NEURAL-ODE & LATENT ODES FOR IRREGULARLY-SAMPLED TIME SERIES

NIPS 2018 (Best Paper) & NIPS 2019

박성현

DAVIAN
Data and Visual Analytics Lab

Overview

- Ordinary Differential Equation
- Numerical Methods for ODE
- Neural ODE
- Latent ODEs for Irregularly-Sampled Time Series

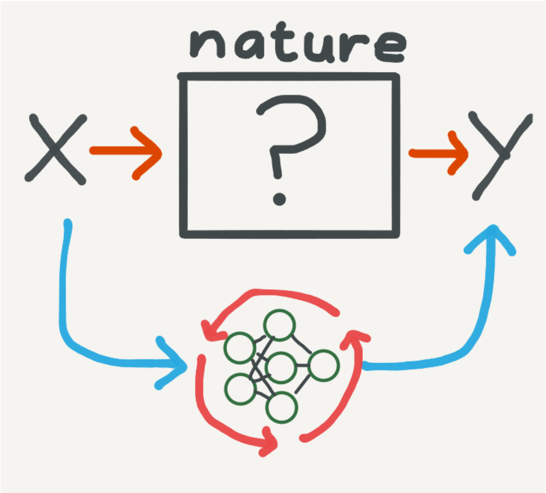
Basic Approaches

- Data-driven approaches
  - Direct method (e.g. Regression, Neural networks)

$$y = f(x) \qquad\qquad f^*(x) = a^*x + b^*$$

  - In-direct method (e.g. Ordinary differential equations)

$$\frac{dy}{dx} = f'(x) \qquad\qquad f(x) = \int f'(x)dx$$



  - If f is differentiable, what if we tried to find its derivative instead?
  - Solving the ODE is equivalent to solving the integral and can be viewed as function approximation, only here we are approximating the derivative instead.

Ordinary Differential Equation

- Ordinary Differential Equations
  - 하나의 독립 변수만을 가지고 있는 미분 방정식
  - Involve one or more ordinary derivatives of unknown functions

$$y' + 3xy = 0$$

- Partial Differential Equations
  - 여러 개의 독립 변수로 구성된 함수와 그 함수의 편미분으로 연관된 방정식
  - (ex. Navier-Stokes equation)
  - Involve one or more partial derivatives of unknown functions

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = f_i - \frac{1}{\rho}\frac{\partial p}{\partial x_i} + \nu \frac{\partial^2 u_i}{\partial x_j \partial x_j}$$

Ordinary Differential Equation

- We are interested in finding some function – called f. What has changed fundamentally, however, is that now this functions describes the rate of change – how y changes as x changes – as opposed to the direct relationship.

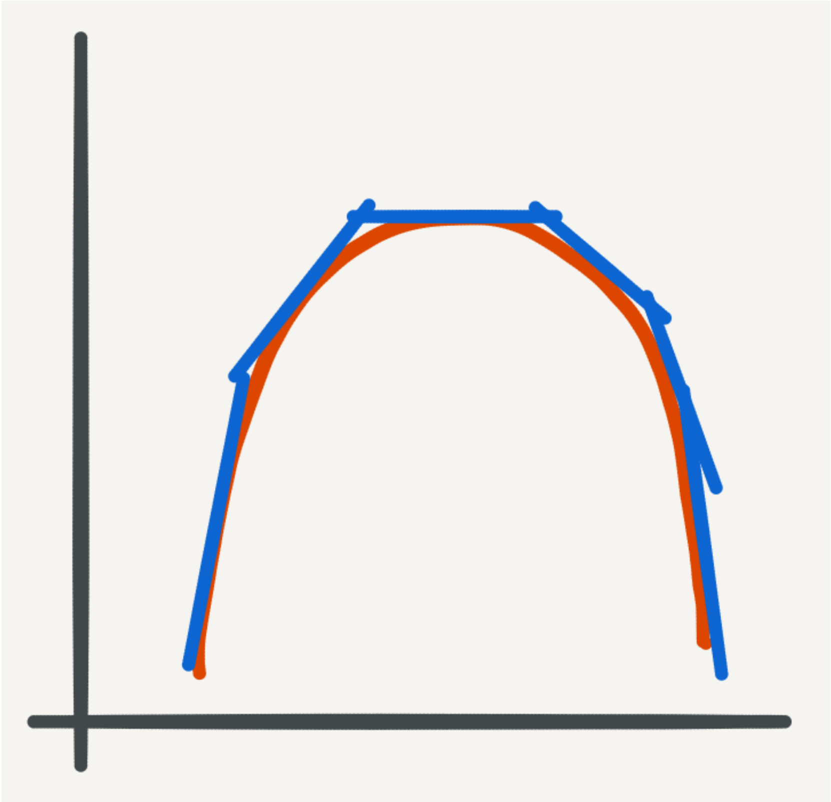
$$y'(x) = f(x, y), \quad y(x_0) = y_0.$$

- Approximating derivatives reduces the number of parameters, and also the number of functions evaluations (computation cost) required to find the optimal parameters

$$y \approx f(x) = ax + b. \qquad \frac{dy}{dx} \approx f(x, y) = a$$

Numerical Methods for ODE

- Most interesting ODE problems can't be solved analytically and require numerical methods.
- These numerical methods are implicit in that they don't analytically give you the integral but rather a set of function evaluations at future points.
- We can start at an initial point and move in the direction of the gradient evaluated at the initial point to get to a new evaluation point. Starting at this second evaluation point we can repeat the same procedure to move on to a third evaluation point, and so on. This is the basic idea behind Euler's method.



