Neural Ordinary Differential Equations

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

[1]Anurendra Kumar
Computer Science, UIUC

CS 598 DGDM, Class Presentation

Problem Setup: Supervised learning

- Traditional ML: $y = ax + b$
- Neural ODE [1](ODE with IVP): $\frac{dy}{dt} = a, y(0) = x$
- Input: Initial Time Point, Output: Final time point

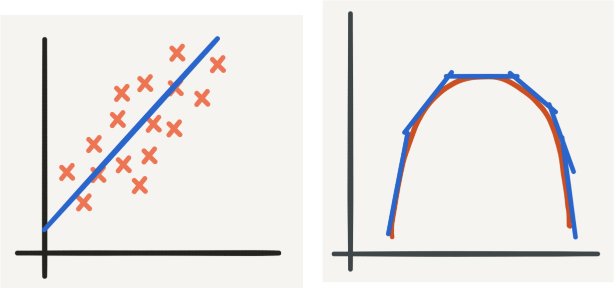


Figure: a)Linear Regression, b)Gradient tracing

[1] https://jontysinai.github.io/jekyll/update/2019/01/18/understanding-neural-odes.html

Background: ODE

- **Physics**: ODEs often used to describe the dynamics.
- **Neural ODEs**: Replace explicit ODEs to learn them via ML.
- **ODE Solvers**: Extensive Research on explicit and implicit solvers.

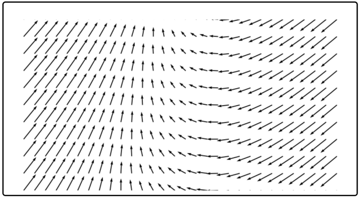


Fig.1: A vector field in 2D space denoting the dynamics of an ODE

Background: Explicit and Implicit ODE Solvers

- $\frac{dy}{dt} = f(t, y(t))$
- **Forward Euler method**: $y_{n+1} = y_n + \delta f(t_n, y_n)$
- **Backward Euler Method**: $y_{n+1} = y_n + \delta f(t_{n+1}, y_{n+1})$
- Forward Euler is an explicit ODESolver and Backward Euler is an implicit ODE Solver.
- Adaptive-step size solvers provide better error handling.
- Sophisticated higher order ODE solvers like Rungakutta exist

Problem Setup: Resnets as an ODE

- Resnet: $\mathbf{h}_{t+1} = \mathbf{h}_t + f(\mathbf{h}_t, \theta_t)$
- Euler Discretization: $\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta)$
- Residual Networks interpreted as an ODE Solver.
- Final output is the composition of all layers.

- **Memory Issues**: Traditionally, each layer with learnable parameters in DNN needs to store its input until the backward pass.

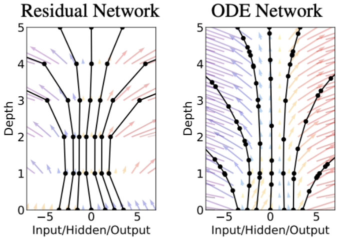


Figure 1: *Left:* A Residual network defines a discrete sequence of finite transformations. *Right:* A ODE network defines a vector field, which continuously transforms the state. *Both:* Circles represent evaluation locations.

“Neural” Ordinary Differential Equations

- Instead of $y = F(x)$, solve, $y = z(t_1)$, given the initial condition $z(0) = x$
- Parameterize $\frac{dz(t)}{dt} = f(z(t), t, \theta)$
- Use existing black box solvers for forward pass.
- Adaptive step size, $O(1)$ memory handling, error estimate

Backprop through Neural ODE

- Ultimately want to optimize some loss

$$L(z(t_1)) = L(z(t_0) + \int_{t_0}^{T} f(z(t), t, \theta) = L(ODESolve(z(t_0), t_0, t_1, \theta)$$

- We want to compute $\frac{dL}{d\theta}$
- Naive approach: Know the solver. Backprop through the solver.
- Problems - Memory-intensive, Family of "implicit" solvers perform inner optimization
- We want backprop without knowledge of the ODE Solver

Adjoint sensitivity analysis: Reverse-mode Autodiff

- The first step is to determining how the gradient of the loss depends on the hidden state $z(t)$ at each instant.
- Define adjoint state $\mathbf{a}(t) = \frac{dL}{d\mathbf{z}(t)}$
- Adjoint follows another ODE,

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

- Recompute $\mathbf{z}(t)$ along with $\mathbf{a}(t)$.
- Another call to an ODE solver. This solver must run backwards, starting from the initial value of $\frac{dL}{d\mathbf{z}(t_1)}$.

