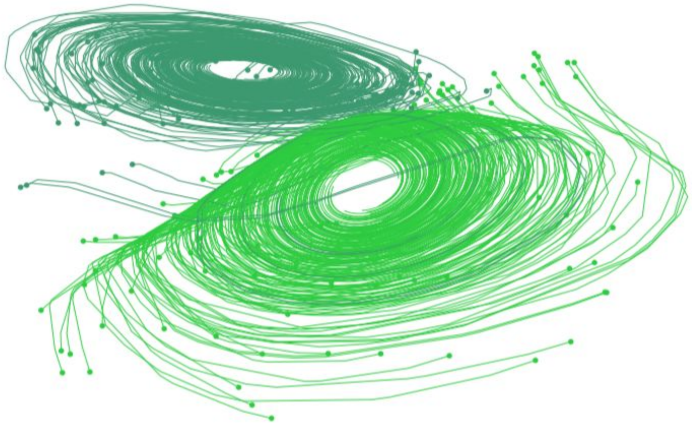
Neural Ordinary Differential Equations



Ricky T. Q. Chen*, Yulia Rubanova*, Jesse Bettencourt*, David Duvenaud

University of Toronto

Background: Ordinary Differential Equations (ODEs)

- Model the instantaneous change of a state.

$$\frac{dz(t)}{dt} = f(z(t), t) \qquad \text{(explicit form)}$$

- Solving an **initial value problem** (IVP) corresponds to integration.

$$z(t) = z(t_0) + \int_{t_0}^{t} f(z(t), t)dt \qquad \text{(solution is a trajectory)}$$

- Euler method approximates with small steps:

$$z(t+h) = z(t) + hf(z(t), t)$$

Residual Networks interpreted as an ODE Solver

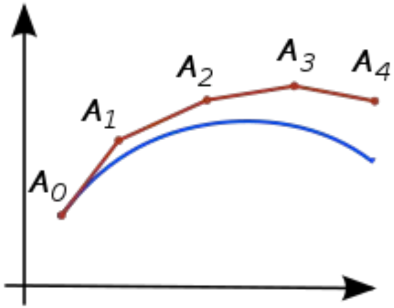
- Hidden units look like: $z_{l+1} = F_l(z_l) = z_l + f_l(z_l)$
- Final output is the composition: $z_L = F_{L-1} \circ F_{L-2} \cdots \circ F_0(z_0)$

Haber & Ruthotto (2017). E (2017).

Residual Networks interpreted as an ODE Solver

- Hidden units look like: $z_{l+1} = F_l(z_l) = z_l + f_l(z_l)$
- Final output is the composition: $z_L = F_{L-1} \circ F_{L-2} \cdots \circ F_0(z_0)$

- This can be interpreted as an **Euler discretization** of an ODE.



- In the limit of smaller steps: $\frac{dz(t)}{dt} = \lim_{h \to 0} \frac{z_{t+h} - z_t}{h} = f(z_t)$

Haber & Ruthotto (2017). E (2017).

Deep Learning as Discretized Differential Equations

Many deep learning networks can be interpreted as ODE solvers.

Network	**Fixed-step Numerical Scheme**
ResNet, RevNet, ResNeXt, etc.	Forward Euler
PolyNet	Approximation to Backward Euler
FractalNet	Runge-Kutta
DenseNet	Runge-Kutta

Lu et al. (2017)
Chang et al. (2018)
Zhu et al. (2018)

Deep Learning as Discretized Differential Equations

Many deep learning networks can be interpreted as ODE solvers.

Network	**Fixed-step Numerical Scheme**
ResNet, RevNet, ResNeXt, etc.	Forward Euler
PolyNet	Approximation to Backward Euler
FractalNet	Runge-Kutta
DenseNet	Runge-Kutta

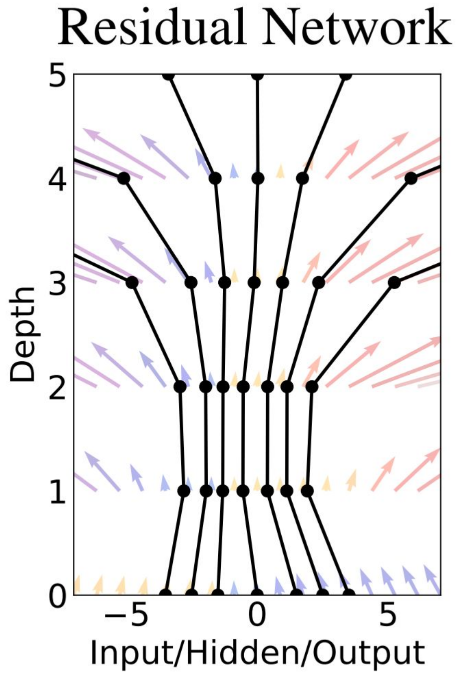
Lu et al. (2017)
Chang et al. (2018)
Zhu et al. (2018)

But:

(1) What is the underlying dynamics?
(2) Adaptive-step size solvers provide better error handling.

