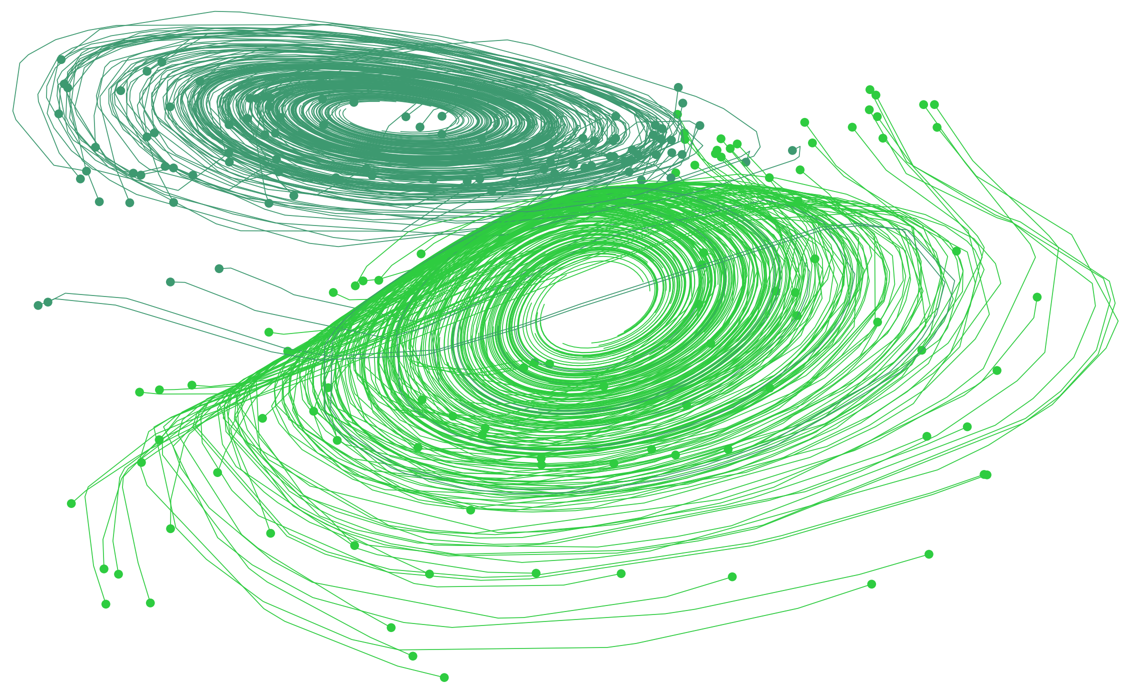
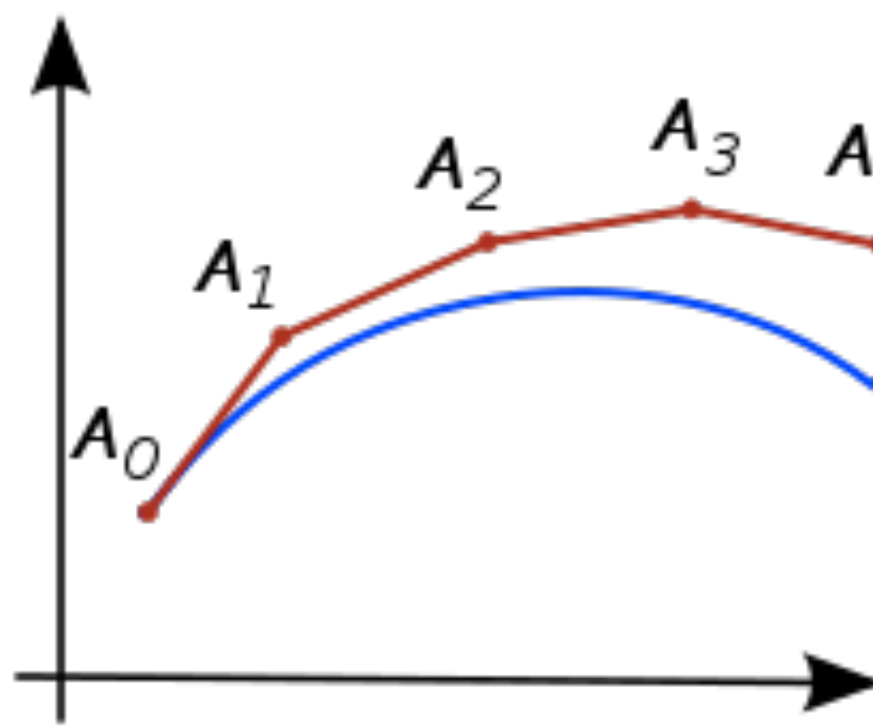
Replacing Neural Networks with Black-Box ODE Solvers



Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, David Duvenaud
University of Toronto, Vector Institute

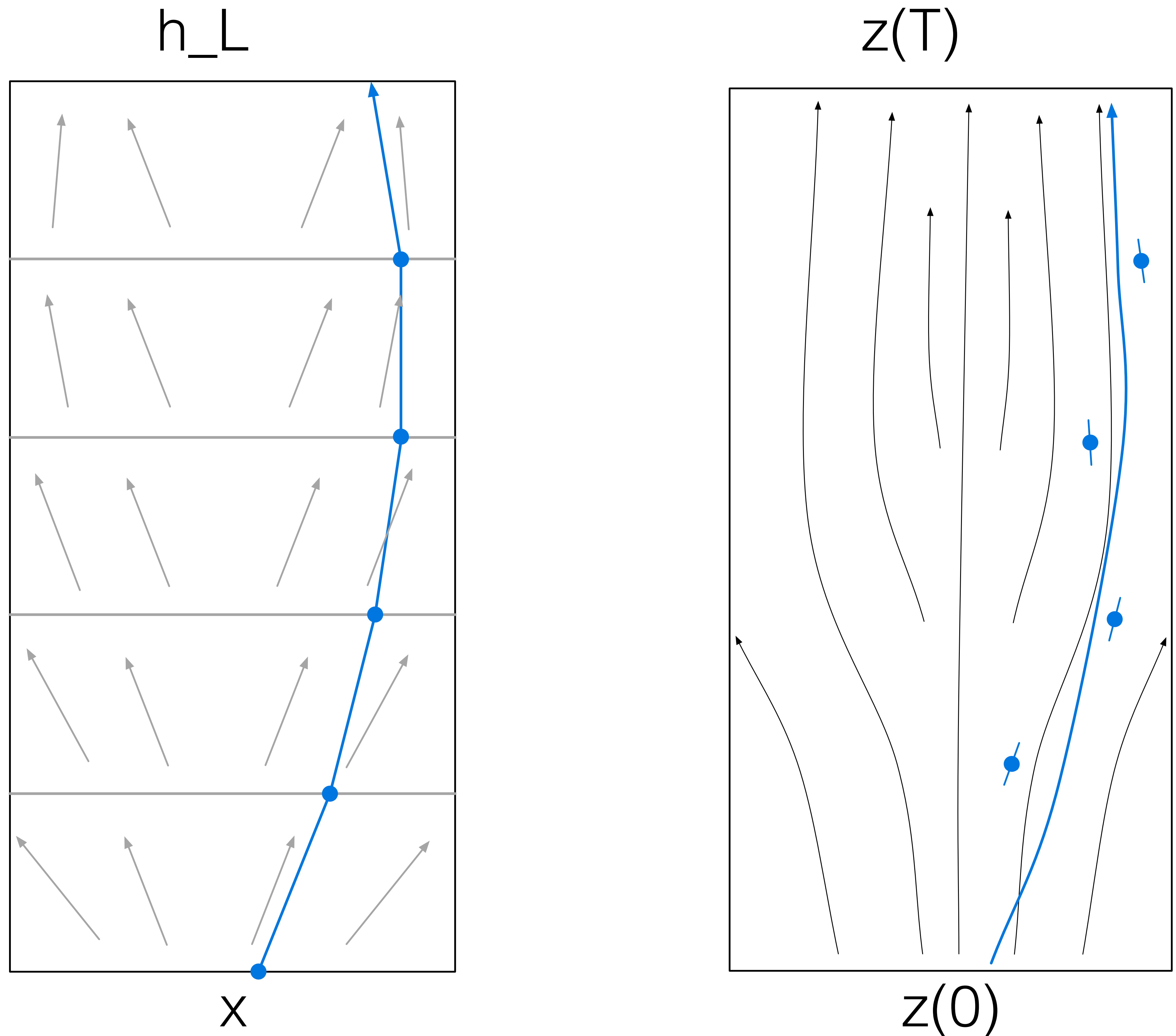
Resnets are Euler integrators

- Middle layers look like: $\mathbf{h}_{t+1} = \mathbf{h}_t + f(\mathbf{h}_t, \theta_t)$



- Limit of smaller steps: $\frac{d\mathbf{z}(t)}{dt} = f(\mathbf{z}(t), \theta(t))$

From Resnets to ODEnets



Why not an ODE solver?

- Parameterize $$\frac{d\mathbf{z}(t)}{dt} = f(\mathbf{z}(t), \theta(t))$$
- Define z(T) to be top layer of residual network, or recurrent neural network, or normalizing flow…
  - RNNs: No need to discretize time
  - Fewer parameters: Neighboring layers automatically similar
  - Density models: Efficiently invertible. Math is nicer.
  - O(1) memory cost, due to reversibility
  - Adaptive, explicit tradeoff between speed and accuracy. No wasted layers?

Backprop through an ODE solver is wasteful

- Ultimately want to optimize some loss

$$L(\mathbf{z}(t_1)) = L\left(\int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)dt\right)$$
$$= L\left(\mathrm{ODESolve}(\mathbf{z}(t_0), f, t_0, t_1, \theta)\right)$$

- How to compute gradients of ODESolve?

