Neural Ordinary Differential Equations

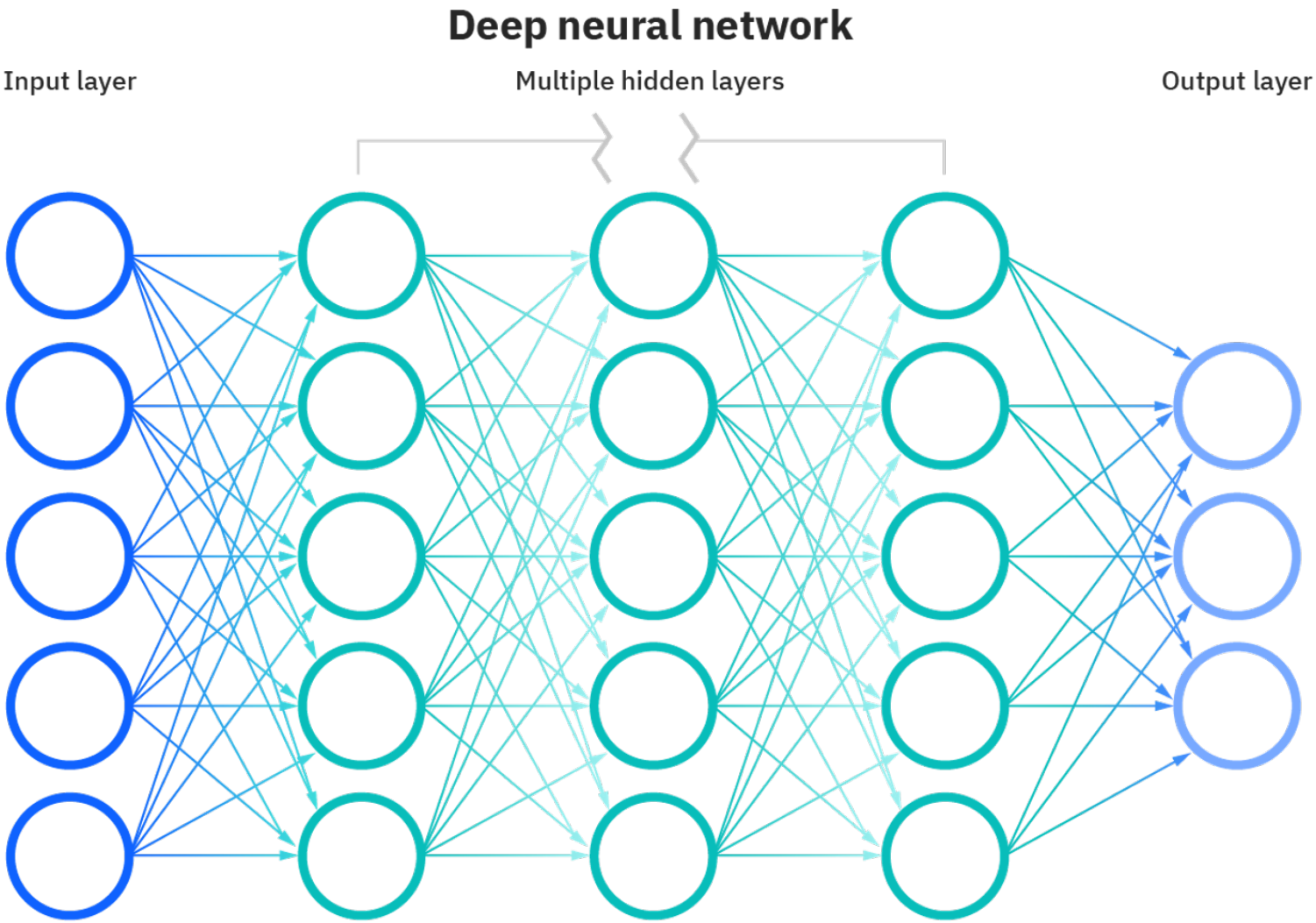
Yukang Jiang

Sun Yat-sen University

November 30, 2022

Outline

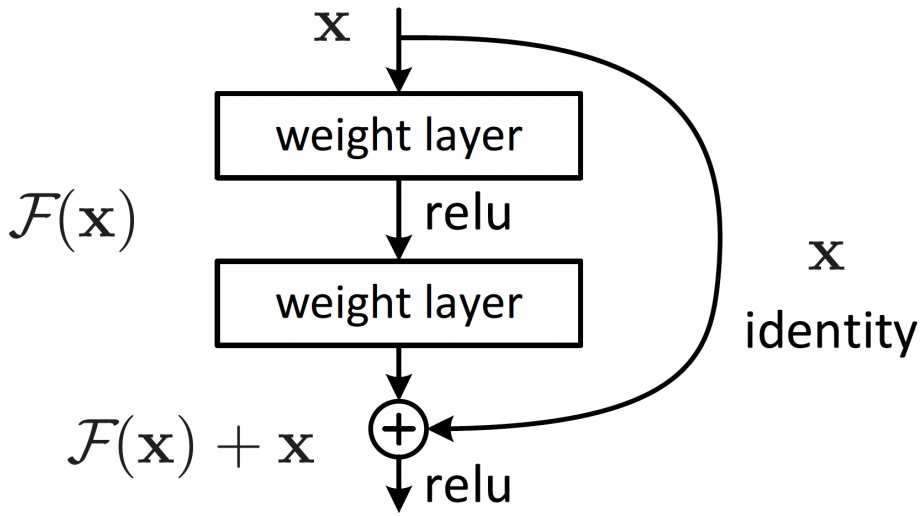
Deep Neural Network



Residual Neural Networks

Deep residual learning framework

- Denoting the desired underlying mapping as $\mathcal{H}(\mathbf{x})$.
- Let the stacked nonlinear layers fit another mapping of $\mathcal{F}(\mathbf{x}) := \mathcal{H}(\mathbf{x}) - \mathbf{x}$.
- The original mapping is recast into $\mathcal{F}(\mathbf{x}) + \mathbf{x}$.
- The formulation of $\mathcal{F}(\mathbf{x}) + \mathbf{x}$ can be realized by feedforward neural networks with **"shortcut connections".**

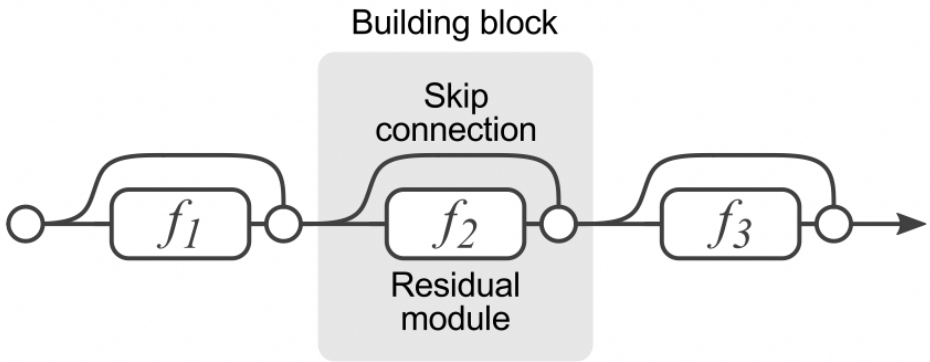


Discretization

- Transformations by composing a sequence of transformations to a hidden state:

$$\mathbf{z}_{t+1} = \mathbf{z}_t + f\left(\mathbf{z}_t, \theta_t\right),$$

where $t \in \{0 \ldots N\}$ and $\mathbf{z}_t \in \mathbb{R}^D$.