“Neural” Ordinary Differential Equations

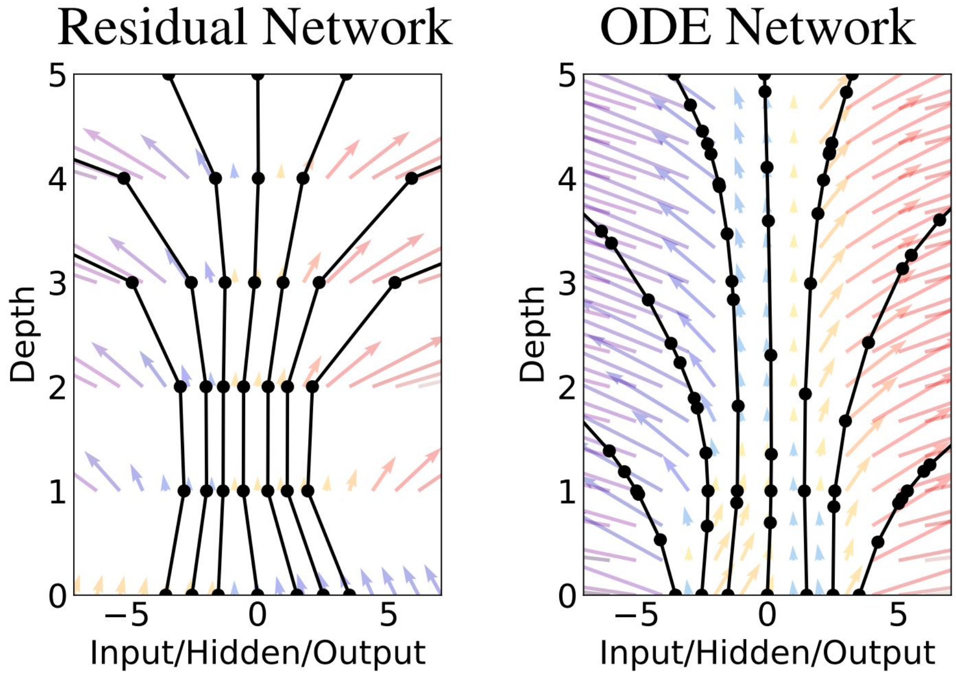
Instead of **y = F(x)**,



“Neural” Ordinary Differential Equations

Instead of **y = F(x)**, solve **y = z(T)** given the initial condition **z(0) = x**.

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



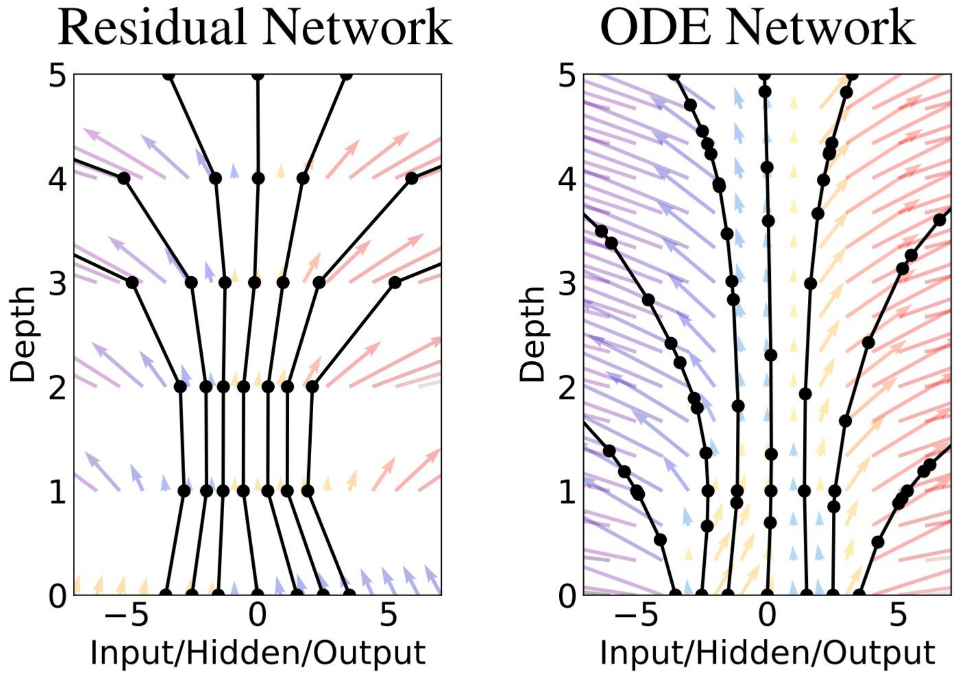
“Neural” Ordinary Differential Equations

Instead of **y = F(x)**, solve **y = z(T)** given the initial condition **z(0) = x**.

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

Solve the dynamic using **any black-box ODE solver**.

- Adaptive step size.
- Error estimate.
- O(1) memory learning.



Backprop without knowledge of the ODE Solver

Ultimately want to optimize some loss

$$L(z(T)) = L\left(z(t_0) + \int_{t_0}^{T} f(z(t), t, \theta)dt\right) = L\left(\text{ODESolve}(z(t_0), t_0, T, \theta)\right)$$

$$\frac{\partial L}{\partial \theta} = ?$$

Backprop without knowledge of the ODE Solver

Ultimately want to optimize some loss

$$L(z(T)) = L\left(z(t_0) + \int_{t_0}^{T} f(z(t), t, \theta) dt\right) = L\left(\text{ODESolve}(z(t_0), t_0, T, \theta)\right)$$

Naive approach: Know the solver. Backprop through the solver.

- Memory-intensive.
- Family of “implicit” solvers perform inner optimization.

Backprop without knowledge of the ODE Solver

Ultimately want to optimize some loss

$$L(z(T)) = L\left(z(t_0) + \int_{t_0}^{T} f(z(t), t, \theta)dt\right) = L\left(\text{ODESolve}(z(t_0), t_0, T, \theta)\right)$$

Naive approach: Know the solver. Backprop through the solver.

- Memory-intensive.
- Family of “implicit” solvers perform inner optimization.

Our approach: **Adjoint sensitivity analysis**. (Reverse-mode Autodiff.)

- Pontryagin (1962).
    + Automatic differentiation.
    + O(1) memory in backward pass.