- Backprop through operations of solver is slow, has bad numerical properties, and high memory cost

Reverse-time autodiff

- Define adjoint: $a(t) = -\partial L / \partial \mathbf{z}(t)$
- Which has dynamics: $\dfrac{da(t)}{dt} = -a(t)^T \dfrac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}}$
- Start adjoint with $\partial L / \partial \mathbf{z}(t_1)$
- And solve a combined ODE backwards in time:

$$\frac{dL}{d\theta} = \int_{t_1}^{t_0} a(t)^T \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \theta} dt$$

[Scalable Inference of Ordinary Differential Equation Models of Biochemical Processes", Froehlich, Loos, Hasenauer, 2017]

Reverse-time autodiff

- In english: Solve the original ODE and the accumulated gradients backwards 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 $\partial L / \partial \mathbf{z}(t_1)$

$\frac{\partial L}{\partial t_1} = \frac{\partial L}{\partial \mathbf{z}(t_N)}^T f(\mathbf{z}(t_1), t_1, \theta)$ ▷ Compute gradient w.r.t. t_1

$s = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, -\frac{\partial L}{\partial t_1}, \mathbf{0}]$ ▷ Define initial augmented state

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

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

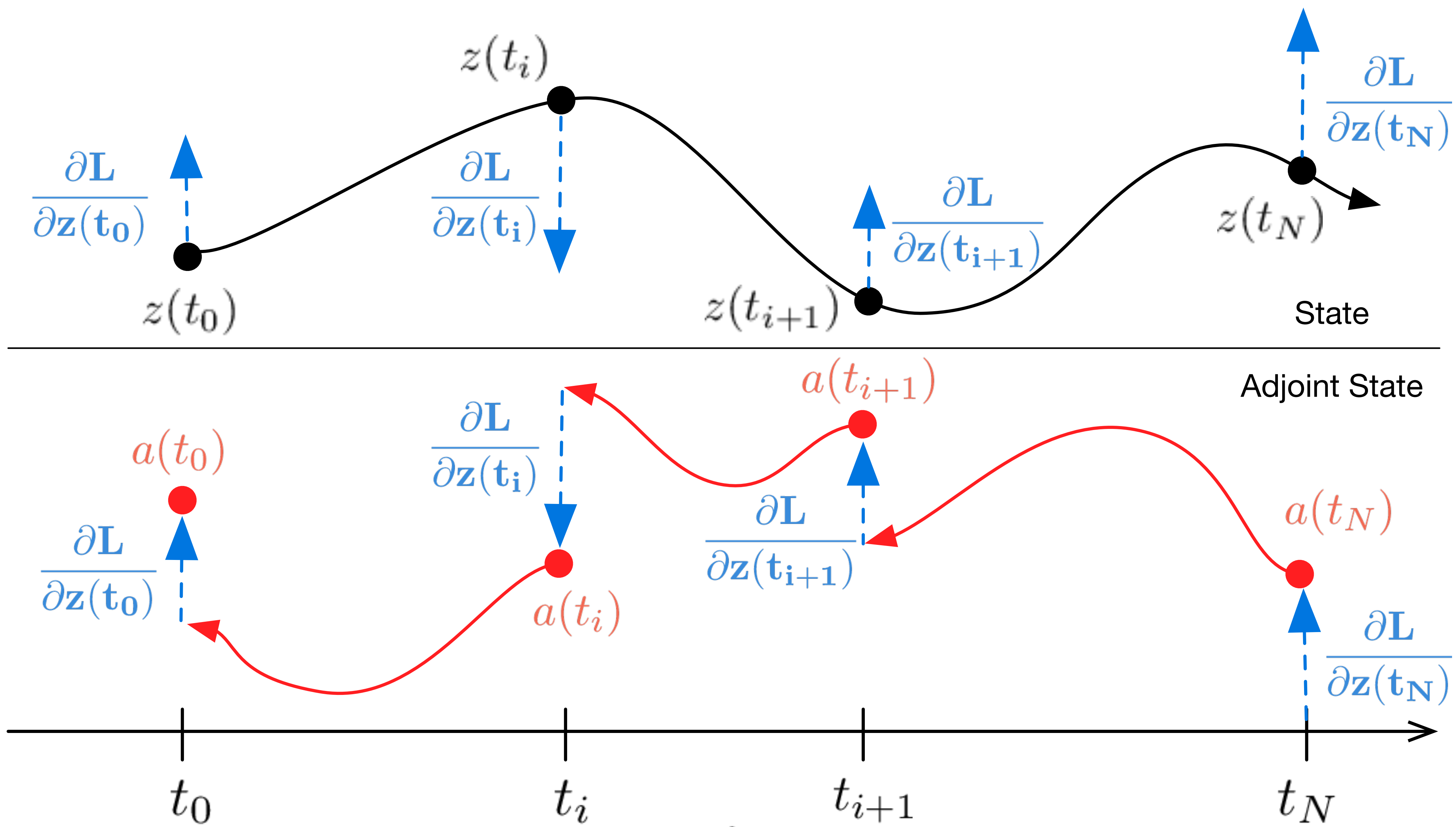
$[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial t_0}] = \text{ODESolve}(s, \text{Dynamics}, t_1, t_0, \theta)$ ▷ Solve reverse-time ODE

return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial t_0}, \frac{\partial L}{\partial t_1}$ ▷ Return all gradients

---

Can ask for multiple measurement times

Reverse pass breaks solution into N-1 chunks



```python
def grad_odeint(yt, func, y0, t, func_args, **kwargs):
    # Extended from "Scalable Inference of Ordinary Differential
    # Equation Models of Biochemical Processes", Sec. 2.4.2
    # Fabian Froehlich, Carolin Loos, Jan Hasenauer, 2017
    # https://arxiv.org/abs/1711.08079

    T, D = np.shape(yt)
    flat_args, unflatten = flatten(func_args)

    def flat_func(y, t, flat_args):
        return func(y, t, *unflatten(flat_args))

    def unpack(x):
        #      y,      vjp_y,       vjp_t,    vjp_args
        return x[0:D], x[D:2 * D], x[2 * D], x[2 * D + 1:]

    def augmented_dynamics(augmented_state, t, flat_args):
        # Orginal system augmented with vjp_y, vjp_t and vjp_args.
        y, vjp_y, _, _ = unpack(augmented_state)
        vjp_all, dy_dt = make_vjp(flat_func, argnum=(0, 1, 2))(y, t, flat_args)
        vjp_y, vjp_t, vjp_args = vjp_all(-vjp_y)
        return np.hstack((dy_dt, vjp_y, vjp_t, vjp_args))

    def vjp_all(g):

        vjp_y = g[-1, :]
        vjp_t0 = 0
        time_vjp_list = []
        vjp_args = np.zeros(np.size(flat_args))

        for i in range(T - 1, 0, -1):

            # Compute effect of moving measurement time.
            vjp_cur_t = np.dot(func(yt[i, :], t[i], *func_args), g[i, :])
            time_vjp_list.append(vjp_cur_t)
            vjp_t0 = vjp_t0 - vjp_cur_t

            # Run augmented system backwards to the previous observation.
            aug_y0 = np.hstack((yt[i, :], vjp_y, vjp_t0, vjp_args))
            aug_ans = odeint(augmented_dynamics, aug_y0,
                             np.array([t[i], t[i - 1]]), tuple((flat_args,)), **kwargs)
            _, vjp_y, vjp_t0, vjp_args = unpack(aug_ans[1])

            # Add gradient from current output.
            vjp_y = vjp_y + g[i - 1, :]

        time_vjp_list.append(vjp_t0)
        vjp_times = np.hstack(time_vjp_list)[::-1]

        return None, vjp_y, vjp_times, unflatten(vjp_args)
    return vjp_all
```

- First implementation of reverse-mode autodiff through black-box ODE solvers
- Solves a system of size 2D + K + 1
- Stan has forward-mode implementation, which solves a system of size D^2 + KD
- Tensorflow has Runge-Kutta 4,5 implemented, but naive autodiff
- Julia has limited support
- We have PyTorch impl

O(1) Memory Cost

- Don't need to store layer activations for reverse pass - just follow dynamics in reverse!

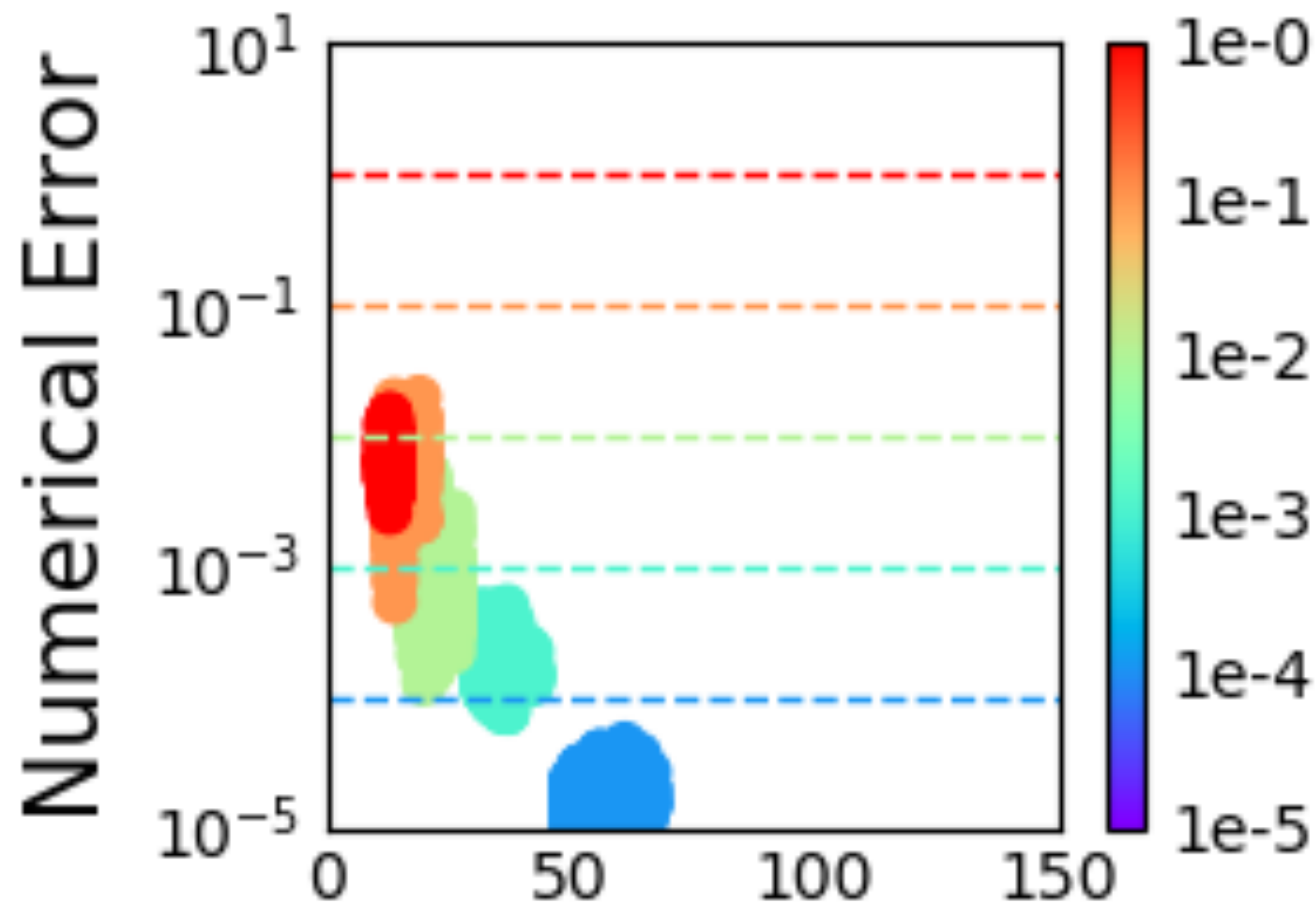
Table 1: Performance on MNIST. †From [23].