What happens as we add more layers and take **smaller steps?**

Neural Ordinary Differential Equations (NODE)

- One of the best research papers of NeurIPS 2018.
- Presents a continuous model.
- Solves the ODE with a black-box ODE solver.

Neural ordinary differential equations

RTQ Chen, Y Rubanova... - Advances in **neural** ..., 2018 - proceedings.neurips.cc

... We introduce a new family of deep **neural** network models. Instead of specifying a discrete ... **neural** network. The output of the network is computed using a blackbox **differential equation** ...

☆ Save 99 Cite Cited by 2664 Related articles All 23 versions »

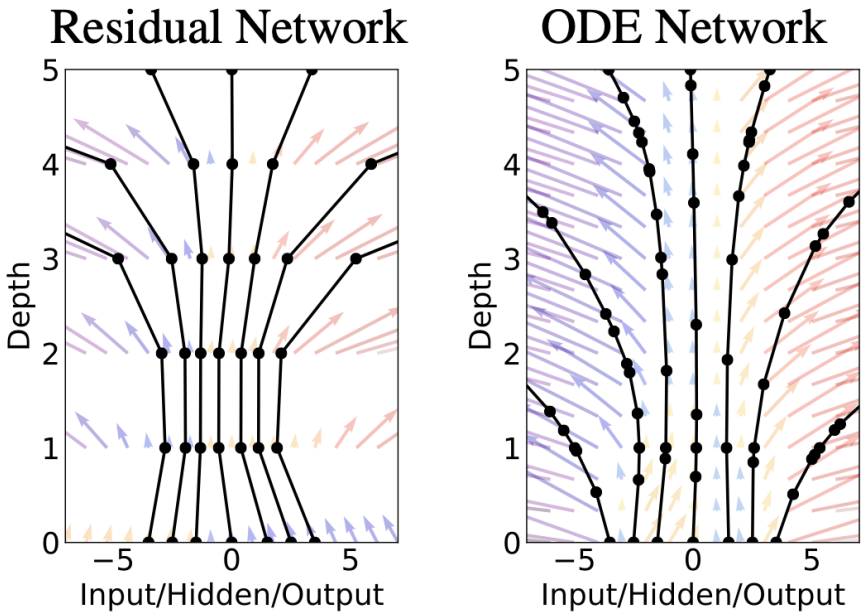
Chen R T Q, Rubanova Y, Bettencourt J, et al. Neural ordinary differential equations[J]. *Advances in neural information processing systems*, 2018, 31.

NODE

- Parameterize the continuous dynamics of hidden units using an ordinary differential equation (ODE):

$$\frac{d\mathbf{z}(t)}{dt} = f(\mathbf{z}(t), t, \theta).$$

- The input layer $\mathbf{z}(0)$ and the output layer $\mathbf{z}(T)$.



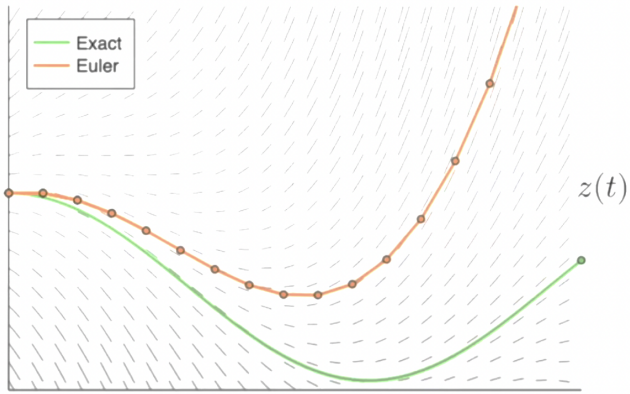
Background: ODE Solvers

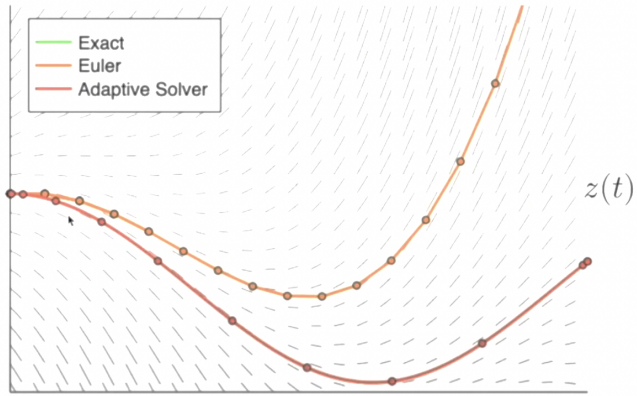
- Vector-valued $\mathbf{z}$ changes in time.
- Time-derivative: $\frac{d\mathbf{z}}{dt} = f(\mathbf{z}(t), t)$.
- Initial-value problem: given $\mathbf{z}(t_0)$, find

$$\mathbf{z}(t_1) = \mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta) dt.$$

- Euler approximates with small steps:

$$\mathbf{z}(t + h) = \mathbf{z}(t) + hf(\mathbf{z}, t).$$





ResNets as Euler Integrators

def $f(z, t, \theta)$:

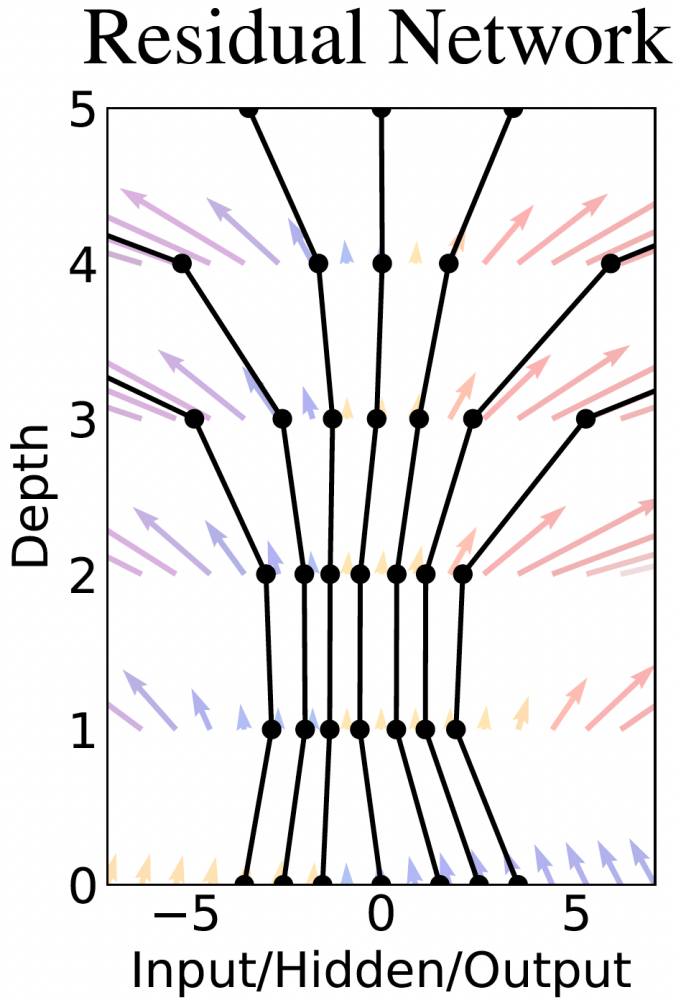
return nnet$([z, t], \theta)$

def resnet(z) :

for t in $[1 : T]$:

$z = z + f(z, t, \theta)$

return z



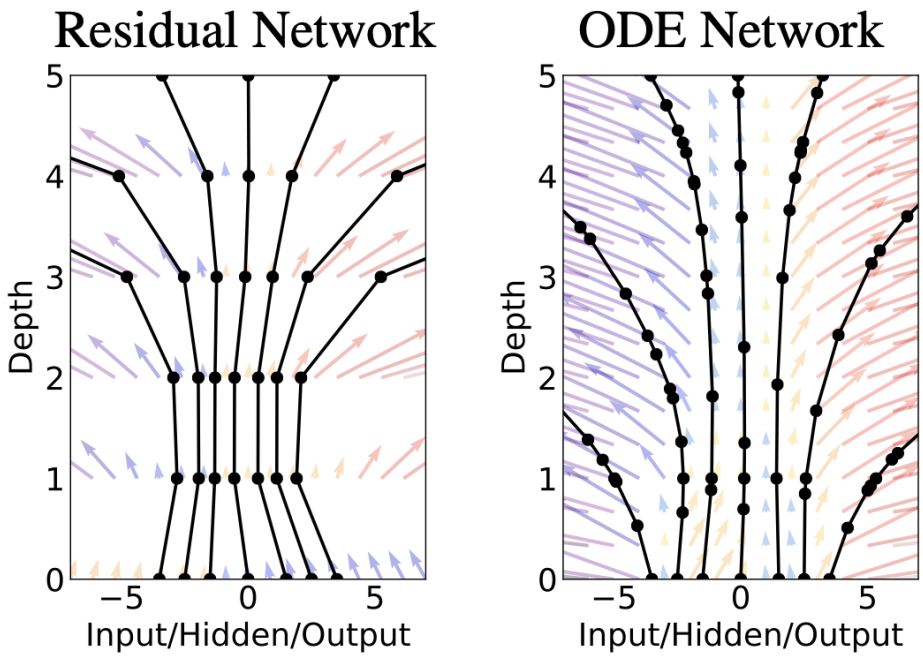
From ResNets to ODENets

def $f(z, t, \theta)$:

return nnet$([z, t], \theta)$

def ODENet(z, θ) :

return ODESolve$(f, z, 0, 1, \theta)$



Outline

Model Optimization

- Consider optimizing a scalar-valued loss function $L(\cdot)$,
- The input is the result of an ODE solver:

$$\begin{aligned} L\left(\mathbf{z}\left(t_1\right)\right) &= L\left(\mathbf{z}\left(t_0\right) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta) dt\right) \\ &= L\left(\text{ODESolve}\left(\mathbf{z}\left(t_0\right), f, t_0, t_1, \theta\right)\right) \end{aligned}$$

How to train an ODENet? $\quad \frac{\partial L}{\partial \theta} = ?$

- Don't backprop through solver: **high memory cost**, **extra numerical error**.

Gradients wrt. θ

Define the adjoint state

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

and we can obtain the **total gradient with respect to the parameters**, by integrating over the full interval.

$$\frac{dL}{d\theta} = \int_{t_N}^{t_0} \mathbf{a}(t) \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \theta} dt.$$

Adjoint Sensitivity Method

- The adjoint state's dynamic is given by ODE:

$$\frac{\mathrm{d}\mathbf{a}(t)}{\mathrm{d}t} = -\mathbf{a}(t)^{\mathsf{T}} \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}}.$$

- We can use pytorch in Python for automatic derivatives and easily get

$$\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}}, \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \theta}, \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial t}.$$

- The proof procedures are detailed in the appendix of slides.

Augmented Adjoint Backward ODE

- Define a vector $\mathbf{b}(t)$ such that $\mathbf{b}(t_1) = \mathbf{0}$ and

$$\dot{\mathbf{b}}(t) = -\mathbf{a}(t)\frac{\partial}{\partial \boldsymbol{\theta}}\mathbf{f}(t, \mathbf{z}(t), \boldsymbol{\theta})$$

  so that $\mathbf{b}(t) = \frac{dL}{d\boldsymbol{\theta}}$.
- To find $\frac{dL}{d\boldsymbol{\theta}}$ and $\frac{dL}{d\mathbf{z}(t_0)}$, we thus need to solve the (augmented) adjoint backward ODE:

$$\frac{d}{dt}\begin{pmatrix} \mathbf{z}(t) \\ \mathbf{a}(t) \\ \mathbf{b}(t) \end{pmatrix} = \begin{pmatrix} \mathbf{f}(t, \mathbf{z}, \boldsymbol{\theta}) \\ -\mathbf{a}(t)\frac{\partial}{\partial \mathbf{z}}\mathbf{f}(t, \mathbf{z}, \boldsymbol{\theta}) \\ -\mathbf{a}(t)\frac{\partial}{\partial \boldsymbol{\theta}}\mathbf{f}(t, \mathbf{z}, \boldsymbol{\theta}) \end{pmatrix} \doteq \mathbf{f}_{aug}\left(t, \begin{pmatrix} \mathbf{z}(t) \\ \mathbf{a}(t) \\ \mathbf{b}(t) \end{pmatrix}, \boldsymbol{\theta}\right)$$

- We can therefore find the sought gradients applying

$$\begin{pmatrix} \cdot \\ \frac{\mathrm{d}L}{\mathrm{d}\mathbf{z}(t_0)} \\ \frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} \end{pmatrix} = \text{ODESolve}\left(\begin{pmatrix} \mathbf{z}(t_1) \\ \frac{\mathrm{d}L}{\mathrm{d}\mathbf{z}(t_1)} \\ \mathbf{0} \end{pmatrix}, \mathbf{f}_{aug}, t_1, t_0, \boldsymbol{\theta}\right).$$

Algorithm

Solve the original ODE and the accumulated gradients backward through time.