Continuous-time Backpropagation

Residual network. $a_t := \frac{\partial L}{\partial z_t}$

Forward: $z_{t+h} = z_t + hf(z_t)$

Backward: $a_t = a_{t+h} + ha_{t+h}\frac{\partial f(z_t)}{\partial z_t}$

Params: $\frac{\partial L}{\partial \theta} = ha_{t+h}\frac{\partial f(z(t), \theta)}{\partial \theta}$

Adjoint method.

Define: $a(t) := \frac{\partial L}{\partial z(t)}$

Continuous-time Backpropagation

Residual network. $a_t := \frac{\partial L}{\partial z_t}$

Forward: $z_{t+h} = z_t + hf(z_t)$

Backward: $a_t = a_{t+h} + ha_{t+h}\frac{\partial f(z_t)}{\partial z_t}$

Params: $\frac{\partial L}{\partial \theta} = ha_{t+h}\frac{\partial f(z(t), \theta)}{\partial \theta}$

Adjoint method. Define: $a(t) := \frac{\partial L}{\partial z(t)}$

Forward: $z(t+1) = z(t) + \int_t^{t+1} f(z(t))\, dt$

Continuous-time Backpropagation

Residual network.

$$a_t := \frac{\partial L}{\partial z_t}$$

Forward: $z_{t+h} = z_t + hf(z_t)$

Backward: $a_t = a_{t+h} + ha_{t+h}\frac{\partial f(z_t)}{\partial z_t}$

Params: $\frac{\partial L}{\partial \theta} = ha_{t+h}\frac{\partial f(z(t), \theta)}{\partial \theta}$

Adjoint method.

Define: $a(t) := \frac{\partial L}{\partial z(t)}$

Forward: $z(t+1) = z(t) + \int_t^{t+1} f(z(t))\, dt$

Backward: $a(t) = a(t+1) + \int_{t+1}^{t} a(t)\frac{\partial f(z(t))}{\partial z(t)} dt$

Adjoint State

Adjoint DiffEq

Continuous-time Backpropagation

Residual network.

$a_t := \frac{\partial L}{\partial z_t}$

Forward: $z_{t+h} = z_t + hf(z_t)$

Backward: $a_t = a_{t+h} + ha_{t+h}\frac{\partial f(z_t)}{\partial z_t}$

Params: $\frac{\partial L}{\partial \theta} = ha_{t+h}\frac{\partial f(z(t),\theta)}{\partial \theta}$

Adjoint method.

Define: $a(t) := \frac{\partial L}{\partial z(t)}$

Forward: $z(t+1) = z(t) + \int_t^{t+1} f(z(t))\, dt$

Backward: $a(t) = a(t+1) + \int_{t+1}^{t} a(t)\frac{\partial f(z(t))}{\partial z(t)} dt$

Adjoint State

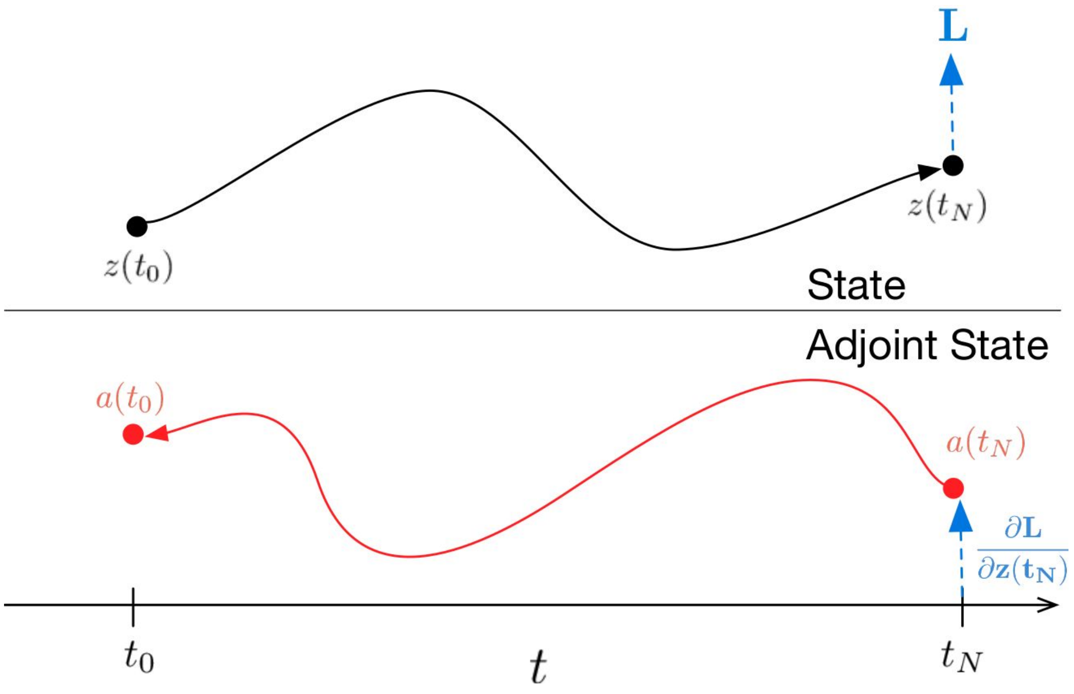
Adjoint DiffEq

Params: $\frac{\partial L}{\partial \theta} = \int_t^{t+1} a(t)\frac{\partial f(z(t),\theta)}{\partial \theta}\, dt$

A Differentiable Primitive for AutoDiff

Forward:

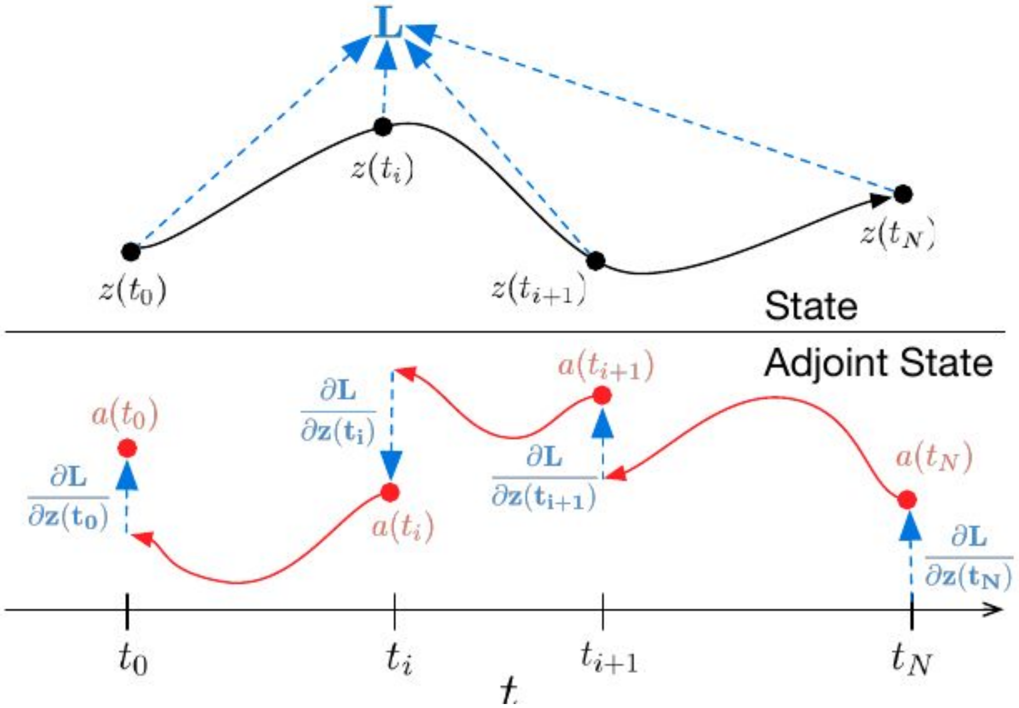
Backward:



A Differentiable Primitive for AutoDiff

Forward:

Backward:



A Differentiable Primitive for AutoDiff

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

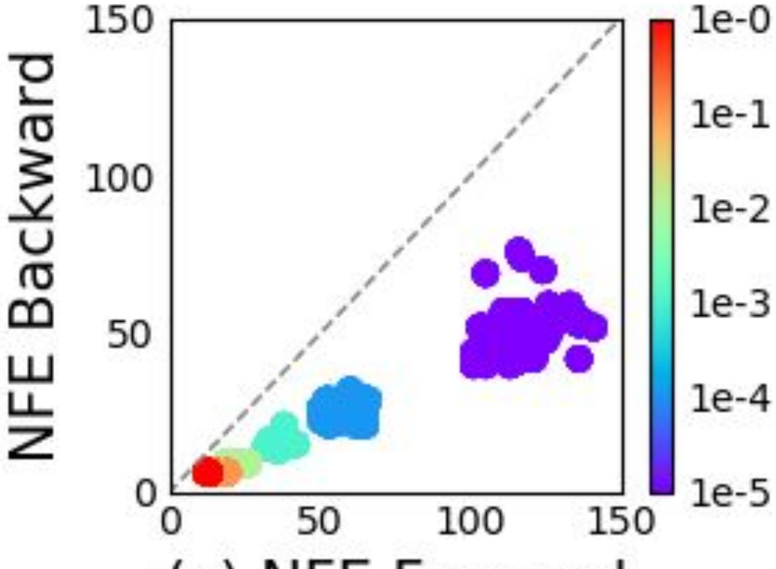
Table 1: Performance on MNIST. †From LeCun et al. (1998).

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

Reversible networks (Gomez et al. 2018) also only require O(1)-memory, but require very specific neural network architectures with partitioned dimensions.

Reverse versus Forward Cost

- Empirically, reverse pass roughly half as expensive as forward pass.
- Adapts to instance difficulty.
- Num evaluations can be viewed as number of layers in neural nets.

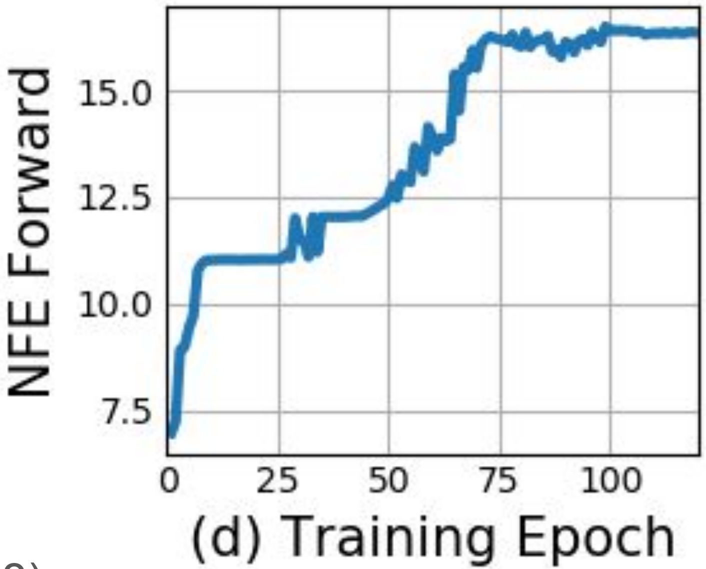


(c) NFE Forward

NFE = Number of Function Evaluations.