	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	$\mathcal{O}(1)$	$\mathcal{O}(\tilde{L})$

- Reversible resnets [Gomez, Ren, Urtasun,Grosse, 2018] also have this property, but require partitioning dimensions

Explicit Error Control

- More fine-grained control than low-precision floats
- Cost scales with instance difficulty

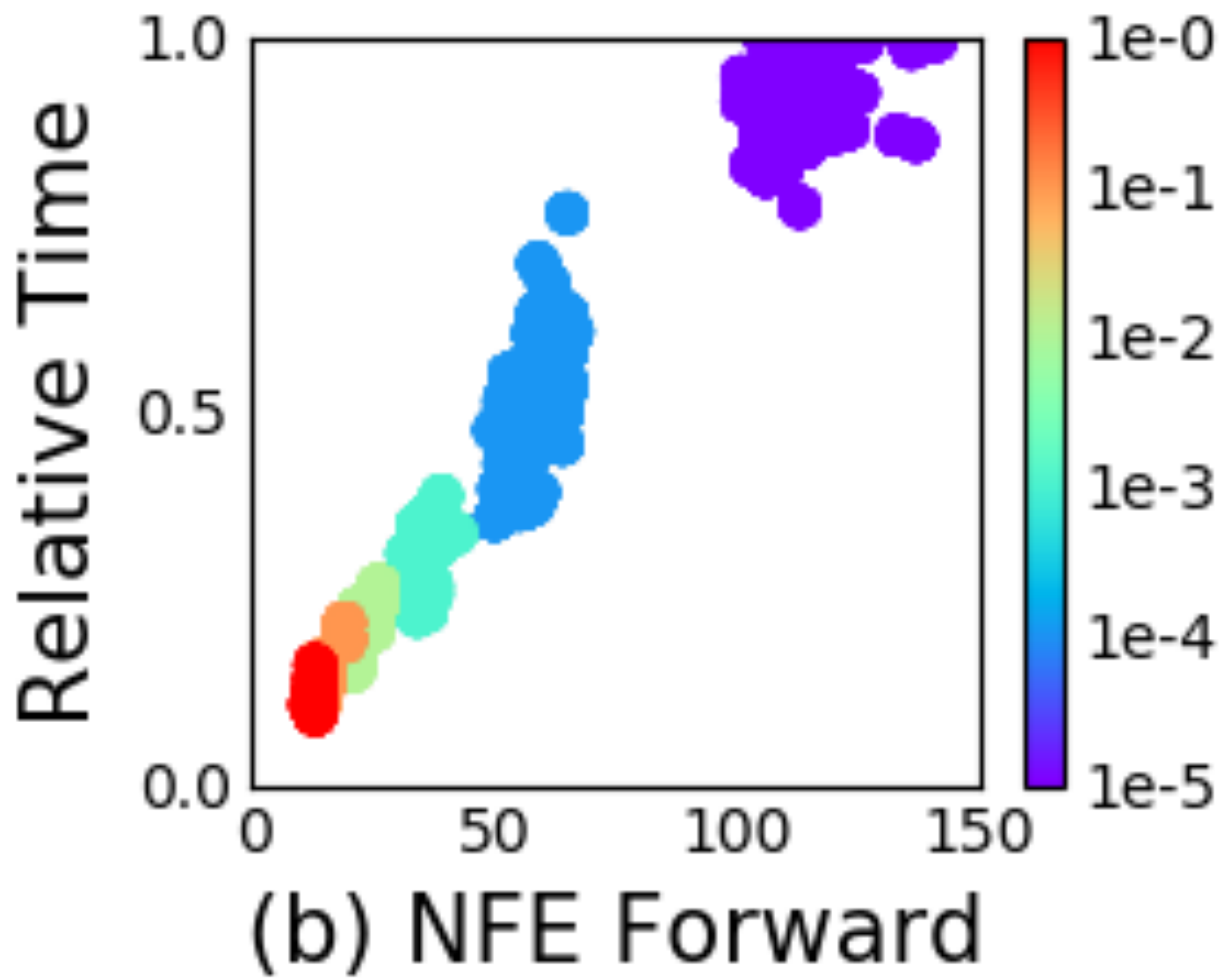


(a) NFE Forward

Speed-Accuracy Tradeoff

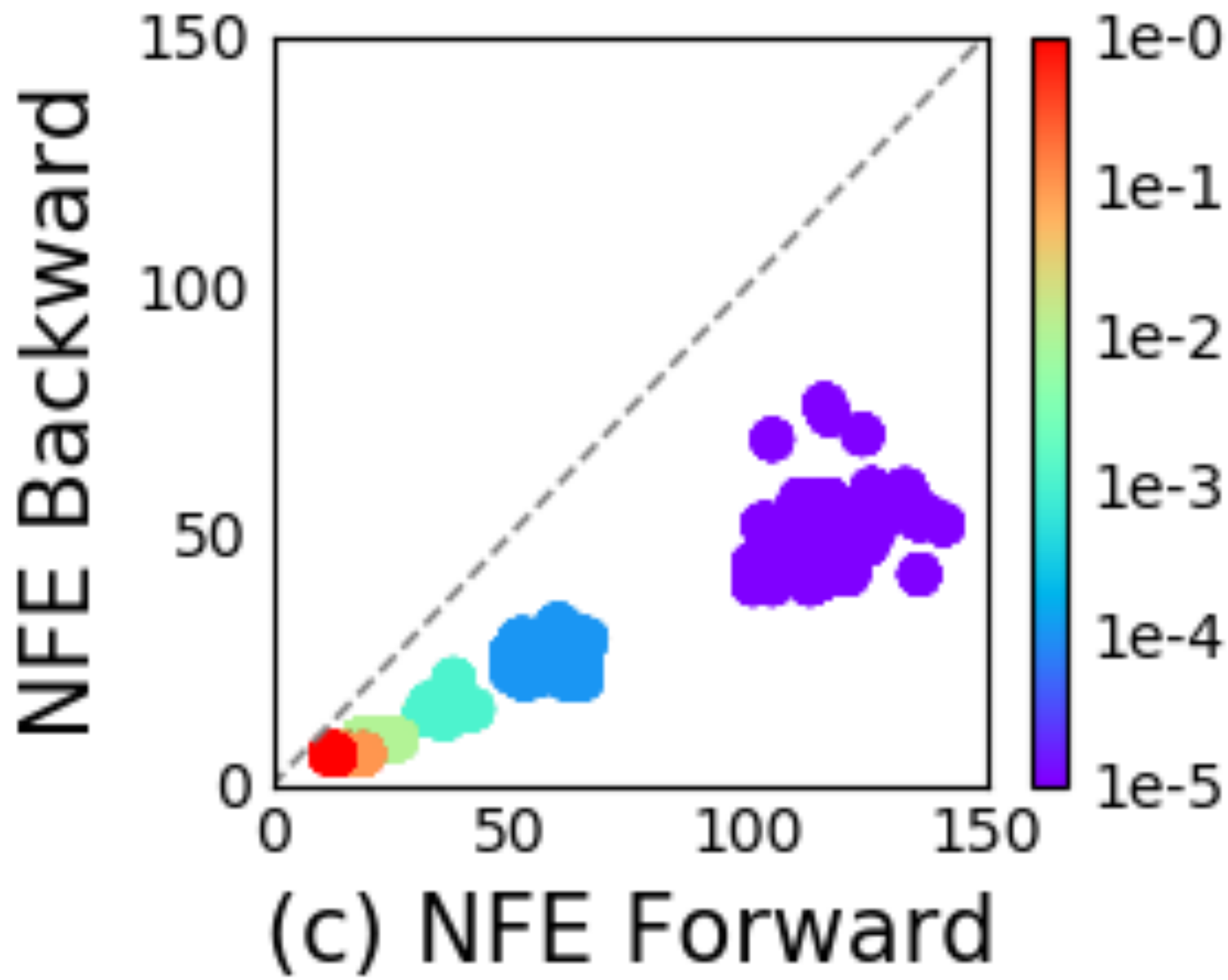
- Time cost is dominated by evaluation of dynamics