Numerical Methods for ODE

One-step Euler [edit]

A simple numerical method is Euler's method:

$$\boxed{y_{n+1} = y_n + hf(t_n, y_n).}$$

Euler's method can be viewed as an explicit multistep method for the degenerate case of one step.

This method, applied with step size $h = \frac{1}{2}$ on the problem $y' = y$, gives the following results:

$$\begin{aligned}
y_1 &= y_0 + hf(t_0, y_0) = 1 + \tfrac{1}{2} \cdot 1 = 1.5, \\
y_2 &= y_1 + hf(t_1, y_1) = 1.5 + \tfrac{1}{2} \cdot 1.5 = 2.25, \\
y_3 &= y_2 + hf(t_2, y_2) = 2.25 + \tfrac{1}{2} \cdot 2.25 = 3.375, \\
y_4 &= y_3 + hf(t_3, y_3) = 3.375 + \tfrac{1}{2} \cdot 3.375 = 5.0625.
\end{aligned}$$

Consider for an example the problem

$$y' = f(t, y) = y, \quad y(0) = 1.$$

The exact solution is $y(t) = e^t$.

Don't know the exact solution

How to approximate y

Two-step Adams–Bashforth [edit]

Euler's method is a one-step method. A simple multistep method is the two-step Adams–Bashforth method

$$\boxed{y_{n+2} = y_{n+1} + \tfrac{3}{2}hf(t_{n+1}, y_{n+1}) - \tfrac{1}{2}hf(t_n, y_n).}$$

This method needs two values, y_{n+1} and y_n, to compute the next value, y_{n+2}. However, the initial value problem provides only one value, $y_0 = 1$. One possibility to resolve this issue is to use the y_1 computed by Euler's method as the second value. With this choice, the Adams–Bashforth method yields (rounded to four digits):

$$\begin{aligned}
y_2 &= y_1 + \tfrac{3}{2}hf(t_1, y_1) - \tfrac{1}{2}hf(t_0, y_0) = 1.5 + \tfrac{3}{2} \cdot \tfrac{1}{2} \cdot 1.5 - \tfrac{1}{2} \cdot \tfrac{1}{2} \cdot 1 = 2.375, \\
y_3 &= y_2 + \tfrac{3}{2}hf(t_2, y_2) - \tfrac{1}{2}hf(t_1, y_1) = 2.375 + \tfrac{3}{2} \cdot \tfrac{1}{2} \cdot 2.375 - \tfrac{1}{2} \cdot \tfrac{1}{2} \cdot 1.5 = 3.7812, \\
y_4 &= y_3 + \tfrac{3}{2}hf(t_3, y_3) - \tfrac{1}{2}hf(t_2, y_2) = 3.7812 + \tfrac{3}{2} \cdot \tfrac{1}{2} \cdot 3.7812 - \tfrac{1}{2} \cdot \tfrac{1}{2} \cdot 2.375 = 6.0234.
\end{aligned}$$

The exact solution at $t = t_4 = 2$ is $e^2 = 7.3891\ldots$, so the two-step Adams–Bashforth method is more accurate than Euler's method. This is always the case if the step size is small enough.

Euler's Method

- By convention we'll interpret t as being a time element in the evolution of y as we iteratively use the method.

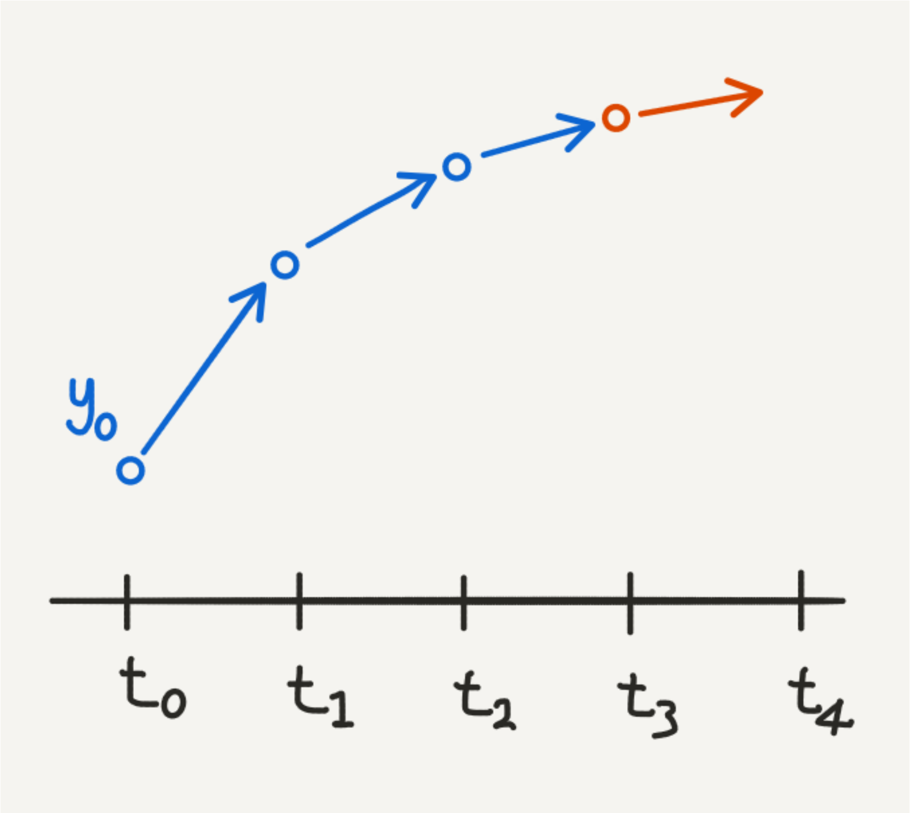
$$\frac{dy}{dt} \approx \frac{y(t+\delta) - y(t)}{\delta}$$