Dynamics Become Increasingly Complex

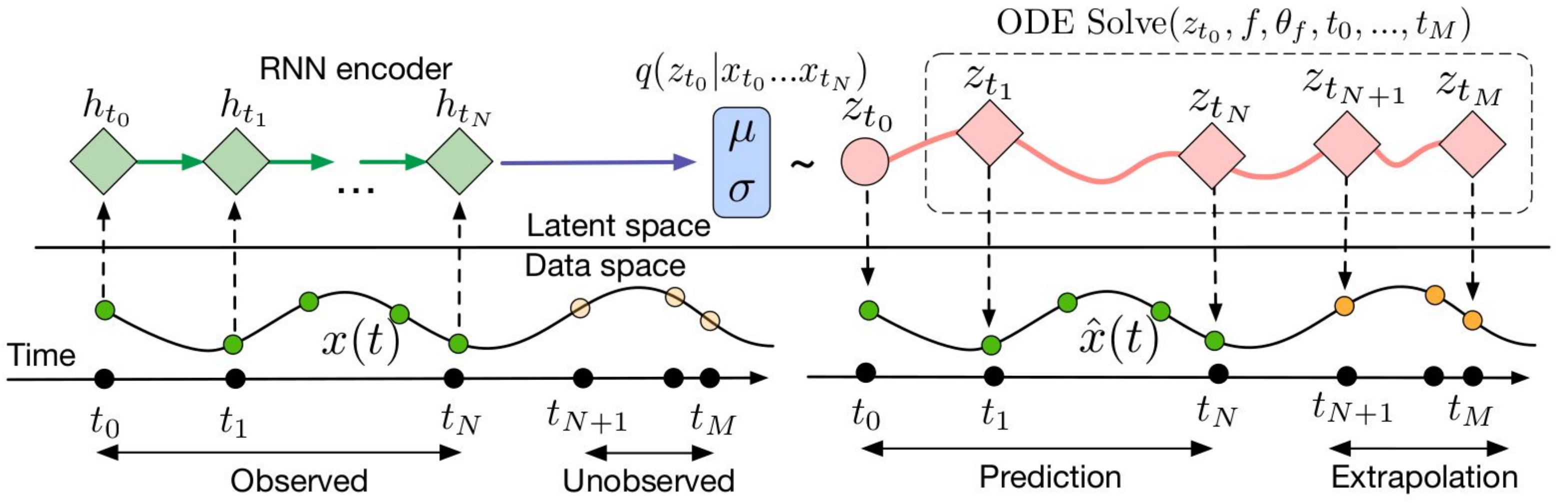
- Dynamics become more demanding to compute during training.
- Adapts computation time according to complexity of diffeq.



In contrast, Chang et al. (ICLR 2018) explicitly add layers during training.

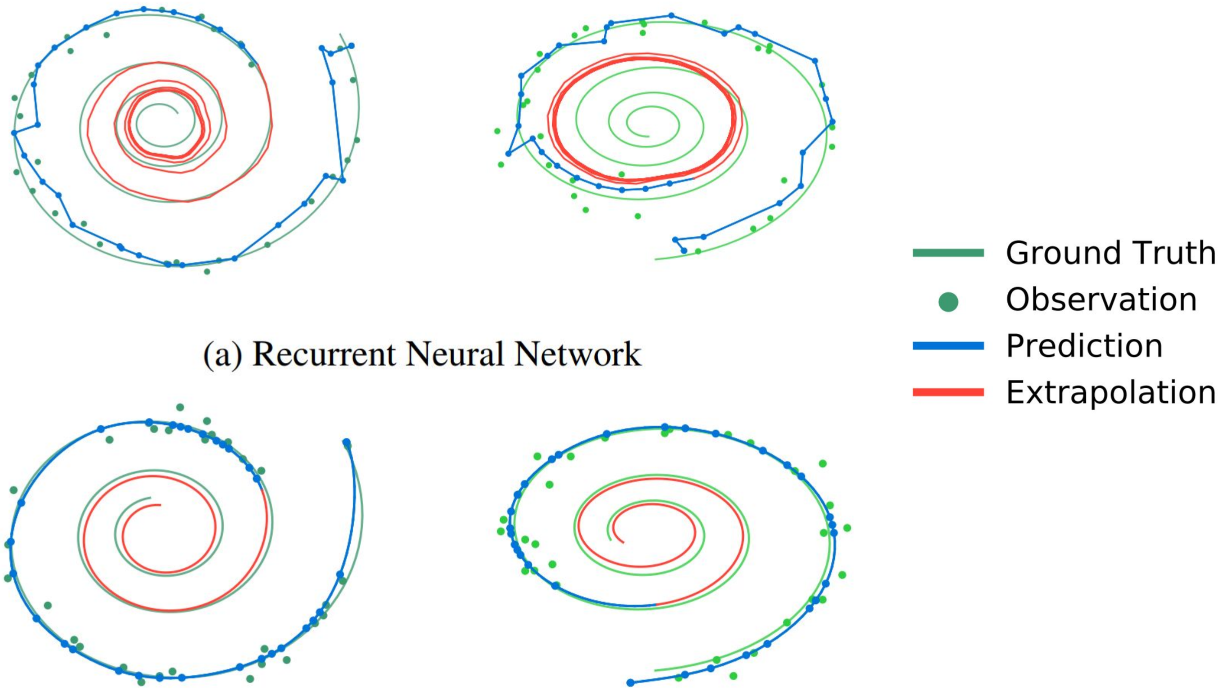
Continuous-time RNNs for Time Series Modeling

- We often want arbitrary measurement times, ie. irregular time intervals.
- Can do VAE-style inference with a latent ODE.



ODEs vs Recurrent Neural Networks (RNNs)

- RNNs learn very stiff dynamics, have exploding gradients.
- Whereas ODEs are guaranteed to be smooth.