- Roughly linear with number of forward evaluations



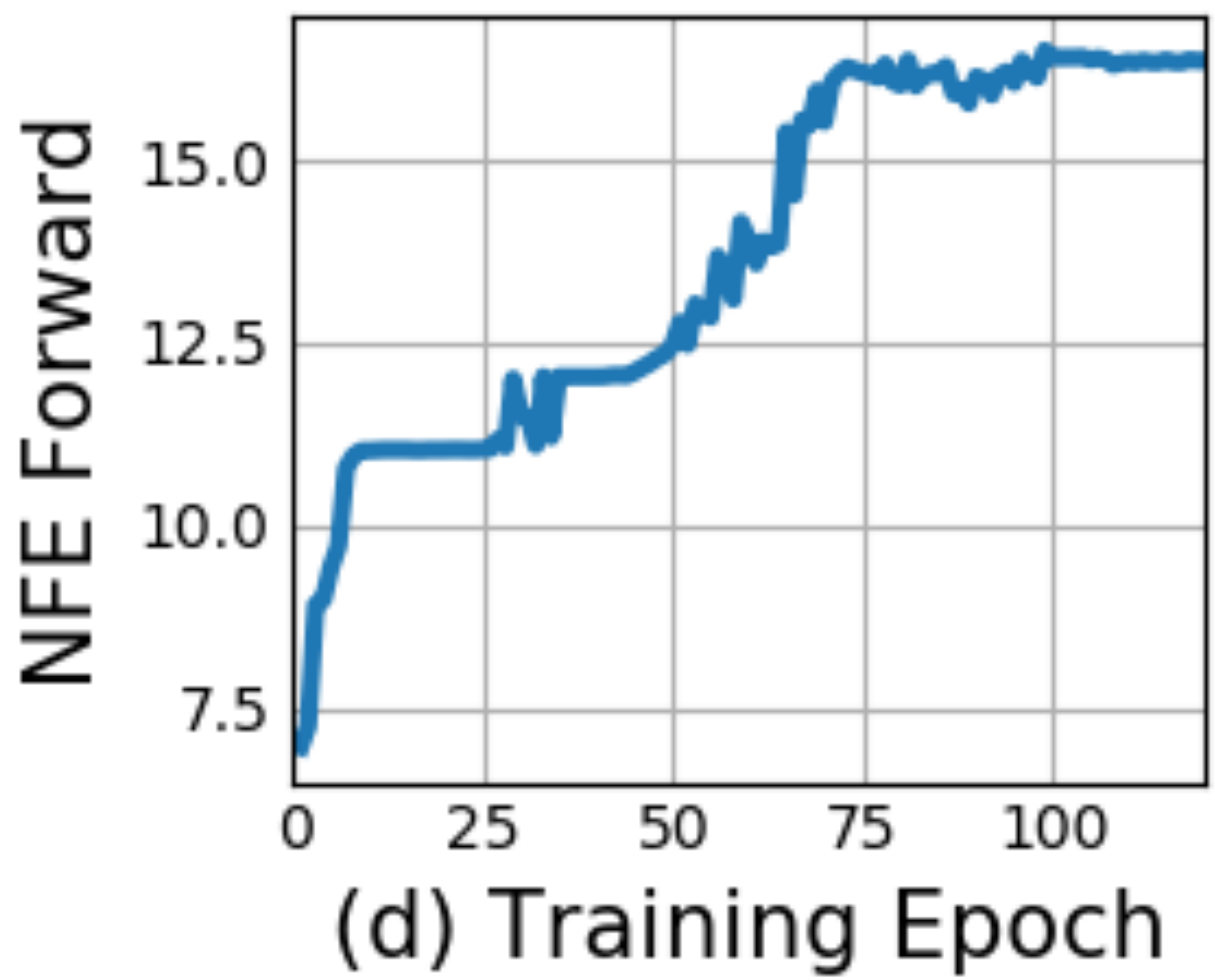
Reverse vs Forward Cost

- Empirically, reverse pass roughly half as expensive as forward pass
- Again, adapts to instance difficulty
- Num evaluations comparable to number of layers in modern nets



How complex are the dynamics?

- Dynamics become more demanding to compute during training



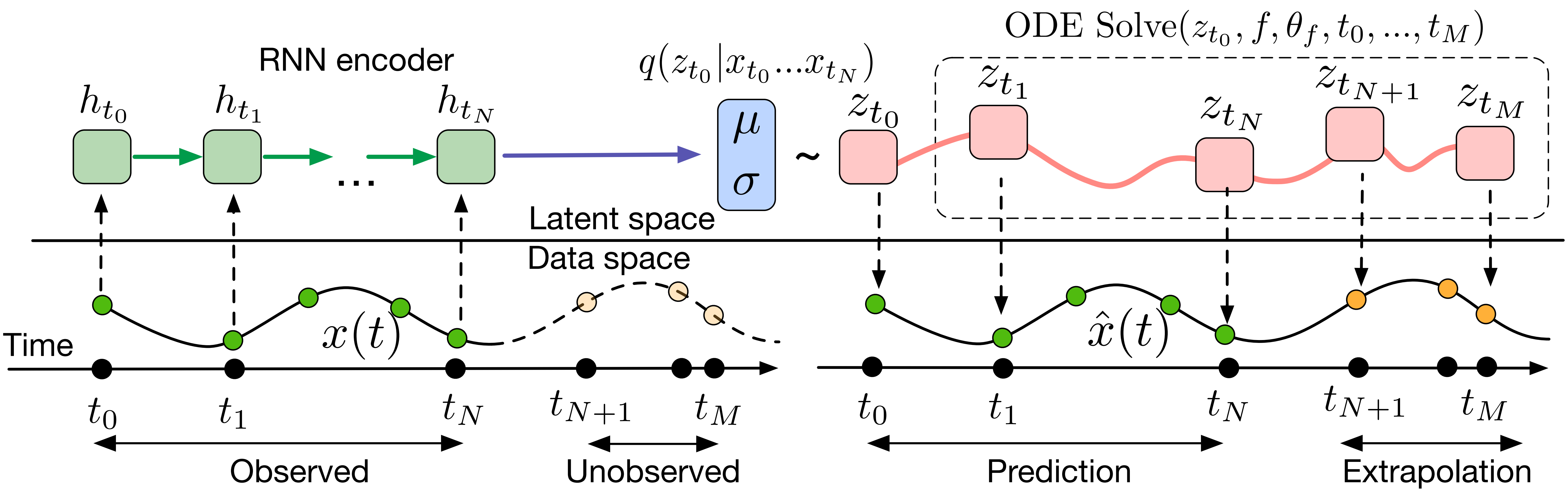
Continuous-time RNNs

- We often want:
  - arbitrary measurement times
  - to decouple dynamics and inference
  - consistently defined state at all times

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

Continuous-time RNNs

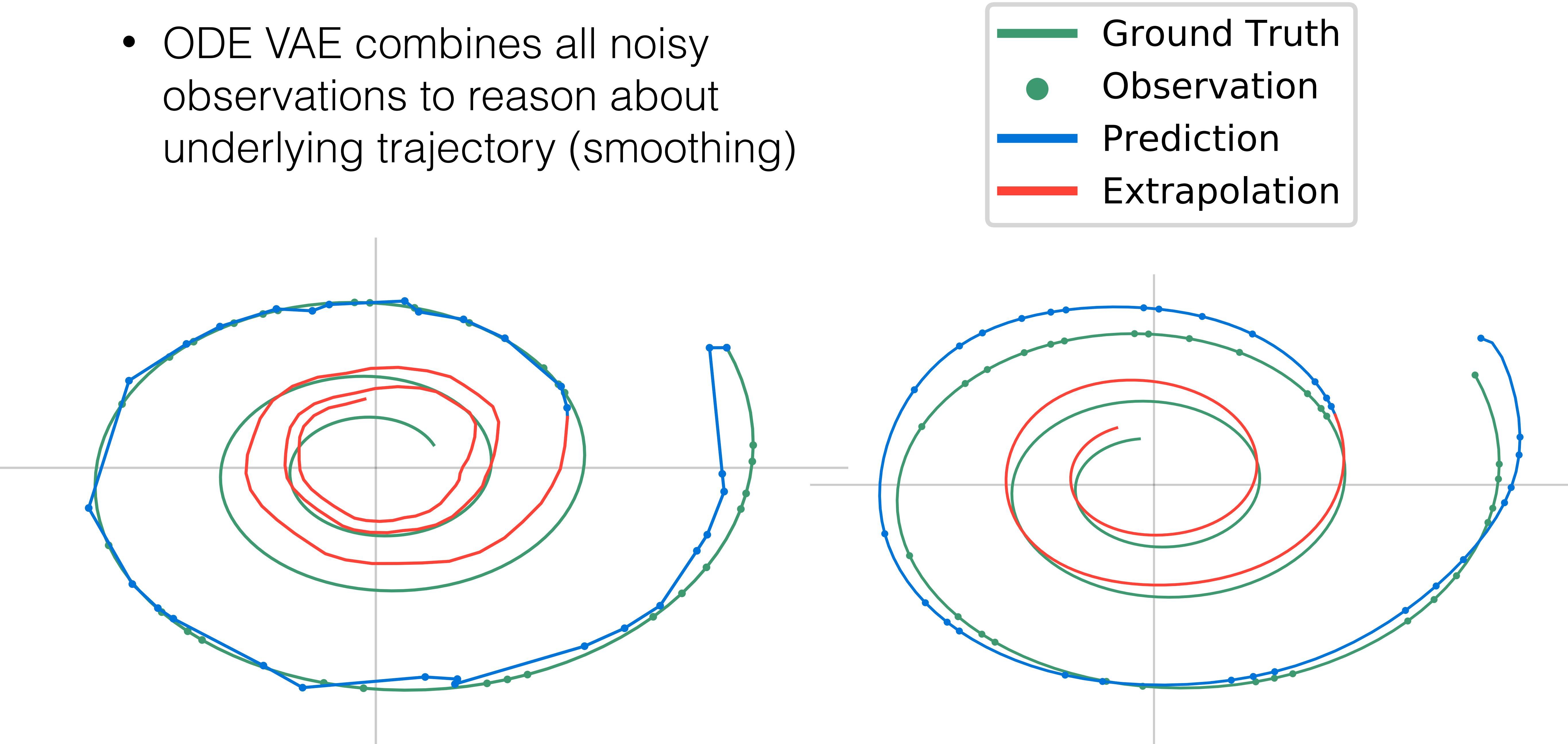
- Can do VAE-style inference with an RNN encoder
- Actually, more like a Deep Kalman Filter



- TODO: move to stochastic differential equations

RNNs vs Latent ODE

- ODE VAE combines all noisy observations to reason about underlying trajectory (smoothing)