$$y(t+\delta) = y(t) + \delta \frac{dy}{dt}$$

- To compute approximations for y using Euler's method, we need to discretize the domains. Starting from an initial point (t_0, y_0), we define a computation trajectory recursively as follows:

$$t_{n+1} = t_n + \delta(n+1), \quad n = 0, 1, 2, \ldots$$

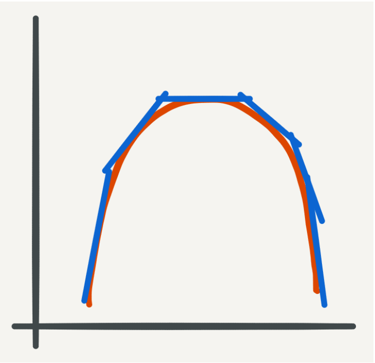
$$y_{n+1} = y_n + \delta f(t_n, y_n), \quad n = 0, 1, 2, \ldots$$



Neural ODE

From Euler's method to Neural Networks

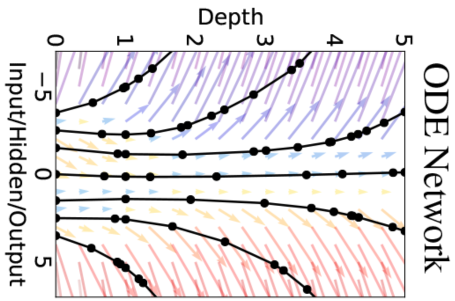
$$y(t+\delta) = y(t) + \delta f(t, y).$$

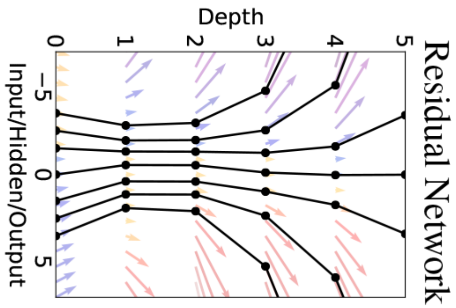


Continuous function → discretized approximation → Euler's method

$$h_{t+1} = h_t + f(h_t, \theta_t).$$

Resblock (neural net) → make it continuous → Continuous neural net





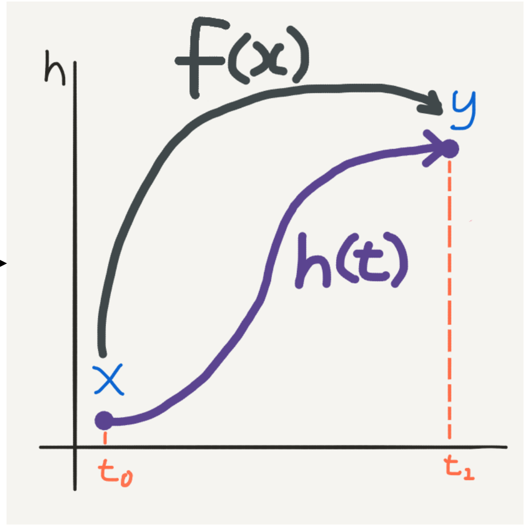
Neural ODE

- Resnet formula

$$h_{t+1} = h_t + f(h_t, \theta_t).$$

- We can rewrite it as follows:

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



- By taking an integral on both side, we have

$$h(t) = \int f(t, h(t), \theta_t) dt.$$

- Integral can be approximated by *numerical ODE* (e.g. Euler's method)

Neural ODE

- Final formula

initial time, end time

$$\hat{y} = h(t_1) = ODESolve(h(t_0), \boxed{t_0}, \boxed{t_1}, \boxed{\theta}, \boxed{f})$$

nn parameter, nn

- Backpropagation

$$\mathscr{L}(t_0, t_1, \theta_t) = \mathscr{L}\Big(ODESolve(h(t_0), t_0, t_1, \theta, f)\Big)$$

Neural ODE

- Backpropagation

$$\mathscr{L}(t_0, t_1, \theta_t) = \mathscr{L}\Big(ODESolve\big(h(t_0), t_0, t_1, \theta, f\big)\Big)$$

- (1) Define adjoint state (how the loss depends on the hidden state)

$$a(t) = -\frac{\partial \mathscr{L}}{\partial h(t)}.$$

- (2) Its time derivative is

$$\frac{da(t)}{dt} = -a(t)^T \frac{\partial f(t, h(t), \theta_t)}{\partial h(t)}.$$

- (3) Then the adjoint state can be obtained via integral *(also solved by ODE Solver)*

$$\frac{\partial \mathscr{L}}{\partial h(t)} = a(t) = \int -a(t)^T \frac{\partial f(t, h(t), \theta_t)}{\partial h(t)} dt$$

Neural ODE – When Naïve backpropagation fails?