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

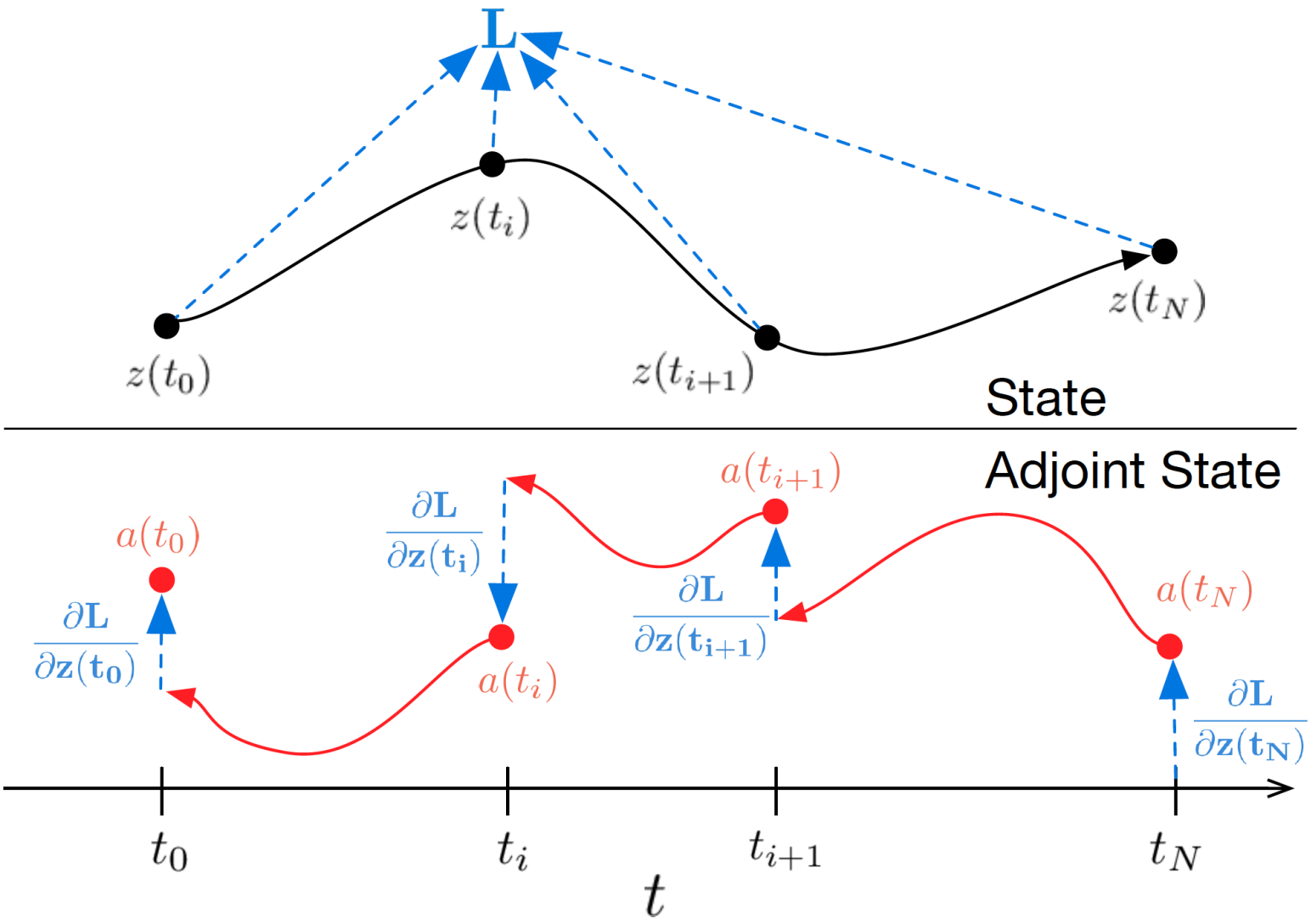
def aug_dynamics($[\mathbf{z}(t), \mathbf{a}(t), \cdot], t, \theta$): ▷ Define dynamics on augmented state

  return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^\mathsf{T} \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^\mathsf{T} \frac{\partial f}{\partial \theta}]$ ▷ 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

Reverse-mode Differentiation of an ODE Solution



Outline

Performance on MNIST

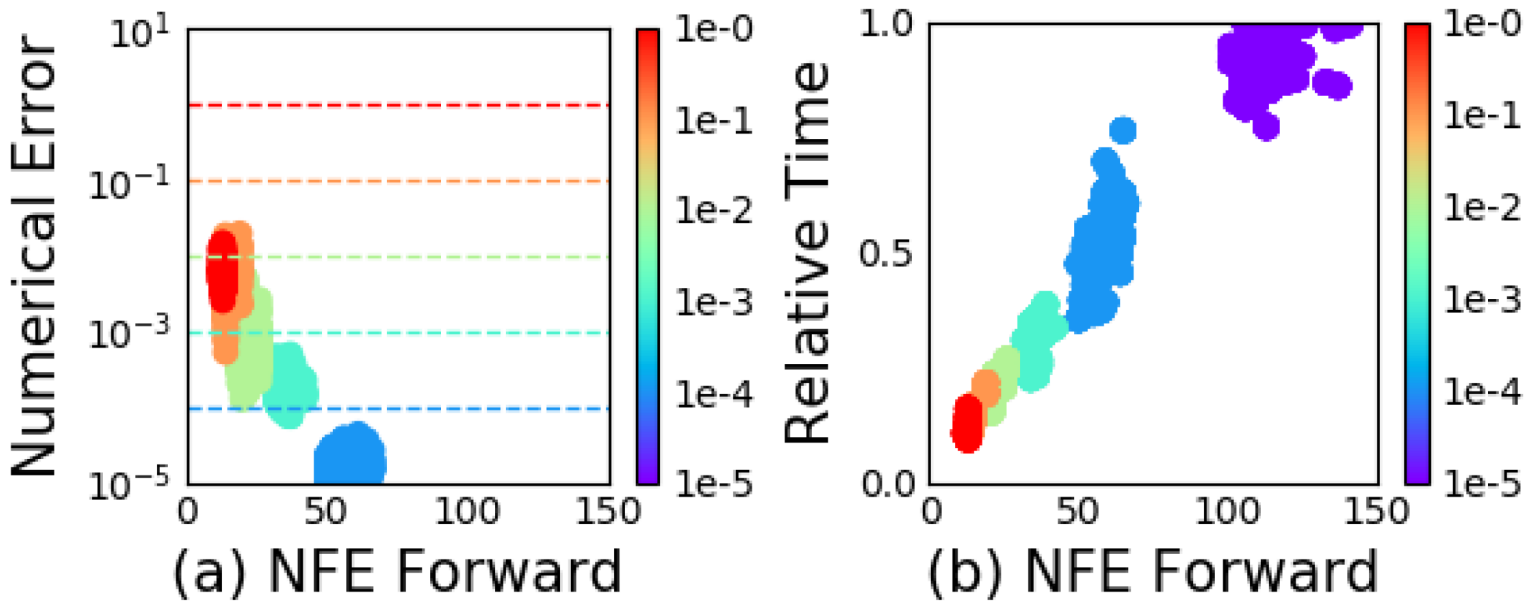
- Don't need to store layer activations for reverse pass.