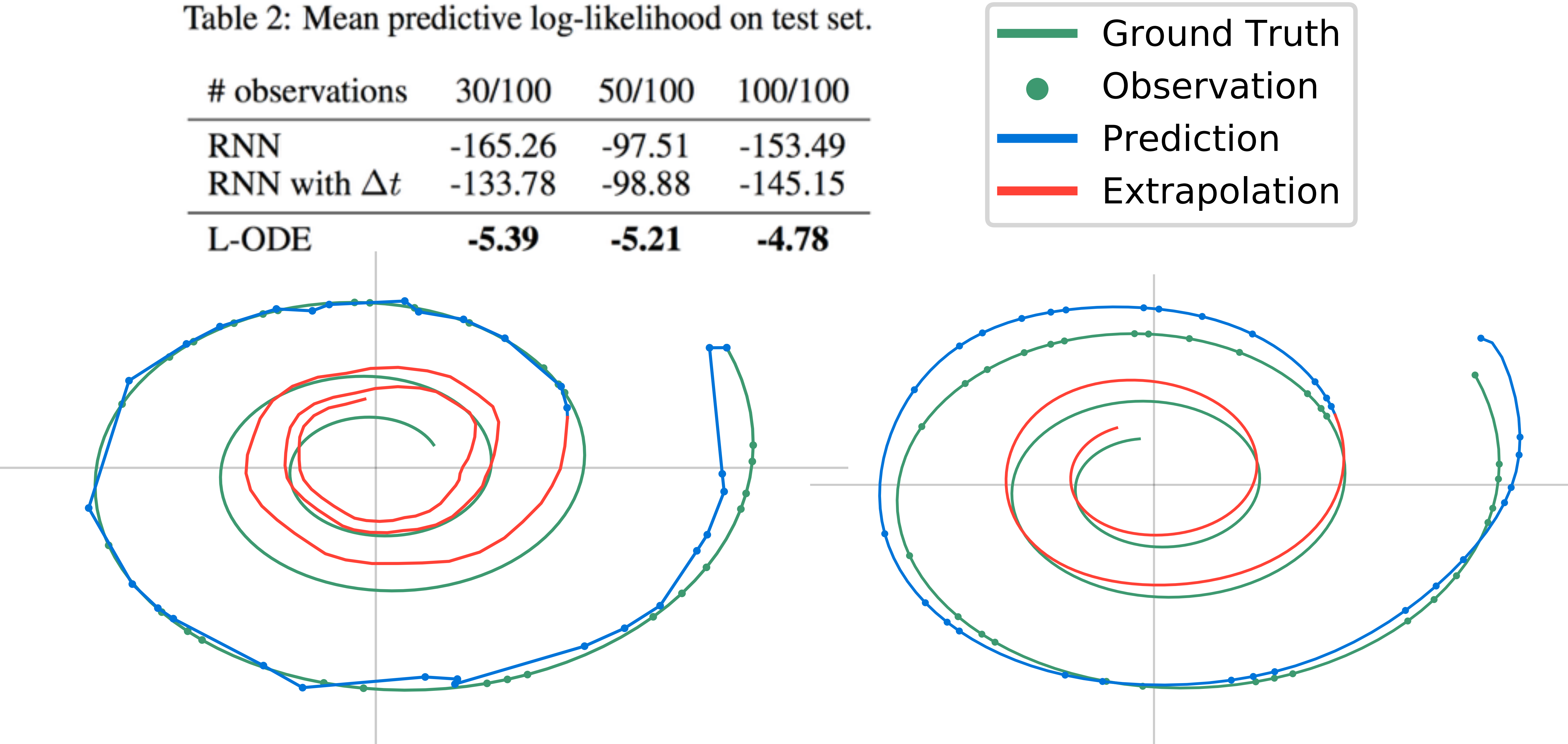
Recurrent Neural Net

Latent ODE

RNNs vs Latent ODE

Table 2: Mean predictive log-likelihood on test set.

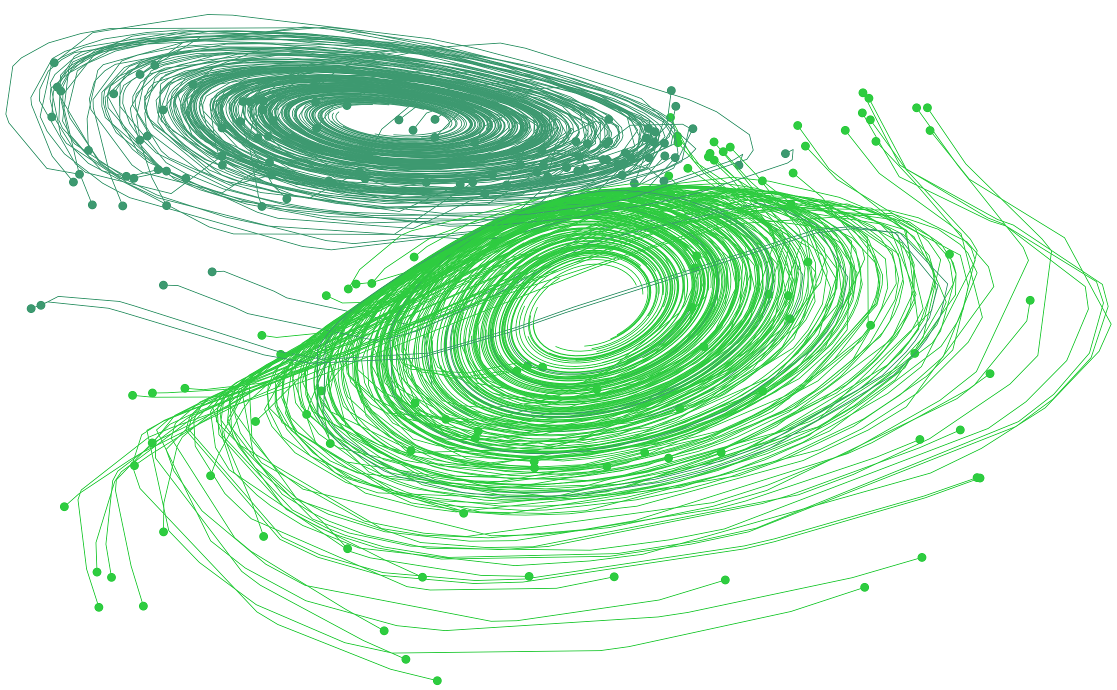
# observations	30/100	50/100	100/100
RNN	-165.26	-97.51	-153.49
RNN with Δt	-133.78	-98.88	-145.15
L-ODE	**-5.39**	**-5.21**	**-4.78**



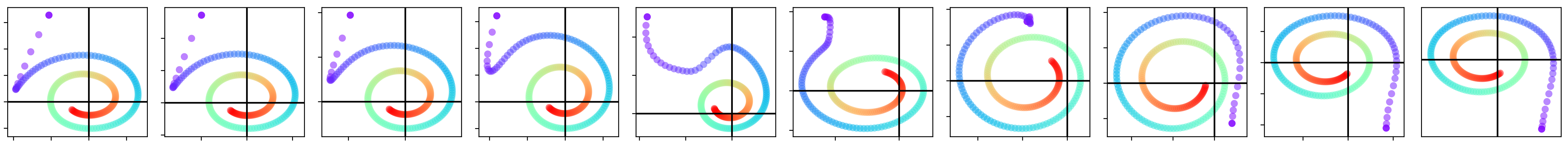
Recurrent Neural Net

Latent ODE

Latent space exploration



Each 3D latent point corresponds to a trajectory

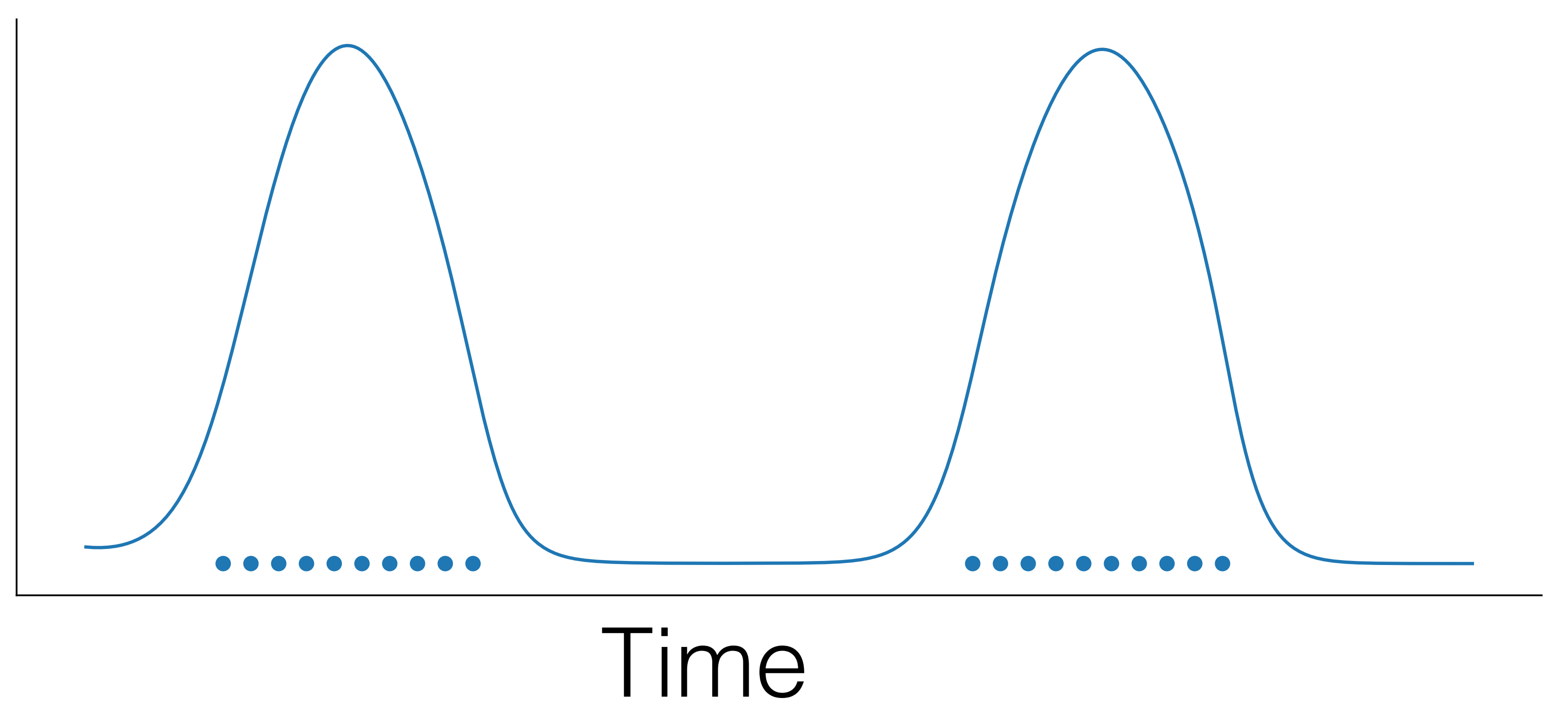


Poisson Process Likelihoods

- Can condition on observation times
- Define rate function as a function of latent state
- Poisson likelihood is just another integral, can be solved along with latent state

$$\log p(t_1, \ldots, t_N | t_{\text{start}}, t_{\text{end}})$$

$$= \sum_{i=1}^{N} \log \lambda(\mathbf{z}(t_i)) - \int_{t_{\text{start}}}^{t_{\text{end}}} \lambda(\mathbf{z}(t)) dt$$