- For example we want to integrate dynamical systems through **1M timesteps**, this would correspond to roughly 1M layer NN, so we will end up with memory issue
- **Memory issues arise** because we **need to store all activations** in the graph and higher order solvers even more activations
- Backpropagation through adaptive solvers maybe infeasible due to numerical errors, instability or just non-differentiability of the solver

Neural ODE – Adjoint Method

- **Adjoint method** can be understand as a continuous version of chain rule
- Chain rule:

$$\mathbf{h}_{t+1} = f(\mathbf{h}_t) \qquad \mathcal{L} = \mathcal{L}(\mathbf{h}_{t+1})$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}$$

- We are interested in continuous change in hidden state:

$$\mathbf{h}(t+\varepsilon) = \mathbf{h}(t) + \int_t^{t+\varepsilon} f(\mathbf{h}(t'), t')\, dt' \quad \text{since} \quad \frac{\mathrm{d}\mathbf{h}(\mathrm{t})}{\mathrm{dt}} = \mathrm{f}(\mathbf{h}(\mathrm{t}), \mathrm{t})$$

- Same **chain rule** can be applied

$$\frac{\partial L}{\partial \mathbf{h}(t)} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}(t+\varepsilon)} \frac{\partial \mathbf{h}(t+\varepsilon)}{\partial \mathbf{h}(t)}$$

- **Adjoint state** is defined as:

$$\mathbf{a}(t) = \frac{\partial L}{\partial \mathbf{h}(t)}$$

Neural ODE – Adjoint Method

- An continuous change in the hidden state (1):

$$\mathbf{h}(t+\varepsilon) = \mathbf{h}(t) + \int_t^{t+\varepsilon} f(\mathbf{h}(t'), t')\, dt' \quad \text{since} \quad \frac{\mathrm{d}\mathbf{h}(\mathrm{t})}{\mathrm{dt}} = \mathrm{f}(\mathbf{h}(\mathrm{t}), \mathrm{t})$$

- Continuous chain rule can be applied (2): $$\frac{\partial L}{\partial \mathbf{h}(t)} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}(t+\varepsilon)} \frac{\partial \mathbf{h}(t+\varepsilon)}{\partial \mathbf{h}(t)}$$

- Adjoint state is defined as (3): $$\mathbf{a}(t) = \frac{\partial L}{\partial \mathbf{h}(t)}$$

- From Eq.(2) and Eq.(3) we get (4): $$\mathbf{a}(t) = \mathbf{a}(t+\varepsilon) \frac{\partial \mathbf{h}(t+\varepsilon)}{\partial \mathbf{h}(t)}$$

- By combining all equations above we can derive differential equation which describes dynamics of adjoint state:

$$\boxed{\frac{d\mathbf{a}(t)}{dt}} = \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t)}{\varepsilon} = \boxed{-\mathbf{a}(t) \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)}}$$

should be **h** instead of **z**

Neural ODE – Adjoint State Proof

$$\frac{dL}{\partial \mathbf{z}(t)} = \frac{dL}{d\mathbf{z}(t+\varepsilon)} \frac{d\mathbf{z}(t+\varepsilon)}{d\mathbf{z}(t)} \quad \text{or} \quad \mathbf{a}(t) = \mathbf{a}(t+\varepsilon) \frac{\partial T_\varepsilon(\mathbf{z}(t), t)}{\partial \mathbf{z}(t)} \tag{38}$$

The proof of (35) follows from the definition of derivative:

$$\boxed{\frac{d\mathbf{a}(t)}{dt}} = \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t)}{\varepsilon} \tag{39}$$

$$= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon) \frac{\partial}{\partial \mathbf{z}(t)} T_\varepsilon(\mathbf{z}(t))}{\varepsilon} \quad \text{(by Eq 38)} \tag{40}$$

$$= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon) \frac{\partial}{\partial \mathbf{z}(t)} \left(\mathbf{z}(t) + \varepsilon f(\mathbf{z}(t), t, \theta) + \mathcal{O}(\varepsilon^2)\right)}{\varepsilon} \quad \text{(Taylor series around } \mathbf{z}(t)\text{)} \tag{41}$$

$$= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon) \left(I + \varepsilon \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + \mathcal{O}(\varepsilon^2)\right)}{\varepsilon} \tag{42}$$

$$= \lim_{\varepsilon \to 0^+} \frac{-\varepsilon \mathbf{a}(t+\varepsilon) \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + \mathcal{O}(\varepsilon^2)}{\varepsilon} \tag{43}$$

$$= \lim_{\varepsilon \to 0^+} -\mathbf{a}(t+\varepsilon) \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + \mathcal{O}(\varepsilon) \tag{44}$$

$$= \boxed{-\mathbf{a}(t) \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)}} \tag{45}$$