	Test Error	# Params	Memory	Time
1-Layer MLP†	1.60%	0.24 M	-	-
ResNet	0.41%	0.60 M	$\mathcal{O}(L)$	$\mathcal{O}(L)$
RK-Net	0.47%	0.22 M	$\mathcal{O}(\tilde{L})$	$\mathcal{O}(\tilde{L})$
ODE-Net	0.42%	0.22 M	$\boldsymbol{\mathcal{O}(1)}$	$\mathcal{O}(\tilde{L})$

- L is the number of layers in the ResNet
- $\tilde{L}$ is the number of function evaluations that the ODE solver requests in a single forward pass.
- $\tilde{L}$ can be interpreted as an implicit number of layers.

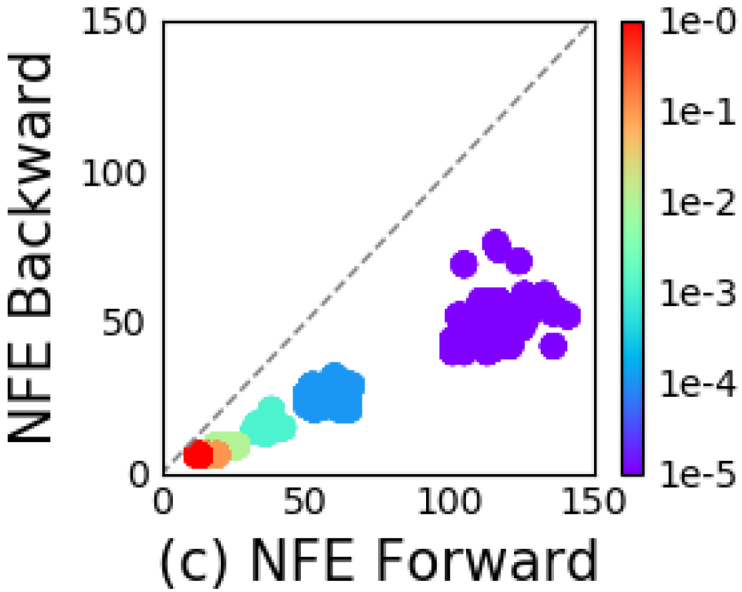
Explicit Error Control & Speed-accuracy Tradeoff

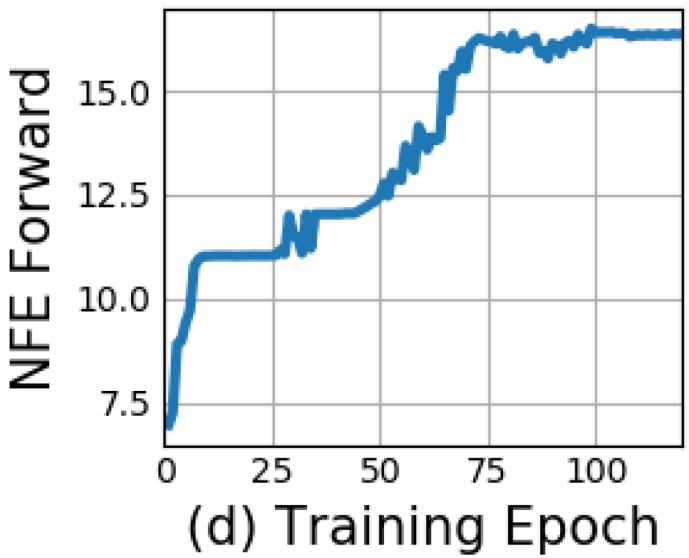
- NFE = number of function evaluations.
- More fine-grained control than lowprecision floats.
- Cost scales with instance difficulty.
- Time cost is dominated by evaluation of dynamics.
- Roughly linear with number of forward evaluations.



Reverse vs. Forward Cost

- Empirically, reverse pass roughly half as expensive as forward pass.
- Num evaluations comparable to number of layers in modern nets.
- Dynamics become more demanding to compute during training.





Example1

- Test a simple linear ODE.
- Dynamics is given with a matrix.