Normalizing Flows

$$x_1 = f(x_0) \implies p(x_1) = p(x_0) \left| \det \frac{\partial f}{\partial x_0} \right|^{-1}$$

- Determinant of Jacobian has cost O(D^3).
- Matrix determinant lemma gives O(DH^3) cost.
- Normalizing flows use 1 hidden unit. Deep & skinny

$$x(t+1) = x(t) + uh(w^T x(t) + b)$$

$$\log p(x(t+1)) = \log p(x(t)) - \log \left| 1 + u^T \frac{\partial h}{\partial x} \right|$$

Continuous Normalizing Flows

- What if we move to continuous transformations?

$$\frac{\partial \log p(x(t))}{\partial t} = -\mathrm{tr}\left(\frac{df}{dx}(t)\right)$$

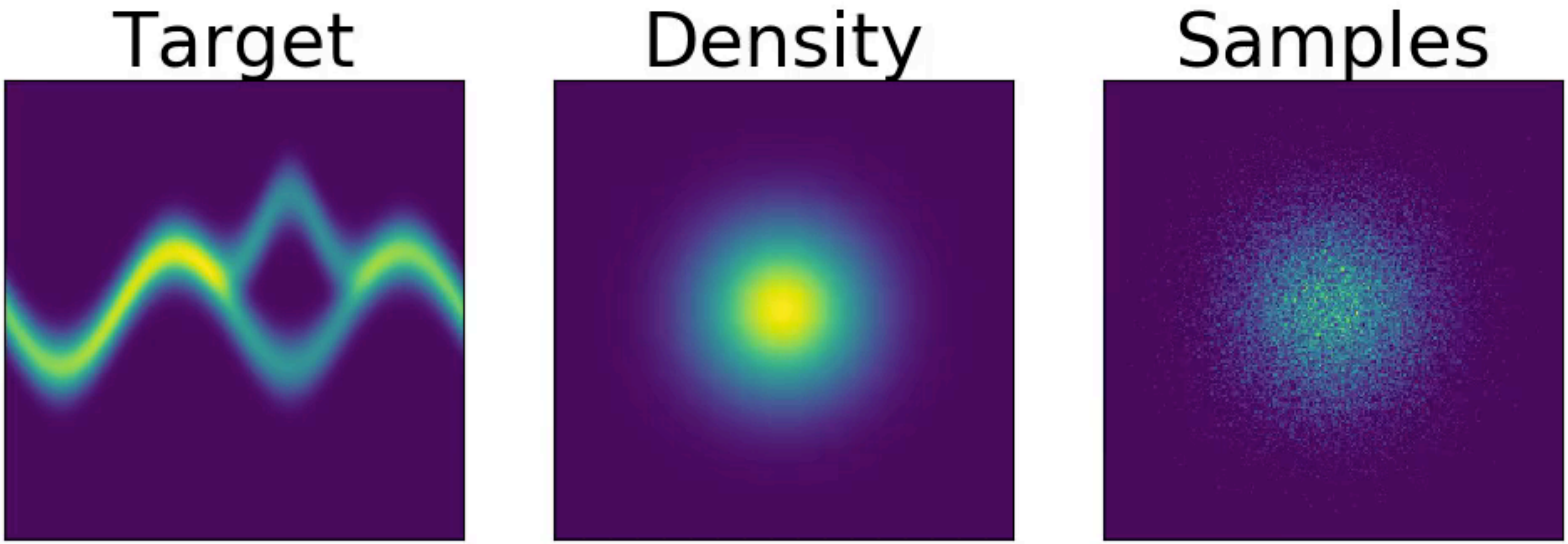
- Time-derivative only depends on trace of Jacobian

$$\frac{dx}{dt} = uh(w^T x + b), \quad \frac{\partial \log p(x)}{\partial t} = -u^T \frac{\partial h}{\partial x}$$

- Trace of sum is sum of traces - O(HD) cost!

$$\frac{dx}{dt} = \sum_n f_n(x), \quad \frac{d \log p(x(t))}{dt} = \sum_n \mathrm{tr}\left(\frac{\partial f}{\partial x}\right)$$

Continuous Normalizing Flows



All videos at https://goo.gl/cqHBzF

Trading Depth for Width

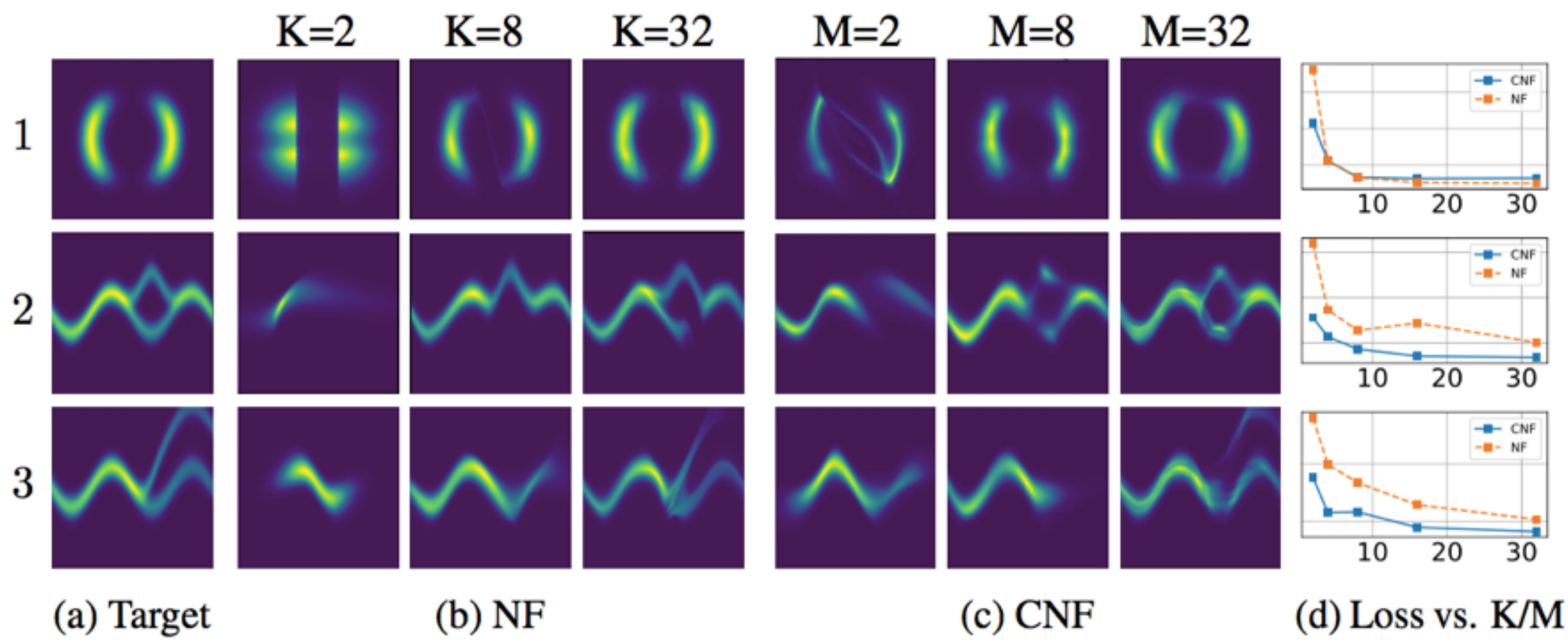
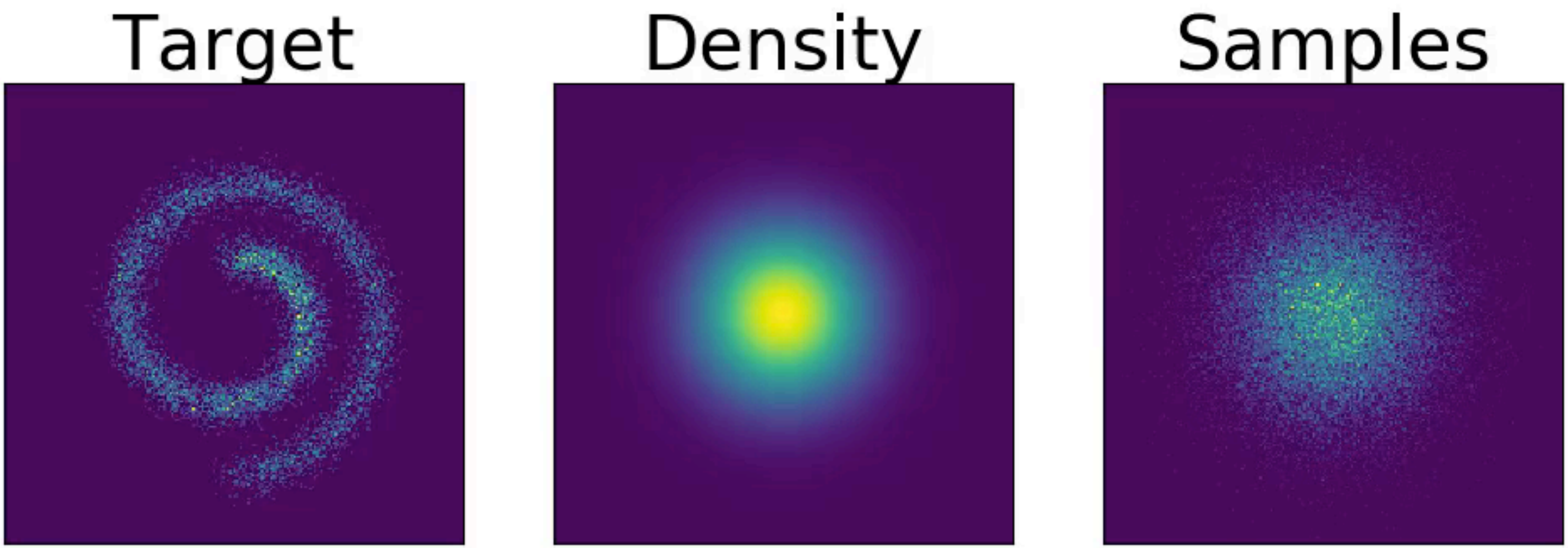


Figure 5: Comparison of NF and CNFs on learning generative models (noise → data) trained to minimize the reverse KL.