Neural ODE – Adjoint Method

- Final formulas:

$$\frac{\mathrm{d}\mathbf{h}(\mathrm{t})}{\mathrm{dt}} = \mathrm{f}(\mathbf{h}(\mathrm{t}), \mathrm{t})$$

[Forward]

$$\mathbf{a}(t) = \frac{\partial L}{\partial \mathbf{h}(t)}$$

[Adjoint state]

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)\frac{\partial f(\mathbf{h}(t), t)}{\partial \mathbf{h}(t)}$$

[Adjoint state]

- Why adjoint state is important and useful?
  - When doing backpropagation we need to compute two quantities $\frac{\partial L}{\partial \mathbf{h}(t_0)}$ $\frac{\partial L}{\partial \theta}$
  - The first one is actually an **adjoint state at $a(t=0)$**
  - We know adjoint state at a$(t = t_{end})$ since this is just a gradient of loss w.r.t. final hidden state
  - We can use adjoint state dynamics equation and integrate it to find $a(t=0)$
  - **This can be done using exactly the same solver applied to forward dynamics**

$$\mathbf{a}(t=0) = \mathbf{a}(t=t_{\text{end}}) - \int_{t=t_{\text{end}}}^{t=0} dt' \mathbf{a}(t') \frac{\partial f(\mathbf{h}(t'), t')}{\partial \mathbf{h}(t')}$$

Neural ODE

Full algorithm of ODE:

Algorithm 1 Reverse-mode derivative of an ODE initial value problem

Input: dynamics parameters θ, start time t_0, stop time t_1, final state $\mathbf{z}(t_1)$, loss gradient ${}^{\partial L}/_{\partial \mathbf{z}(t_1)}$

$s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}_{|\theta|}]$ ▷ Define initial augmented state

def aug_dynamics($[\mathbf{z}(t), \mathbf{a}(t), \cdot], t, \theta$): ▷ Define dynamics on augmented state

return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^\intercal \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^\intercal \frac{\partial f}{\partial \theta}]$ ▷ Compute vector-Jacobian products

$[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}] = \text{ODESolve}(s_0, \text{aug\_dynamics}, t_1, t_0, \theta)$ ▷ Solve reverse-time ODE

return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}$ ▷ Return gradients

Neural ODE

Time-series model:

$$\begin{aligned} \mathbf{z}_{t_0} &\sim p(\mathbf{z}_{t_0}) \\ \mathbf{z}_{t_1}, \mathbf{z}_{t_2}, \ldots, \mathbf{z}_{t_N} &= \text{ODESolve}(\mathbf{z}_{t_0}, f, \theta_f, t_0, \ldots, t_N) \\ \text{each} \quad \mathbf{x}_{t_i} &\sim p(\mathbf{x}|\mathbf{z}_{t_i}, \theta_\mathbf{x}) \end{aligned}$$

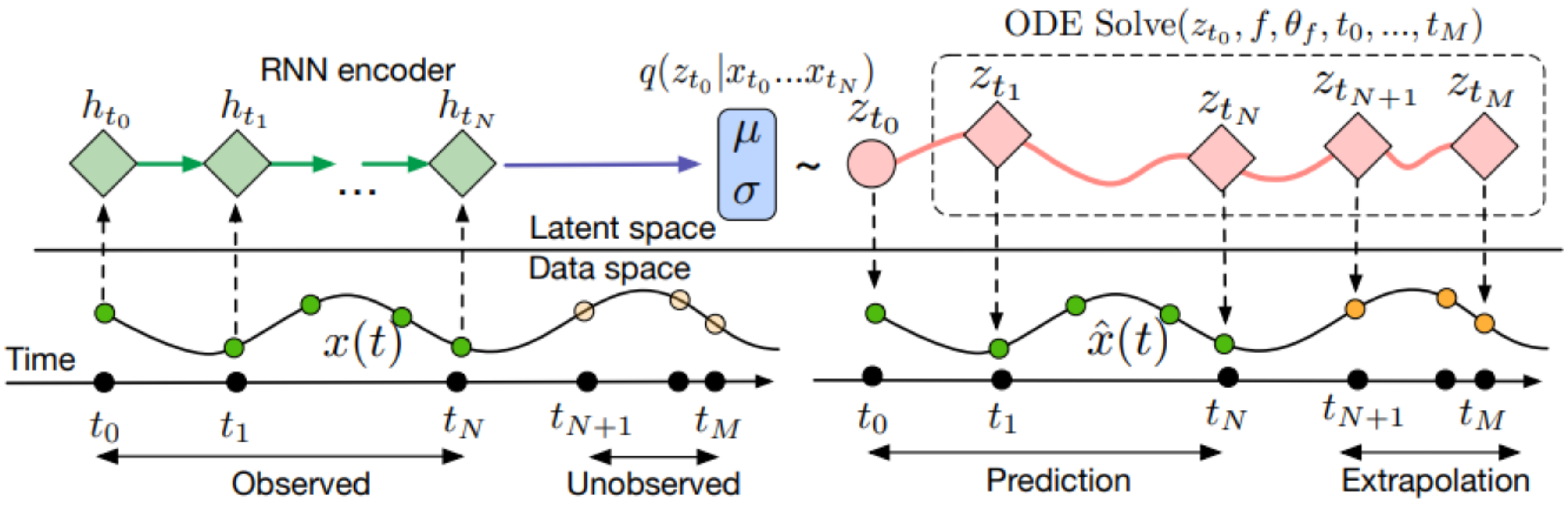


Figure 6: Computation graph of the latent ODE model.