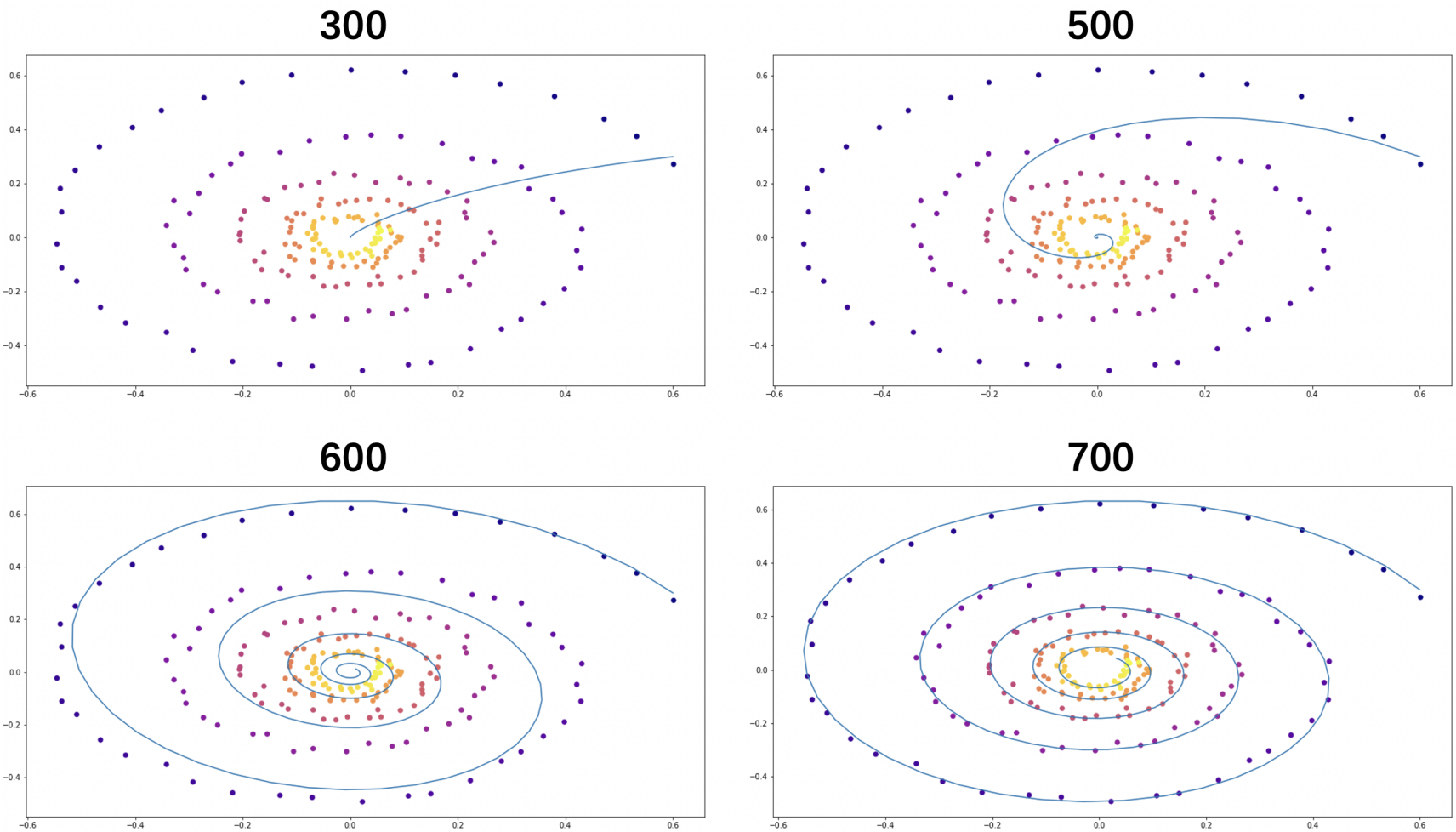
$$\frac{dz}{dt} = \begin{bmatrix} -0.1 & -1.0 \\ 1.0 & -0.1 \end{bmatrix} z.$$

Python code:

```python
class LinearODEF(ODEF):
    def __init__(self, W):
        super(LinearODEF, self).__init__()
        self.lin = nn.Linear(2, 2, bias=False)
        self.lin.weight = nn.Parameter(W)

    def forward(self, x, t):
        return self.lin(x)

class SpiralFunctionExample(LinearODEF):
    def __init__(self):
        super(SpiralFunctionExample, self).__init__(Tensor([[-0.1, -1.], [1., -0.1]]))
```

✓ 0.1s Python

Example1



Example2

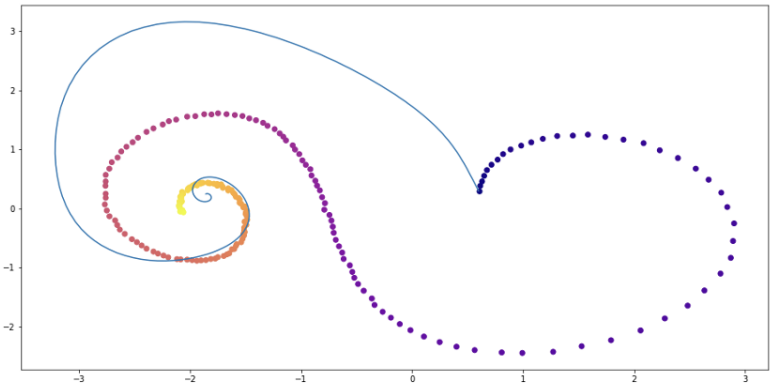
- More sophisticated dynamics for creating trajectories.

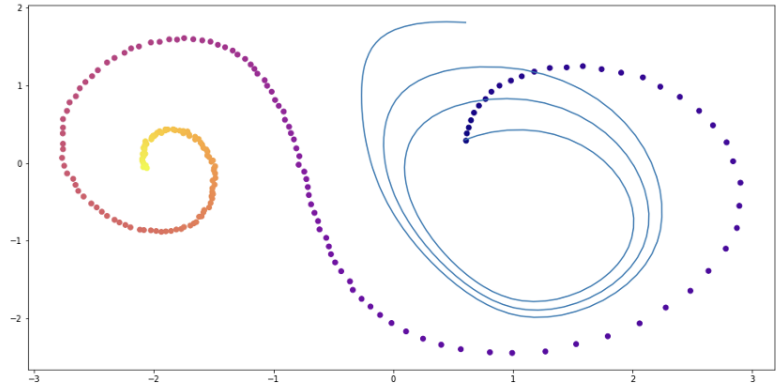
Python code:

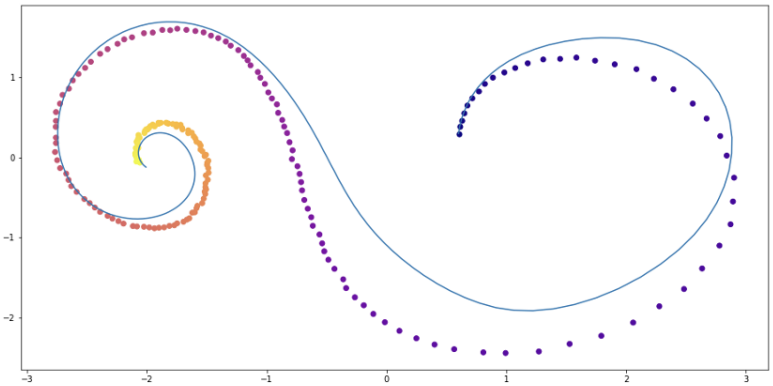
```python
class EX2ODEF(ODEF):
    def __init__(self, A, B, x0):
        super(EX2ODEF, self).__init__()
        self.A = nn.Linear(2, 2, bias=False)
        self.A.weight = nn.Parameter(A)
        self.B = nn.Linear(2, 2, bias=False)
        self.B.weight = nn.Parameter(B)
        self.x0 = nn.Parameter(x0)

    def forward(self, x, t):
        xTx0 = torch.sum(x*self.x0, dim=1)
        dxdt = torch.sigmoid(xTx0) * self.A(x - self.x0) + torch.sigmoid(-xTx0) * self.B(x + self.x0)
        return dxdt

func = EX2ODEF(Tensor([[-0.1, -0.5], [0.5, -0.1]]), Tensor([[0.2, 1.], [-1, 0.2]]), Tensor([[-1., 0.]]))
```