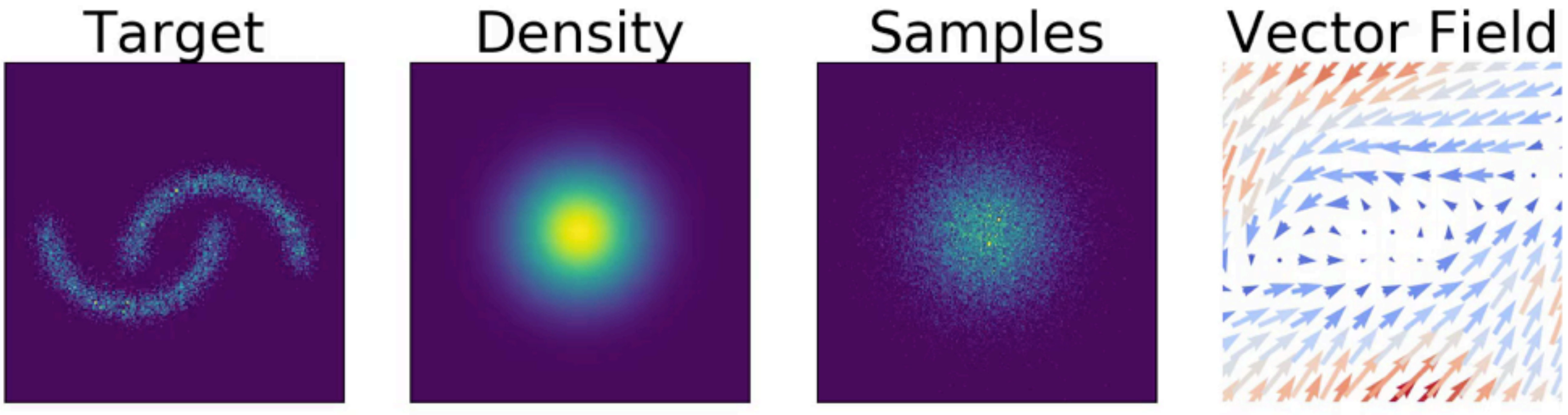
Training directly from data

- Standard NF is one-to-one but expensive to invert.
- Continuous NF is about as easy inverted as forward
- So can train directly from data, like Real NVP



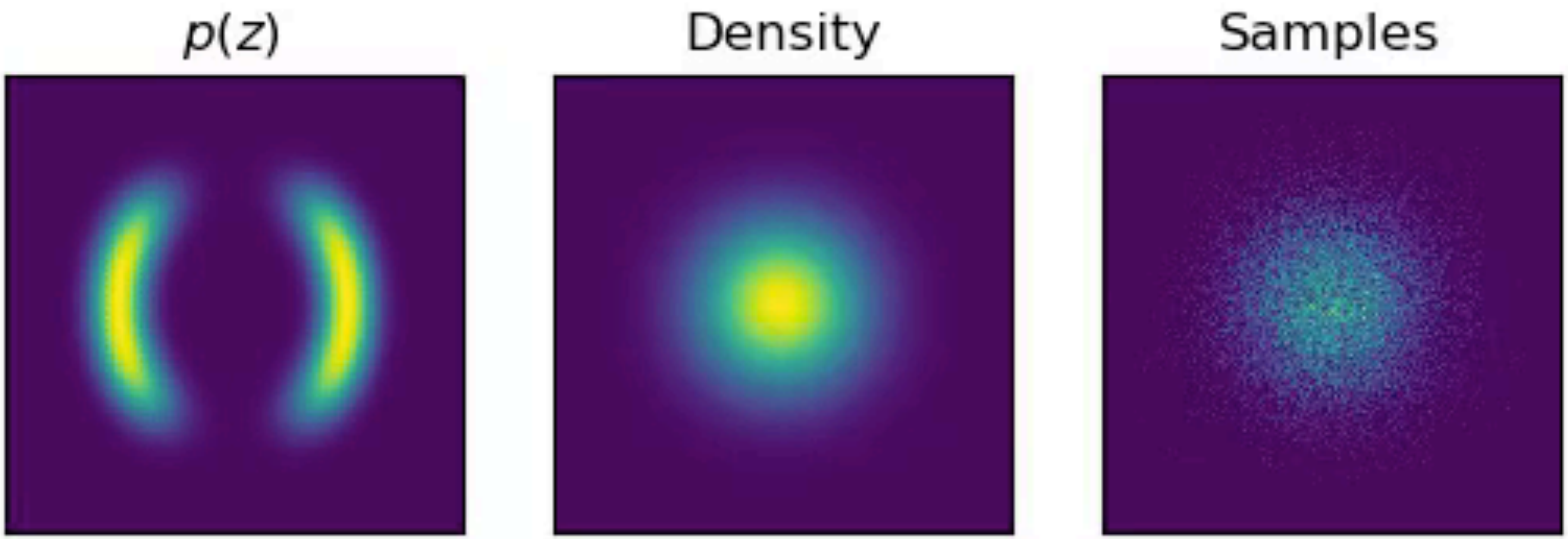
Training directly from data

- Best of all worlds:
  - Wide layers
  - No need to partition dimensions
  - Can evaluate density tractably



What about numerical error?

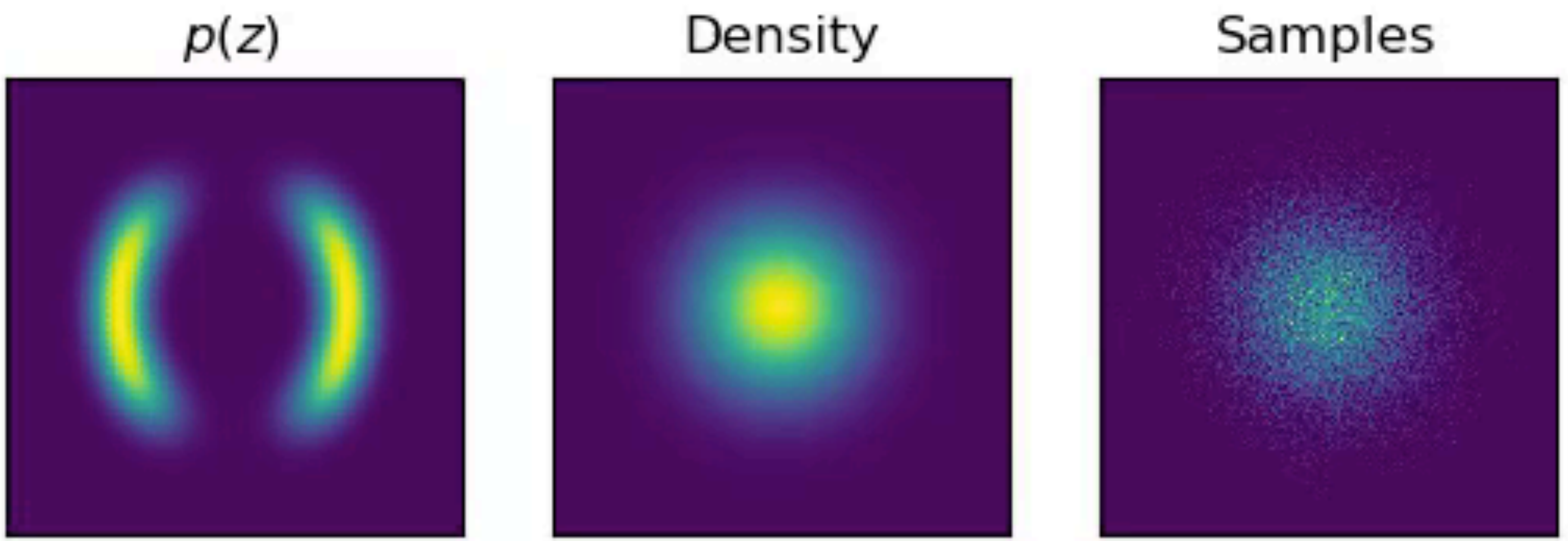
- Are we really inverting exactly?
- Can ask for desired error level.



Absolute and relative tolerance: 0.01

What about numerical error?