Time-series model:

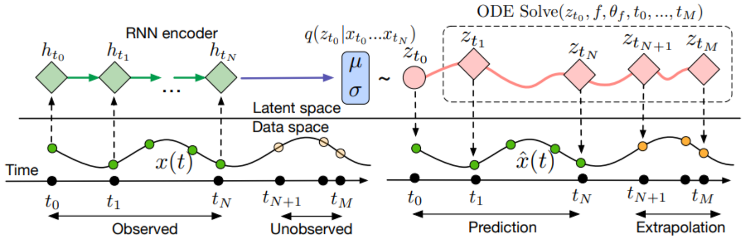


Figure 6: Computation graph of the latent ODE model.

Appendix E Algorithm for training the latent ODE model

To obtain the latent representation $\mathbf{z}_{t_0}$, we traverse the sequence using RNN and obtain parameters of distribution $q(\mathbf{z}_{t_0}|\{\mathbf{x}_{t_i}, t_i\}_i, \theta_{enc})$. The algorithm follows a standard VAE algorithm with an RNN variational posterior and an ODESolve model:

1. Run an RNN encoder through the time series and infer the parameters for a posterior over $\mathbf{z}_{t_0}$:

$$q(\mathbf{z}_{t_0}|\{\mathbf{x}_{t_i}, t_i\}_i, \phi) = \mathcal{N}(\mathbf{z}_{t_0}|\mu_{\mathbf{z}_{t_0}}, \sigma_{\mathbf{z}_0}), \tag{53}$$

   where $\mu_{\mathbf{z}_0}, \sigma_{\mathbf{z}_0}$ comes from hidden state of RNN($\{\mathbf{x}_{t_i}, t_i\}_i, \phi$)

2. Sample $\mathbf{z}_{t_0} \sim q(\mathbf{z}_{t_0}|\{\mathbf{x}_{t_i}, t_i\}_i)$
3. Obtain $\mathbf{z}_{t_1}, \mathbf{z}_{t_2}, \dots, \mathbf{z}_{t_M}$ by solving ODE ODESolve($\mathbf{z}_{t_0}, f, \theta_f, t_0, \dots, t_M$), where f is the function defining the gradient $d\mathbf{z}/dt$ as a function of $\mathbf{z}$
4. Maximize ELBO $= \sum_{i=1}^{M} \log p(\mathbf{x}_{t_i}|\mathbf{z}_{t_i}, \theta_\mathbf{x}) + \log p(\mathbf{z}_{t_0}) - \log q(\mathbf{z}_{t_0}|\{\mathbf{x}_{t_i}, t_i\}_i, \phi)$, where $p(\mathbf{z}_{t_0}) = \mathcal{N}(0, 1)$

Latent ODEs for Irregularly-Sampled Time Series

ODE-RNN:

Algorithm 1 The ODE-RNN. The only difference, highlighted in blue, from standard RNNs is that the pre-activations h' evolve according to an ODE between observations, instead of being fixed.

Input: Data points and their timestamps $\{(x_i, t_i)\}_{i=1..N}$
$h_0 = \mathbf{0}$
for i in 1, 2, . . . , N **do**
  $h'_i = \text{ODESolve}(f_\theta, h_{i-1}, (t_{i-1}, t_i))$ ▷ Solve ODE to get state at t_i
  $h_i = \text{RNNCell}(h'_i, x_i)$ ▷ Update hidden state given current observation x_i
end for
$o_i = \text{OutputNN}(h_i)$ for all $i = 1..N$
Return: $\{o_i\}_{i=1..N}; h_N$

Latent ODEs for Irregularly-Sampled Time Series

Latent ODEs:

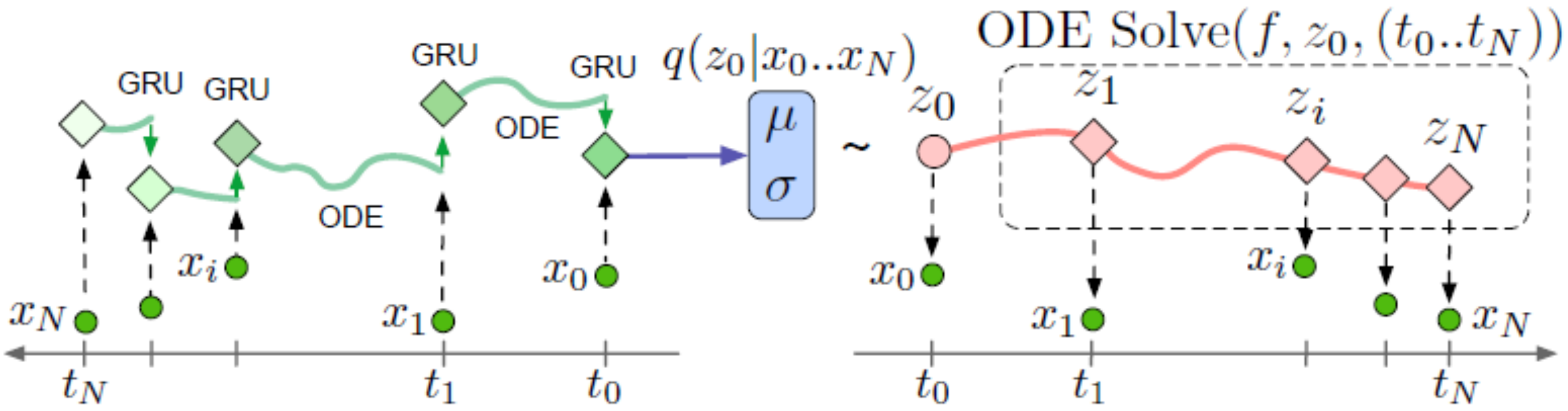


Figure 2: The Latent ODE model with an ODE-RNN encoder. To make predictions in this model, the ODE-RNN encoder is run backwards in time to produce an approximate posterior over the initial state: $q(z_0|\{x_i, t_i\}_{i=0}^N)$. Given a sample of z_0, we can find the latent state at any point of interest by solving an ODE initial-value problem. Figure adapted from Chen et al. [2018].

$$\mathrm{ELBO}(\theta, \phi) = \mathbb{E}_{z_0 \sim q_\phi(z_0|\{x_i, t_i\}_{i=0}^N)} \left[\log p_\theta(x_0, \ldots, x_N))\right] - \mathrm{KL}\left[q_\phi(z_0|\{x_i, t_i\}_{i=0}^N)||p(z_0)\right]$$

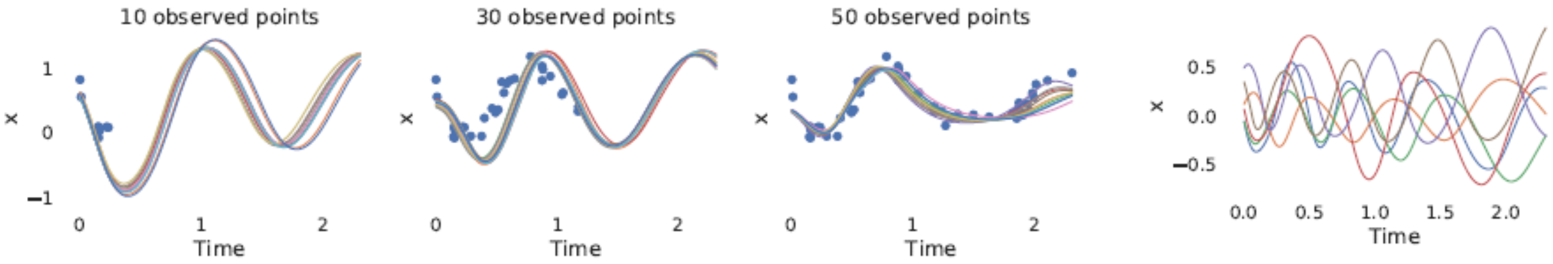
When should you use an ODE-based model over a standard RNN

- Standard RNNs are ignore the time gaps between points. As such, standard RNNs work well on regularly spaced data, with few missing values, or when the time intervals between points are short.
- ODE-RNNs can be used on sparse and/or irregular data without making strong assumptions about the dynamics of the time series.

Experiments

Toy Dataset

- 1,000 periodic trajectories with variable frequency and the same amplitude.
- Sample the initial point from a standard Gaussian, and added Gaussian noise to the observations.
- Each trajectory has 100 irregularly-sampled time points.
- During training, subsample a fixed number of points at random, and attempt to reconstruct the full set of 100 points.



(a) Conditioning on increasing number of observations (b) Prior samples

Figure 4: (a) A Latent ODE model conditioned on a small subset of points. This model, trained on exactly 30 observations per time series, still correctly extrapolates when more observations are provided. (b) Trajectories sampled from the prior $p(z_0) \sim \mathrm{Normal}(z_0; 0, I)$ of the trained model, then decoded into observation space.

Experiments

Figure 5: **(a)** Approximate posterior samples from a Latent ODE trained with an RNN recognition network, as in Chen et al. [2018]. **(b)** Approximate posterior samples from a Latent ODE trained with an ODE-RNN recognition network (ours). At training time, the Latent ODE conditions on points in red area, and reconstruct points in blue area. At test time, we condition the model on 20 points in red area, and solve the generative ODE on a larger time interval.

Experiments

MuJoCo Physics Simulation:

Table 3: Test Mean Squared Error (MSE) ($\times 10^{-2}$) on the MuJoCo dataset.

