi) = \mathbb{E}_{q_\phi(z_0|\{y_i,t_i\}_{i=0}^N)}[\log p_\theta(y_0, \ldots, y_N)] - KL[q_\phi(z_0|\{y_i, t_i\}_{i=0}^N)||p(z_0)]$$

The first term measures how well the model reconstructs the input data, while the second term minimize the divergence of the approximate posterior $q_\phi(\cdot \mid y)$ from the exact posterior $p(z0)$. The KL divergence is given by:

$$KL[q_\phi(z_0|\{y_i, t_i\}_{i=0}^N)||p(z_0)] = \frac{1}{2}\sum_{i=1}^{\dim(z)}(\sigma_i^2(y) + \mu_i^2(y) - \log(\sigma_i^2(y)) - 1)$$

Latent ODE-RNN provide a principled framework for learning latent representations of data while simultaneously generating new data samples.

4.6 Latent ODE-RNN

4.7 Related literature

`https://www.sciencedirect.com/science/article/pii/S0021999119304504?via=ihub`

5 Appendix

5.1 Calculating Gradients

Simplified Neural Network.

Let's assume:

- W_1 is a single weight (scalar) for the input to the hidden layer.
- b_1 is a single bias (scalar) for the hidden layer.
- W_2 is a single weight (scalar) for the hidden layer to the output.
- b_2 is a single bias (scalar) for the output layer.

The neural network function is: $\hat{y}(t) = W_2 \cdot \tanh\left(W_1 \cdot \hat{y}(t-1) + b_1\right) + b_2$

Loss function (for simplicity, let's consider only one sample, so $N = 1$): $L = (y - \hat{y})^2$

1. Gradient with respect to W_2

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial W_2}$$
$$\frac{\partial L}{\partial \hat{y}} = 2 \cdot (y - \hat{y})$$
$$\frac{\partial \hat{y}}{\partial W_2} = \tanh\left(W_1 \cdot \hat{y}(t-1) + b_1\right),$$
$$\frac{\partial L}{\partial W_2} = 2 \cdot (y - \hat{y}) \cdot \tanh\left(W_1 \cdot \hat{y}(t-1) + b_1\right).$$

2. Gradient with respect to b_2
Similar to W_2, but $\frac{\partial \hat{y}}{\partial b_2} = 1$

$$\frac{\partial L}{\partial b_2} = 2 \cdot (y - \hat{y}).$$

3. Gradient with respect to W_1

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \tanh} \cdot \frac{\partial \tanh}{\partial \left(W_1 \cdot \hat{y}(t-1) + b_1\right)} \cdot \frac{\partial \left(W_1 \cdot \hat{y}(t-1) + b_1\right)}{\partial W_1}$$
$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh^2(x)$$
$$\frac{\partial L}{\partial W_1} = 2 \cdot (y - \hat{y}) \cdot W_2 \cdot \left(1 - \tanh^2\left(W_1 \cdot \hat{y}(t-1) + b_1\right)\right) \cdot \hat{y}(t-1).$$

4. Gradient with respect to b_1

$$\frac{\partial \left(W_1 \cdot \hat{y}(t-1) + b_1\right)}{\partial b_1} = 1,$$

$$\frac{\partial L}{\partial b_1} = 2 \cdot (y - \hat{y}) \cdot W_2 \cdot \left(1 - \tanh^2\left(W_1 \cdot \hat{y}(t-1) + b_1\right)\right).$$

Complex Neural Network.
1. Gradient with respect to W_2 $(\nabla_{W_2} L)$

$$\nabla_{W_2} L = 2 \sum_{i=1}^{N} \left(W_2 \cdot \tanh\left(W_1 \cdot \hat{y}(t-1) + b_1\right) + b_2 - y_i\right) \cdot \left(\tanh W_1 \cdot \hat{y}(t-1) + b_1^T\right).$$

2. Gradient with respect to b_2 $(\nabla_{b_2} L)$

$$\nabla_{b_2} L = 2 \sum_{i=1}^{N} \left(W_2 \cdot \tanh\left(W_1 \cdot \hat{y}(t-1) + b_1\right) + b_2 - y_i\right).$$

3. Gradient with respect to W_1 $(\nabla_{W_1} L)$

$$\nabla_{W_1} L = 2 \sum_{i=1}^{N} W_2^T \cdot \left(W_2 \cdot \tanh\left(W_1 \cdot \hat{y}(t-1) + b_1\right) + b_2 - y_i\right) \cdot \left(1 - \tanh^2\left(W_1 \cdot \hat{y}(t-1) + b_1\right)\right) \cdot \hat{y}(t-1)^T.$$

4. Gradient with respect to b_1 $(\nabla_{b_1} L)$

$$\nabla_{b_1} L = 2 \sum_{i=1}^{N} W_2^T \cdot \left(W_2 \cdot \tanh\left(W_1 \cdot \hat{y}(t-1) + b_1\right) + b_2 - y_i\right) \cdot \left(1 - \tanh^2\left(W_1 \cdot \hat{y}(t-1) + b_1\right)\right).$$