(a) Recurrent Neural Network

(b) Latent Neural Ordinary Differential Equation

Continuous Normalizing Flows

Instantaneous Change of variables (iCOV):

- For a Lipschitz continuous function f

$$\frac{dh}{dt} = f(h(t), t) \implies \frac{\partial \log p(h(t))}{\partial t} = -\mathrm{tr}\left(\frac{\partial f}{\partial h(t)}\right)$$

Continuous Normalizing Flows

Instantaneous Change of variables (iCOV):

- For a Lipschitz continuous function f

$$\frac{dh}{dt} = f(h(t), t) \implies \frac{\partial \log p(h(t))}{\partial t} = -\mathrm{tr}\left(\frac{\partial f}{\partial h(t)}\right)$$

- In other words,

$$h(t_0) = x, h(t_1) = z \implies \log p(x) = \log p(z) + \int_{t_0}^{t_1} \mathrm{tr}\left(\frac{\partial f}{\partial h(t)}\right)$$

Continuous Normalizing Flows

Instantaneous Change of variables (iCOV):

- For a Lipschitz continuous function f

$$\frac{dh}{dt} = f(h(t), t) \implies \frac{\partial \log p(h(t))}{\partial t} = -\mathrm{tr}\left(\frac{\partial f}{\partial h(t)}\right)$$

- In other words,

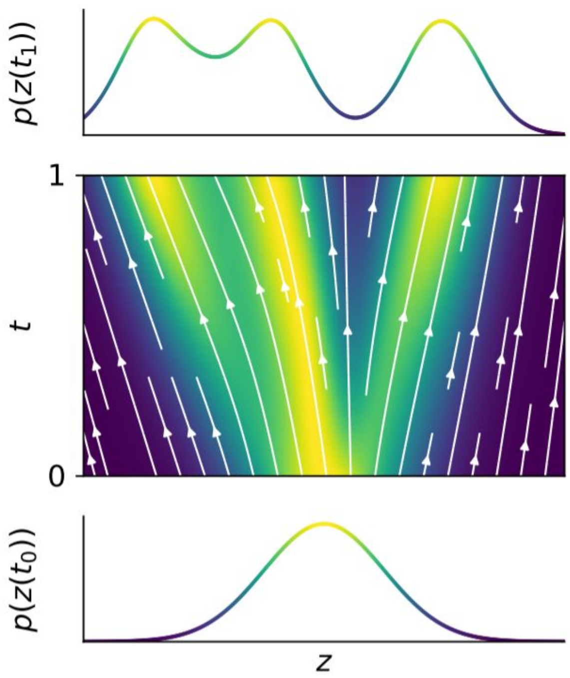
$$h(t_0) = x, h(t_1) = z \implies \log p(x) = \log p(z) + \int_{t_0}^{t_1} \mathrm{tr}\left(\frac{\partial f}{\partial h(t)}\right)$$

With an invertible F:

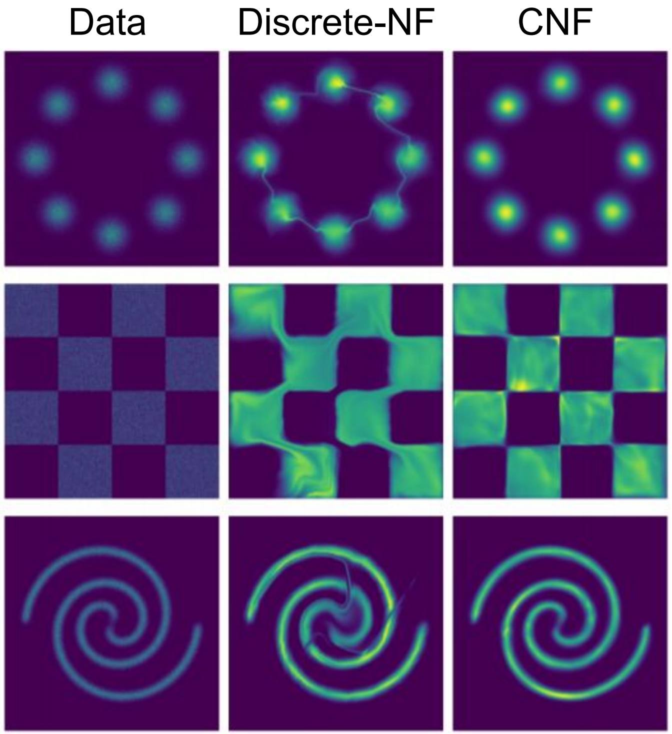
$$F(x) = z \implies \log p(x) = \log p(z) + \log\left|\det\frac{\partial F}{\partial x}\right|$$