	Model	Interpolation (% Observed Pts.) 10%	20%	30%	50%	Extrapolation (% Observed Pts.) 10%	20%	30%	50%
Autoreg	RNN Δ_t	2.454	1.714	1.250	0.785	**7.259**	**6.792**	**6.594**	30.571
	RNN GRU-D	1.968	1.421	1.134	0.748	38.130	20.041	13.049	**5.833**
	ODE-RNN (Ours)	**1.647**	**1.209**	**0.986**	**0.665**	13.508	31.950	15.465	26.463
Enc-Dec	RNN-VAE	6.514	6.408	6.305	6.100	2.378	2.135	2.021	1.782
	Latent ODE (RNN enc.)	2.477	0.578	2.768	0.447	1.663	1.653	1.485	1.377
	Latent ODE (ODE enc, ours)	**0.360**	**0.295**	**0.300**	**0.285**	**1.441**	**1.400**	**1.175**	**1.258**

Experiments

MuJoCo Physics Simulation:

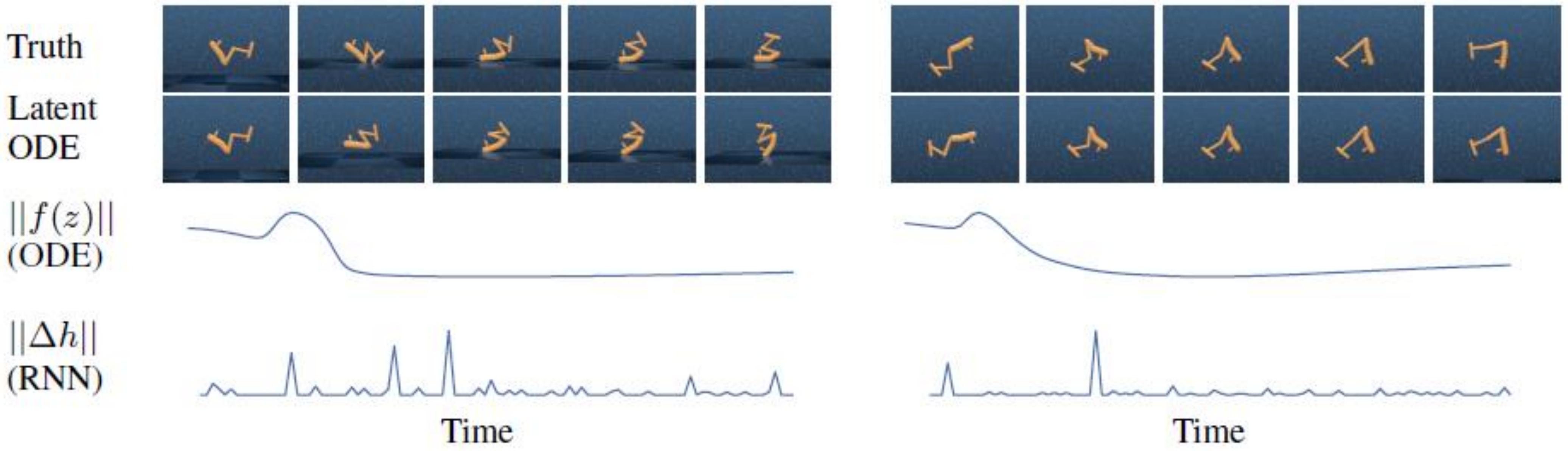


Figure 6: *Top row:* True trajectories from MuJoCo dataset. *Second row:* Trajectories reconstructed by a latent ODE model. *Third row:* Norm of the dynamics function f_θ in the latent space of the latent ODE model. *Fourth row:* Norm of the hidden state of a RNN trained on the same dataset.

Experiments

Physionet & Human Activity dataset:

Table 4: Test MSE (mean ± std) on PhysioNet. **Autoregressive** models.

Model	Interp ($\times 10^{-3}$)
RNN Δ_t	3.520 ± 0.276
RNN-Impute	3.243 ± 0.275
RNN-Decay	3.215 ± 0.276
RNN GRU-D	3.384 ± 0.274
ODE-RNN (Ours)	$\mathbf{2.361 \pm 0.086}$

Table 5: Test MSE (mean ± std) on PhysioNet. **Encoder-decoder** models.

Model	Interp ($\times 10^{-3}$)	Extrap ($\times 10^{-3}$)
RNN-VAE	5.930 ± 0.249	3.055 ± 0.145
Latent ODE (RNN enc.)	3.907 ± 0.252	3.162 ± 0.052
Latent ODE (ODE enc)	$\mathbf{2.118 \pm 0.271}$	$\mathbf{2.231 \pm 0.029}$
Latent ODE + Poisson	2.789 ± 0.771	$\mathbf{2.208 \pm 0.050}$

Table 6: **Per-sequence classification.** AUC on Physionet.

Method	AUC
RNN Δ_t	0.787 ± 0.014
RNN-Impute	0.764 ± 0.016
RNN-Decay	0.807 ± 0.003
RNN GRU-D	$\mathbf{0.818 \pm 0.008}$
RNN-VAE	0.515 ± 0.040
Latent ODE (RNN enc.)	0.781 ± 0.018
ODE-RNN	$\mathbf{0.833 \pm 0.009}$
Latent ODE (ODE enc)	$\mathbf{0.829 \pm 0.004}$
Latent ODE + Poisson	$\mathbf{0.826 \pm 0.007}$

Table 7: **Per-time-point classification.** Accuracy on Human Activity.

Method	Accuracy
RNN Δ_t	0.797 ± 0.003
RNN-Impute	0.795 ± 0.008
RNN-Decay	0.800 ± 0.010
RNN GRU-D	0.806 ± 0.007
RNN-VAE	0.343 ± 0.040
Latent ODE (RNN enc.)	0.835 ± 0.010
ODE-RNN	0.829 ± 0.016
Latent ODE (ODE enc)	$\mathbf{0.846 \pm 0.013}$

Thank you!