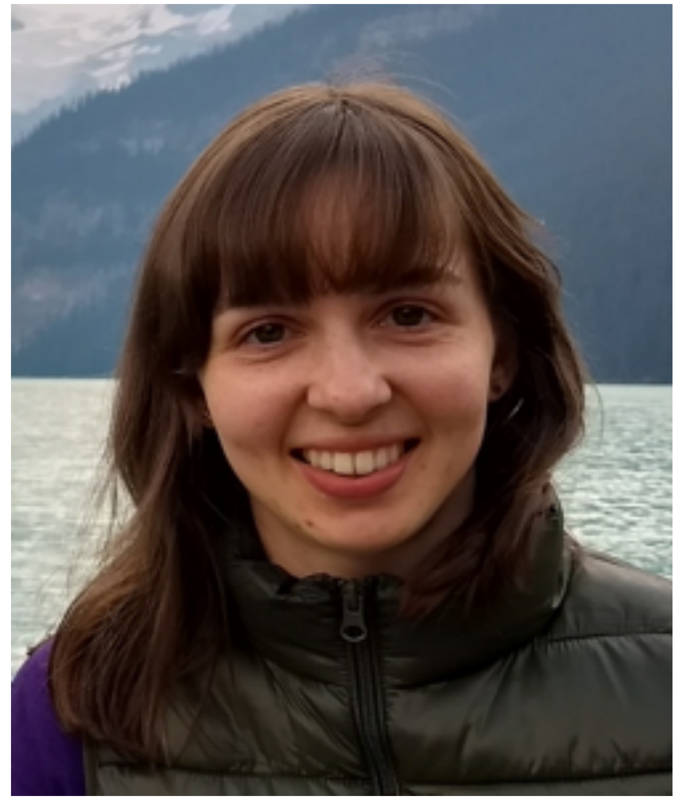
- Are we really inverting exactly?
- Can ask for desired error level.

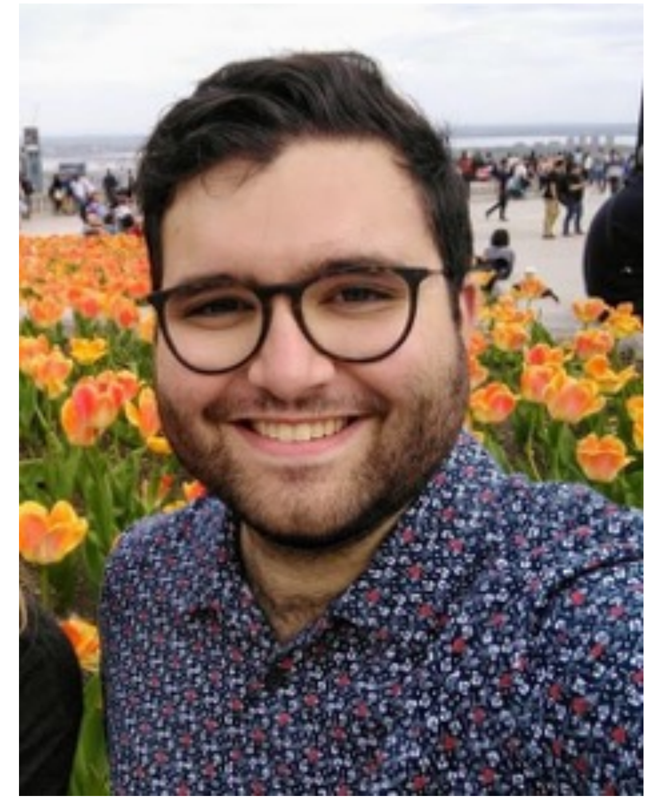


Absolute and relative tolerance: 0.00001

Continuous everything

- Next steps:
  - Pytorch & Tensorflow versions of ODE backprop
  - Scale up continuous normalizing flows
  - Extend time-series model to SDEs
- Other directions:
  - Continuous-time HMC?
  - Backprop through physical simulations?
  - Better neural physics models?
  - More efficient neural architectures??





Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, David Duvenaud

Thanks!

VELUT ARBOR ÆVO

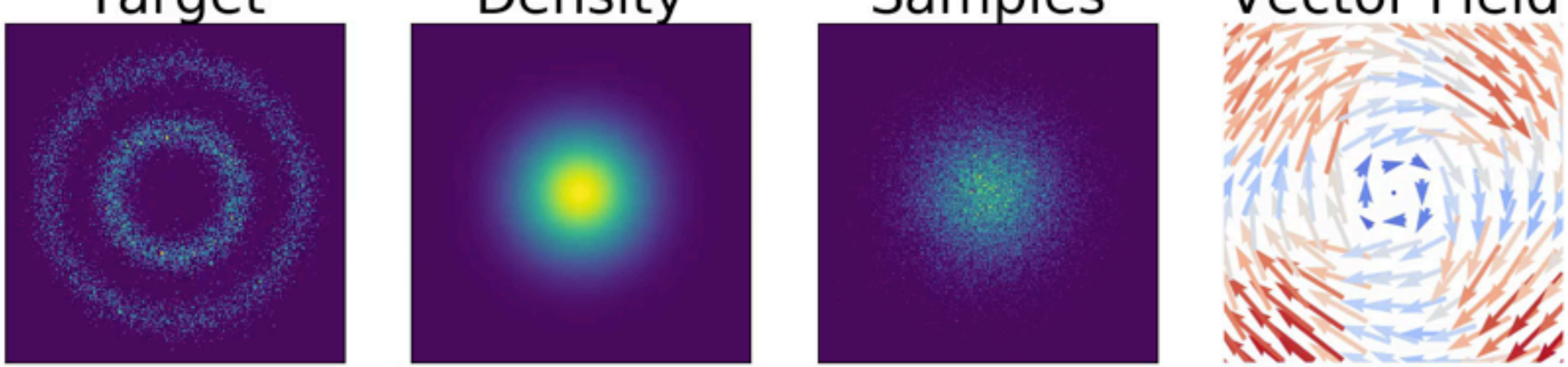
VECTOR INSTITUTE

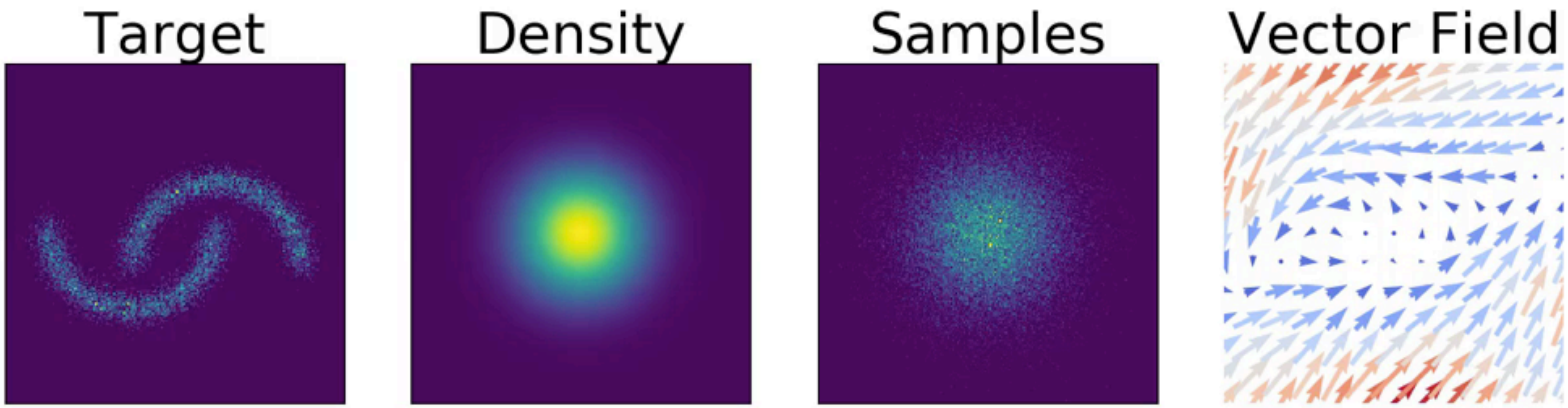
Extra Slides

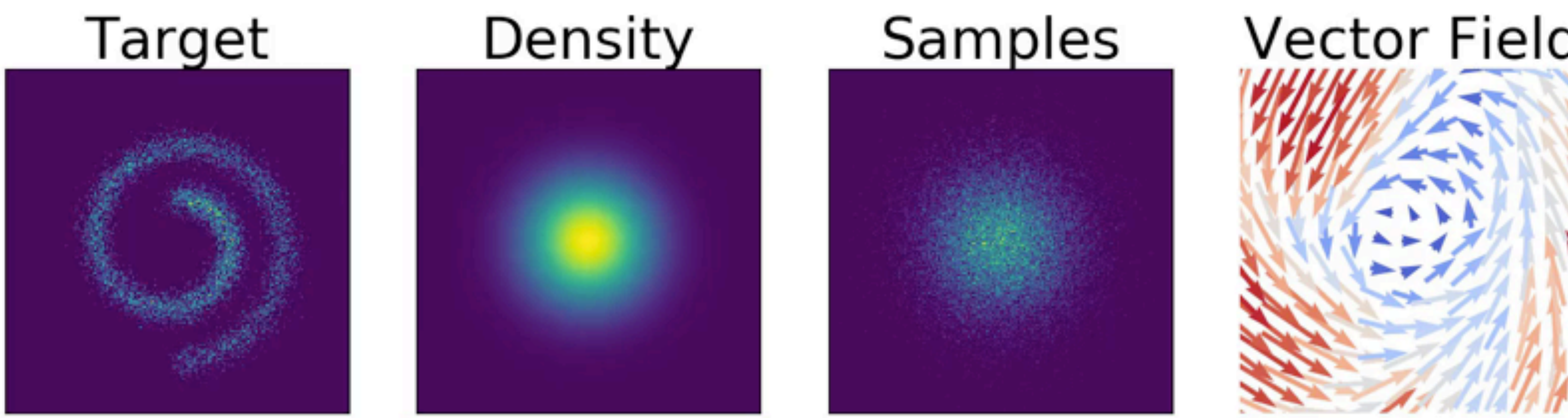
Instantaneous Change of Variables

Theorem 1 (Instantaneous Change of Variables)**.** *Let $x(t)$ be a finite continuous random variable with probability $p(x(t))$ dependent on time. Let $\frac{dx}{dt} = f(x(t), t)$ be a differential equation describing a continuous-time transformation of $x(t)$. Assuming that f is uniformly Lipschitz continuous in x and continuous in t, then the change in log probability also follows a differential equation,*

$$\frac{\partial \log p(x(t))}{\partial t} = -\mathrm{tr}\left(\frac{df}{dx}(t)\right) \tag{8}$$




Target
Density
Samples
Vector Field


Target
Density
Samples
Vector Field