Continuous Normalizing Flows

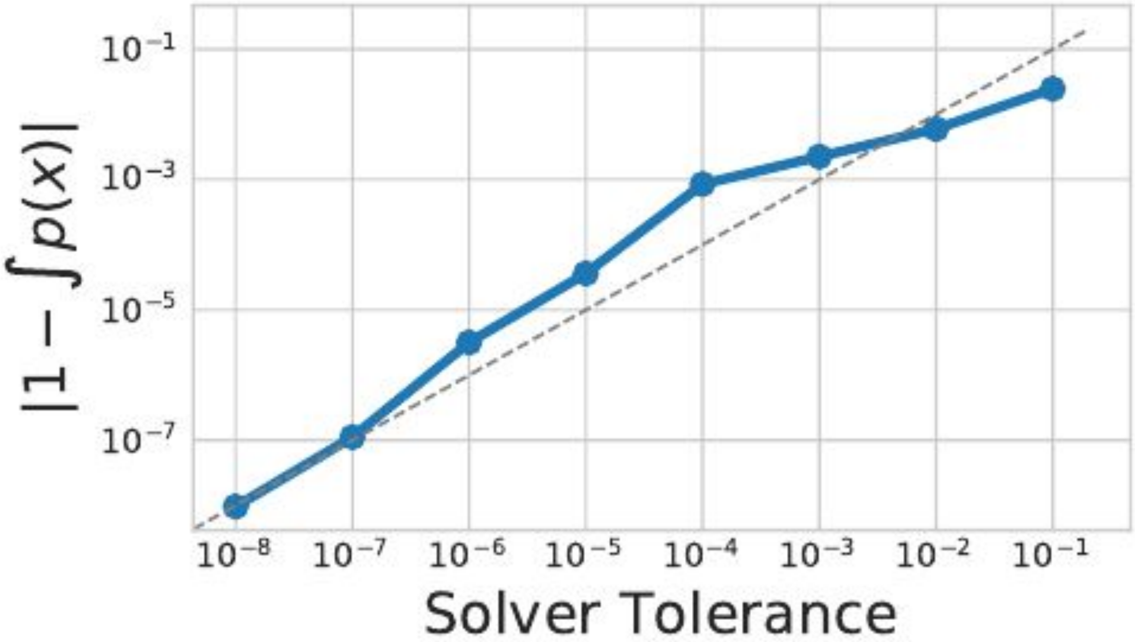
1D:



2D:



Is the ODE being correctly solved?



Stochastic Unbiased Log Density

$$\log p(x) = \log p(z) + \int_{t_0}^{t_1} \operatorname{tr}\left(\frac{\partial f}{\partial h(t)}\right) \quad \in \mathcal{O}(D^2)$$

Stochastic Unbiased Log Density

$$\log p(x) = \log p(z) + \int_{t_0}^{t_1} \operatorname{tr}\left(\frac{\partial f}{\partial h(t)}\right) \qquad \in \mathcal{O}(D^2)$$

Can further reduce time complexity using **stochastic** estimators.

$$\operatorname{tr}(A) = \mathbb{E}[\underbrace{v^T A v}_{\text{trace estimator}}] \quad \text{if } \mathbb{E}[vv^T] = I$$

$$\int_{t_0}^{t_1} \operatorname{tr}\left(\frac{\partial f}{\partial h(t)}\right) = \int_{t_0}^{t_1} \underbrace{\mathbb{E}\left[v^T \frac{\partial f}{\partial h(t)} v\right]}_{\text{Not an ODE}} = \mathbb{E}\left[\int_{t_0}^{t_1} v^T \frac{\partial f}{\partial h(t)} v\right]$$

$$\in \mathcal{O}(D)$$

Grathwohl et al. (2019)

FFJORD - Stochastic Continuous Flows

MNIST - Model Samples

CIFAR10 - Model Samples

Grathwohl et al. (2019)

Variational Autoencoders with FFJORD

$$\text{ELBO}(x) = \mathbb{E}_{q(z|x)}[\log p(x|z) + \log p(z) - \log q(z|x)]$$
$$= \mathbb{E}_{p(v)q(z|x)}[\log p(x|z) + \log p(z) - \log q(z|x, v)]$$

	MNIST	Omniglot	Frey Faces	Caltech Silhouettes
No Flow	$86.55 \pm .06$	$104.28 \pm .39$	$4.53 \pm .02$	$110.80 \pm .46$
Planar	$86.06 \pm .31$	$102.65 \pm .42$	$4.40 \pm .06$	$109.66 \pm .42$
IAF	$84.20 \pm .17$	$102.41 \pm .04$	$4.47 \pm .05$	$111.58 \pm .38$
Sylvester	$83.32 \pm .06$	$99.00 \pm .04$	$4.45 \pm .04$	$104.62 \pm .29$
FFJORD	$\mathbf{82.82} \pm \mathbf{.01}$	$\mathbf{98.33} \pm \mathbf{.09}$	$\mathbf{4.39} \pm \mathbf{.01}$	$\mathbf{104.03} \pm \mathbf{.43}$

ODE Solving as a Modeling Primitive

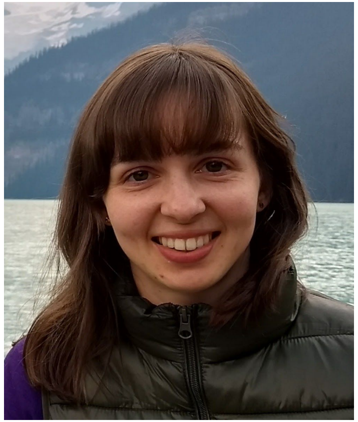
Adaptive-step solvers with O(1) memory backprop.

github.com/rtqichen/torchdiffeq

Future directions we're currently working on:

- Latent Stochastic Differential Equations.
- Network architectures suited for ODEs.
- Regularization of dynamics to require fewer evaluations.

Co-authors:



Yulia Rubanova

Jesse Bettencourt

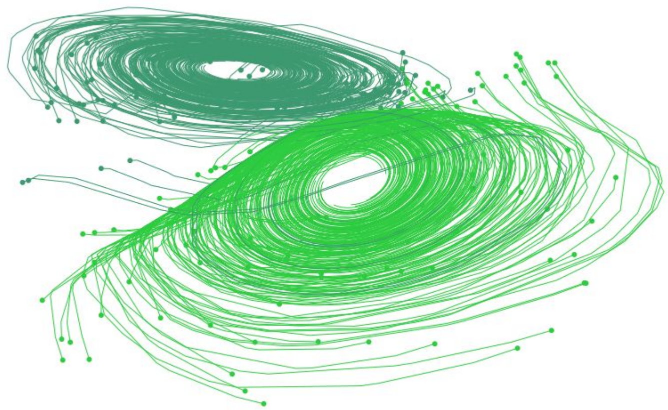
David Duvenaud

Thanks!