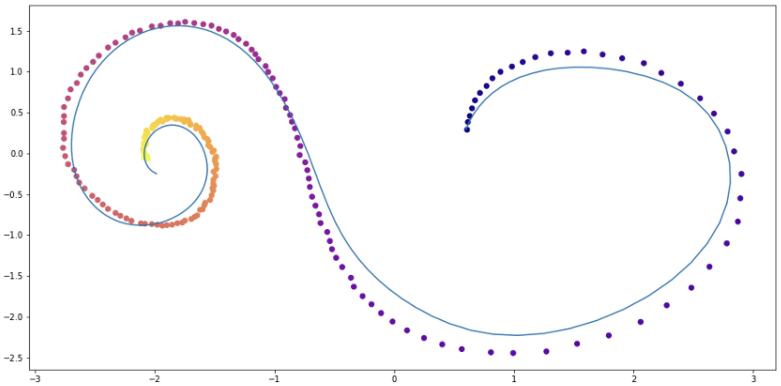
✓ 0.1s Python

Example2









Outline

Continuous-time RNNs

We often want:

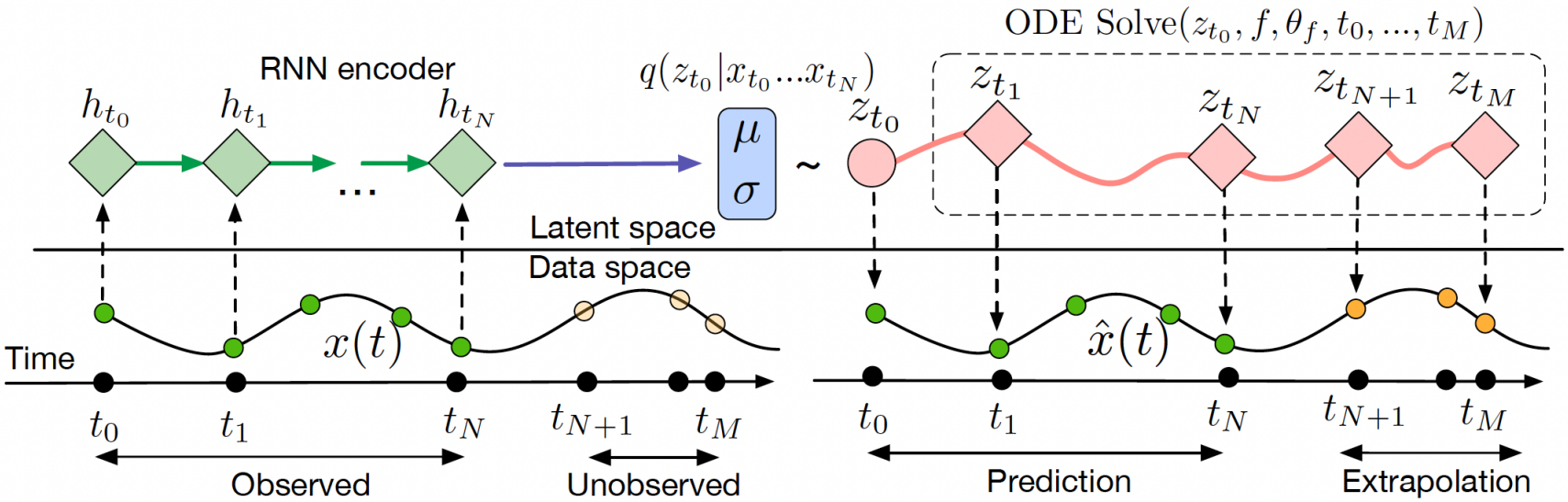
- arbitrary measurement times.
- to decouple dynamics and inference.
- consistently defined state at all times.

A generative model through sampling procedure:

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

Continuous-time RNNs

- VAE-style inference with an RNN encoder.



Variational Autoencoder (VAE)

1. Run the RNN encoder through the time series backwards in time to infer the parameters $\mu_{z_{t_0}}$, $\sigma_{z_{t_0}}$ of variational posterior and sample from it

$$z_{t_0} \sim q\left(z_{t_0} \mid x_{t_0}, ..., x_{t_M}; t_0, ..., t_M; \theta_q\right) = \mathcal{N}\left(z_{t_0} \mid \mu_{z_{t_0}} \sigma_{z_{t_0}}\right).$$

2. Obtain the latent trajectory

$$z_{t_1}, z_{t_2}, ..., z_{t_N} = \text{ODESolve}(z_{t_0}, f, \theta_f, t_0, ..., t_N),$$

$$\text{where } \frac{dz}{dt} = f(z, t; \theta_f).$$

3. Map the latent trajectory onto the data space using another neural network:

$$\hat{x_{t_i}}(z_{t_i}, t_i; \theta_x)$$

4. Maximize **E**vidence **L**ower **BO**und estimate for sampled trajectory

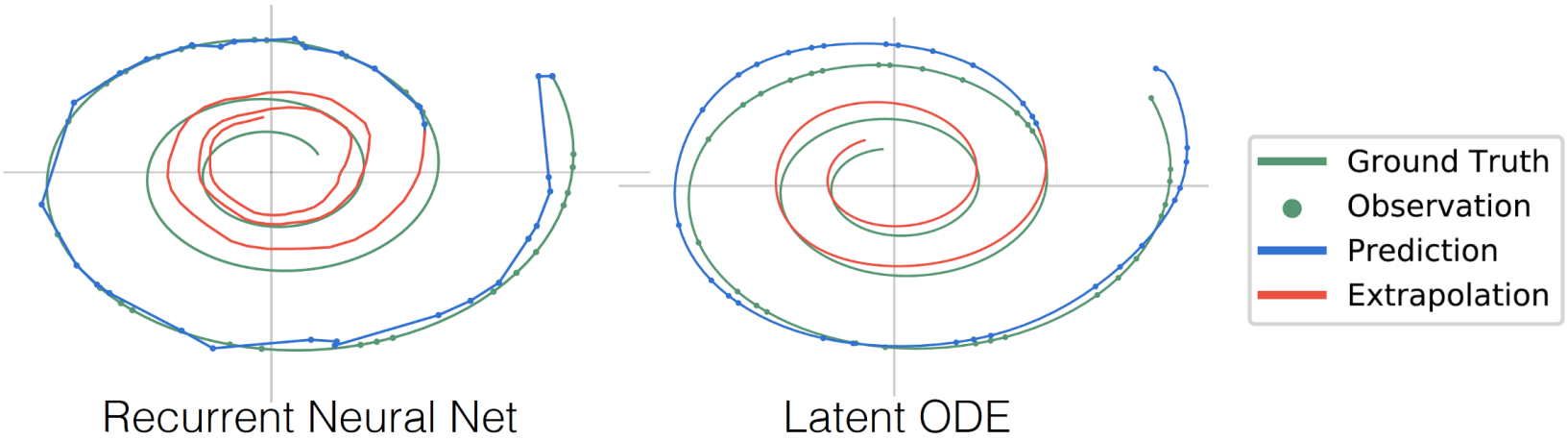
$$\text{ELBO} \approx N\Big(\sum_{i=0}^{M} \log p(x_{t_i} \mid z_{t_i}(z_{t_0}; \theta_f); \theta_x) + KL\left(q(z_{t_0} \mid x_{t_0}, ..., x_{t_M}; t_0, ..., t_M; \theta_q) \parallel \mathcal{N}(0, I)\right) \Big).$$

And in case of Gaussian posterior $p(x \mid z_{t_i}; \theta_x)$ and known noise level σ_x

$$\text{ELBO} \approx -N\Big(\sum_{i=1}^{M} \frac{(x_i - \hat{x}_i)^2}{\sigma_x^2} - \log \sigma_{z_{t_0}}^2 + \mu_{z_{t_0}}^2 + \sigma_{z_{t_0}}^2 \Big) + C$$

Continuous-time RNNs

- ODE VAE combines all noisy observations to reason about underlying trajectory (smoothing).
- Compare reconstructed trajectories using initial moment of time observations of Neural ODE and simple RNN.



