Background: Ordinary Differential Equations (ODEs)

- Model the instantaneous change of a state.
  \[
  \frac{dz(t)}{dt} = f(z(t), t) \quad \text{(explicit form)}
  \]

- Solving an **initial value problem** (IVP) corresponds to integration.
  \[
  z(t) = z(t_0) + \int_{t_0}^{t} f(z(t), t) dt \quad \text{(solution is a trajectory)}
  \]

- 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: $z_L = F_{L-1} \circ F_{L-2} \cdots \circ F_0(z_0)$

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: \( \frac{dz(t)}{dt} = \lim_{h \to 0} \frac{z_{t+h} - z_t}{h} = f(z_t) \)

Deep Learning as Discretized Differential Equations

Many deep learning networks can be interpreted as ODE solvers.

<table>
<thead>
<tr>
<th>Network</th>
<th>Fixed-step Numerical Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet, RevNet, ResNeXt, etc.</td>
<td>Forward Euler</td>
</tr>
<tr>
<td>PolyNet</td>
<td>Approximation to Backward Euler</td>
</tr>
<tr>
<td>FractalNet</td>
<td>Runge-Kutta</td>
</tr>
<tr>
<td>DenseNet</td>
<td>Runge-Kutta</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Network</th>
<th>Fixed-step Numerical Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet, RevNet, ResNeXt, etc.</td>
<td>Forward Euler</td>
</tr>
<tr>
<td>PolyNet</td>
<td>Approximation to Backward Euler</td>
</tr>
<tr>
<td>FractalNet</td>
<td>Runge-Kutta</td>
</tr>
<tr>
<td>DenseNet</td>
<td>Runge-Kutta</td>
</tr>
</tbody>
</table>

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 $\frac{dz(t)}{dt} = f(z(t), \theta(t))$. 
“Neural” Ordinary Differential Equations

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

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

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

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

\[ \frac{\partial L}{\partial \theta} = ? \]
Backprop without knowledge of the ODE Solver

Ultimately want to optimize some loss

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

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

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

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

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

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

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

**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: \( a(t) = a(t+1) + \int_t^{t+1} a(t) \frac{\partial f(z(t))}{\partial z(t)} \, dt \)

\[ \text{Adjoint State} \]
\[ \text{Adjoint DiffEq} \]
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:** \( a(t) = a(t+1) + \int_t^{t+1} a(t) \frac{\partial f(z(t))}{\partial z(t)} \, dt \)

[**Adjoint State**] \( a(t) \)

[**Adjoint DiffEq**] \( \int_t^{t+1} a(t) \frac{\partial f(z(t), \theta)}{\partial \theta} \, dt \)

**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
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).

<table>
<thead>
<tr>
<th></th>
<th>Test Error</th>
<th>Memory</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-Layer MLP†</td>
<td>1.60%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ResNet</td>
<td>0.41%</td>
<td>$O(L)$</td>
<td>$O(L)$</td>
</tr>
<tr>
<td>RK-Net</td>
<td>0.47%</td>
<td>$\tilde{O}(L)$</td>
<td>$O(\tilde{L})$</td>
</tr>
<tr>
<td>ODE-Net</td>
<td>0.42%</td>
<td>$O(1)$</td>
<td>$O(\tilde{L})$</td>
</tr>
</tbody>
</table>

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.

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, i.e. 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.
Continuous Normalizing Flows

Instantaneous Change of variables (iCOV):

- For a Lipschitz continuous function $f$

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

Instantaneous Change of variables (iCOV):

- For a Lipschitz continuous function \( f \)

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

- In other words,

\[
h(t_0) = x, h(t_1) = z \implies \log p(x) = \log p(z) + \int_{t_0}^{t_1} \text{tr} \left( \frac{\partial f}{\partial h(t)} \right)
\]
Continuous Normalizing Flows

Instantaneous Change of variables (iCOV):

- For a Lipschitz continuous function $f$

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

- In other words,

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

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:

Data  Discrete-NF  CNF
Is the ODE being correctly solved?
Stochastic Unbiased Log Density

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

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

Can further reduce time complexity using stochastic estimators.

\[ \text{tr}(A) = \mathbb{E} \left[ v^T A v \right] \quad \text{if} \quad \mathbb{E}[vv^T] = I \]

trace estimator

\[ \int_{t_0}^{t_1} \text{tr} \left( \frac{\partial f}{\partial h(t)} \right) = \int_{t_0}^{t_1} \mathbb{E} \left[ v^T \frac{\partial f}{\partial h(t)} v \right] = \mathbb{E} \left[ \int_{t_0}^{t_1} v^T \frac{\partial f}{\partial h(t)} v \right] \in \mathcal{O}(D) \]

Grathwohl et al. (2019)
FFJORD - Stochastic Continuous Flows

Grathwohl et al. (2019)
Variational Autoencoders with FFJORD

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

<table>
<thead>
<tr>
<th></th>
<th>MNIST</th>
<th>Omniglot</th>
<th>Frey Faces</th>
<th>Caltech Silhouettes</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Flow</td>
<td>86.55 ± .06</td>
<td>104.28 ± .39</td>
<td>4.53 ± .02</td>
<td>110.80 ± .46</td>
</tr>
<tr>
<td>Planar</td>
<td>86.06 ± .31</td>
<td>102.65 ± .42</td>
<td>4.40 ± .06</td>
<td>109.66 ± .42</td>
</tr>
<tr>
<td>IAF</td>
<td>84.20 ± .17</td>
<td>102.41 ± .04</td>
<td>4.47 ± .05</td>
<td>111.58 ± .38</td>
</tr>
<tr>
<td>Sylvester</td>
<td>83.32 ± .06</td>
<td>99.00 ± .04</td>
<td>4.45 ± .04</td>
<td>104.62 ± .29</td>
</tr>
<tr>
<td>FFJORD</td>
<td><strong>82.82 ± .01</strong></td>
<td><strong>98.33 ± .09</strong></td>
<td><strong>4.39 ± .01</strong></td>
<td><strong>104.03 ± .43</strong></td>
</tr>
</tbody>
</table>
ODA Solving as a Modeling Primitive

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

github.com/rtqichen/torchdiffeq

Future directions we’re currently working on:

- Network architectures suited for ODEs.
- Regularization of dynamics to require fewer evaluations.
Co-authors:

Yulia Rubanova  Jesse Bettencourt  David Duvenaud

Thanks!
Extra Slides
Latent Space Visualizations
- 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.

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.