VECTOR INSTITUTE

Extra Slides

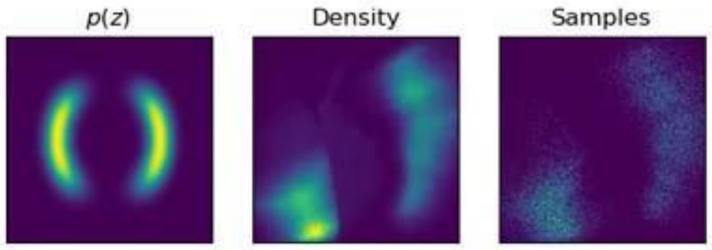
Latent Space Visualizations

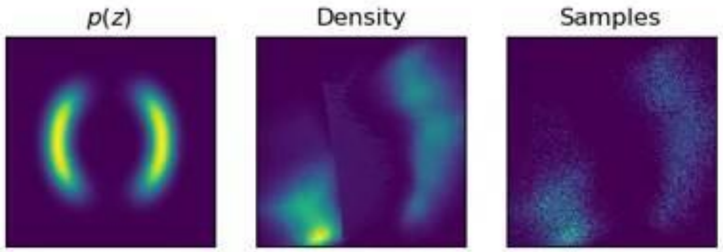


```python
def grad_odeint(yt, func, y0, t, func_args, **kwargs):
    # Extended from "Scalable Inference of Ordinary Differential
    # Equation Models of Biochemical Processes", Sec. 2.4.2
    # Fabian Froehlich, Carolin Loos, Jan Hasenauer, 2017
    # https://arxiv.org/abs/1711.08079

    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
```

- Released an implementation of reverse-mode autodiff through black-box ODE solvers.
- Solves a system of size 2D + K + 1.
- In contrast, forward-mode implementation solves a system of size D^2 + KD.
- Tensorflow has Dormand-Prince-Shampine Runge-Kutta 5(4) implemented, but uses naive autodiff for backpropagation.

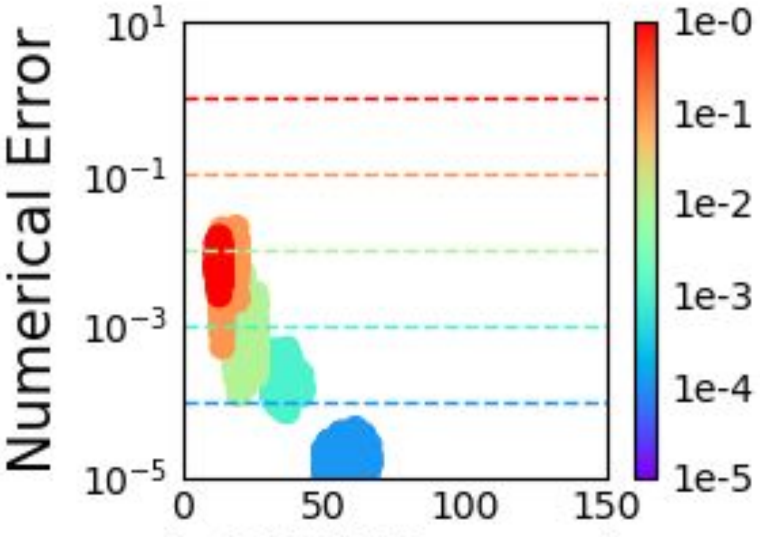
How much precision is needed?





Explicit Error Control

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

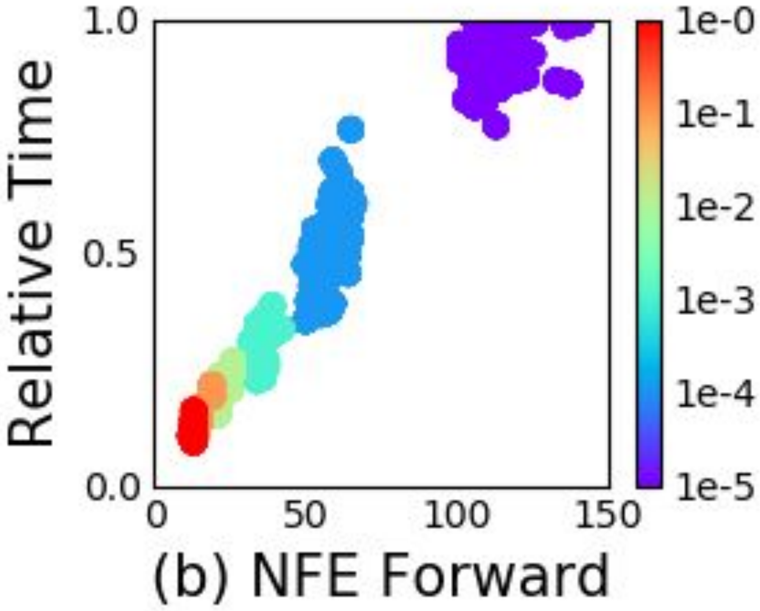


(a) NFE Forward

NFE = Number of Function Evaluations.

Computation Depends on Complexity of Dynamics

- Time cost is dominated by evaluation of dynamics f.



NFE = Number of Function Evaluations.

Why *not* use an ODE solver as modeling primitive?

- Solving an ODE is expensive.

Future Directions

- Stochastic differential equations and Random ODEs. Approximates stochastic gradient descent.
- Scaling up ODE solvers with machine learning.
- Partial differential equations.
- Graphics, physics, simulations.