Continuous-time RNNs

- Randomly sample points from each trajectory without replacement ($n = \{30, 50, 100\}$)
- Predictive rootmean-squared error (RMSE) on 100 time points extending beyond those that were used for training.

# Observations	30/100	50/100	100/100
RNN	0.3937	0.3202	0.1813
Latent ODE	**0.1642**	**0.1502**	**0.1346**

What's More

- **Generative Model**

Score-based training scales to 1024x1024

Exact density available, but expensive

[Song, Sohl-Dickstein, Kingma, Abhishek, Ermon, Poole. Score-Based Generative Modeling through Stochastic Differential Equations, 2020]

Outline

Proof: Adjoint State

- In standard neural networks, the chain rule is

$$\frac{dL}{d\mathbf{h}_t} = \frac{dL}{d\mathbf{h}_{t+1}} \frac{d\mathbf{h}_{t+1}}{d\mathbf{h}_t}$$

- With a continuous hidden state, we can write the transformation after an ε change in time as

$$\mathbf{z}(t+\varepsilon) = \int_t^{t+\varepsilon} f(\mathbf{z}(t), t, \theta) dt + \mathbf{z}(t),$$

and chain rule can also be applied

$$\frac{dL}{\partial \mathbf{z}(t)} = \frac{dL}{d\mathbf{z}(t+\varepsilon)} \frac{d\mathbf{z}(t+\varepsilon)}{d\mathbf{z}(t)}.$$

Proof: Adjoint State

$$
\begin{aligned}
\frac{d\mathbf{a}(t)}{dt} &= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t)}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon)\frac{\partial}{\partial \mathbf{z}(t)}\mathbf{z}(t+\varepsilon)}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon)\frac{\partial}{\partial \mathbf{z}(t)}\left(\mathbf{z}(t) + \varepsilon f(\mathbf{z}(t), t, \theta) + \mathcal{O}\left(\varepsilon^2\right)\right)}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon)\left(\mathbf{I} + \varepsilon \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + \mathcal{O}\left(\varepsilon^2\right)\right)}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} \frac{-\varepsilon \mathbf{a}(t+\varepsilon)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + \mathcal{O}\left(\varepsilon^2\right)}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} -\mathbf{a}(t+\varepsilon)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + \mathcal{O}(\varepsilon) \\
&= -\mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)}
\end{aligned}
$$

Adjoint Sensitivity Method

- We specify the constraint on the last time point, which is simply the gradient of the loss wrt. the last time point.

$$\underbrace{\mathbf{a}(t_N) = \frac{dL}{d\mathbf{z}(t_N)}}_{\text{initial condition of adjoint diffeq.}} \qquad \underbrace{\mathbf{a}(t_0) = \int_{t_N}^{t_0} \mathbf{a}(t) \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} dt}_{\text{gradient wrt. initial value}}$$

- Loss function L depends only on the last time point t_N.
- We can repeat the adjoint step for each of the intervals $[t_{N-1}, t_N]$, $[t_{N-2}, t_{N-1}]$ in the backward order and sum up the obtained gradients.

Gradients wrt. θ and t

- We view θ and t as states with constant differential equations and write

$$\frac{\partial \theta(t)}{\partial t} = \mathbf{0}, \quad \frac{dt(t)}{dt} = 1$$

- We can then combine these with z to form an augmented state with corresponding differential equation and adjoint state,

$$\frac{d}{dt}\begin{bmatrix} \mathbf{z} \\ \theta \\ t \end{bmatrix}(t) = f_{\text{aug}}\left([\mathbf{z}, \theta, t]\right) := \begin{bmatrix} f([\mathbf{z}, \theta, t]) \\ \mathbf{0} \\ 1 \end{bmatrix},$$

$$\mathbf{a}_{\text{aug}} := \begin{bmatrix} \mathbf{a} \\ \mathbf{a}_\theta \\ \mathbf{a}_t \end{bmatrix}, \mathbf{a}_\theta(t) := \frac{dL}{d\theta(t)}, \mathbf{a}_t(t) := \frac{dL}{dt(t)}$$

Gradients wrt. θ and t

- Note this formulates the augmented ODE as an autonomous (time-invariant) ODE, but the derivations in the previous section still hold as this is a special case of a time-variant ODE. The Jacobian of f has the form

$$\frac{\partial f_{aug}}{\partial [\mathbf{z}, \theta, t]} = \begin{bmatrix} \frac{\partial f}{\partial \mathbf{z}} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

where each $\mathbf{0}$ is a matrix of zeros with the appropriate dimensions. We can obtain

$$\begin{aligned} \frac{d\mathbf{a}_{aug}(t)}{dt} &= -\begin{bmatrix} \mathbf{a}(t) & \mathbf{a}_\theta(t) & \mathbf{a}_t(t) \end{bmatrix} \frac{\partial f_{\text{aug}}}{\partial [\mathbf{z}, \theta, t]}(t) \\ &= -\begin{bmatrix} \mathbf{a}\frac{\partial f}{\partial \mathbf{z}} & \mathbf{a}\frac{\partial f}{\partial \theta} & \mathbf{a}\frac{\partial f}{\partial t} \end{bmatrix}(t) \end{aligned}$$

Gradients wrt. θ and t

- We obtain the total gradient with respect to the parameters, by integrating over the full interval.

$$\frac{dL}{d\theta} = \int_{t_N}^{t_0} \mathbf{a}(t) \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \theta} dt$$

- Note the negative sign cancels out since we integrate backwards from t_N to t_0.
- We also get gradients with respect to t_0 and t_N:

$$\frac{dL}{dt_N} = -\mathbf{a}\left(t_N\right) \frac{\partial f\left(\mathbf{z}\left(t_N\right), t_N, \theta\right)}{\partial t_N},$$

$$\frac{dL}{dt_0} = \int_{t_N}^{t_0} \mathbf{a}(t) \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial t} dt.$$