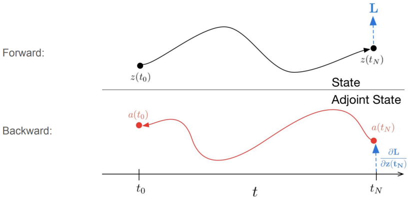
Adjoint sensitivity analysis: Reverse-mode Autodiff

- Third integral which depends on both $\mathbf{z}(t)$ and $\mathbf{a}(t)$

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

- The vector jacobian products $\mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t),t,\theta)}{\partial \mathbf{z}}$ and $\mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t),t,\theta)}{\partial \theta}$ can be computed using automatic differentiation in similar time cost as of f.

Augmented Dynamics [2]



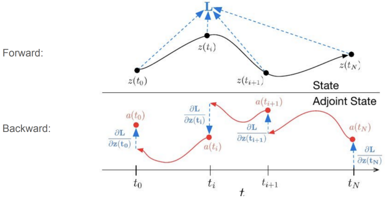


Figure: a)Single Observation time, b)Many Observation Time

[2] https://www.cs.toronto.edu/ rtqichen/pdfs/neural_ode_slides.pdf

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

Neural ODE vs Resnet [3]

Continuous-time Backpropagation

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

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

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

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

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

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

Backward: $\underbrace{a(t)}_{\text{Adjoint State}} = a(t+1) + \displaystyle\int_{t+1}^{t} \underbrace{a(t)\frac{\partial f(z(t))}{\partial z(t)}}_{\text{Adjoint DiffEq}} dt$

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

[2] https://www.cs.toronto.edu/ rtqichen/pdfs/neural_ode_slides.pdf

Results: Neural ODE vs Resnet (Supervised Learning)

- ODENet: Implicit Adams method
- RKNet: Explicit Runge Kutta method
- Similar Performance with Resnet. Low number of parameters and memory.

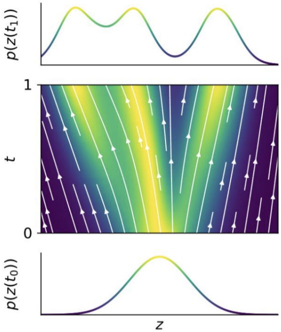
Table 1: Performance on MNIST. †From LeCun et al. (1998).

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

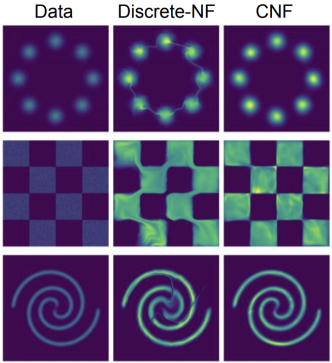
Results: Continuous Normalizing Flows

- Instantaneous change of Formula (See paper)

1D:



2D:



Time Series Latent ODE

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

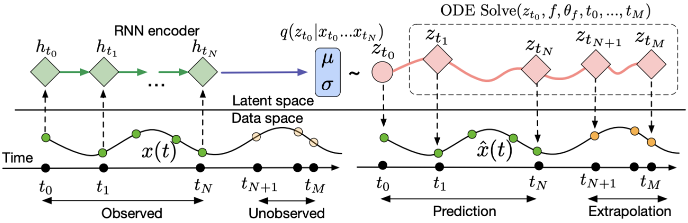
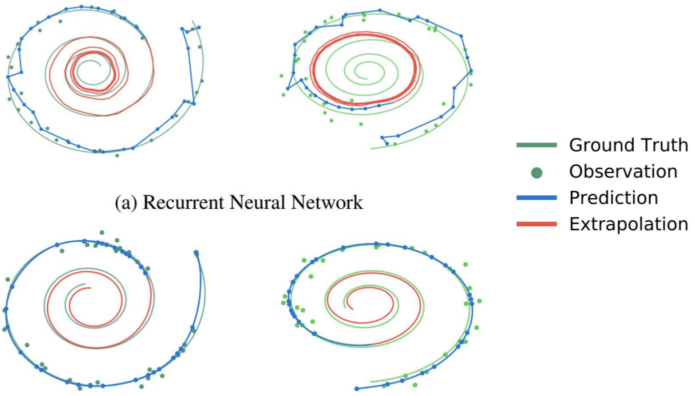


Figure 6: Computation graph of the latent ODE model.

Results: Time Series Latent ODE

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



(a) Recurrent Neural Network

(b) Latent Neural Ordinary Differential Equation

Contributions (Conclusion)

- **Memory efficiency** : Adjoint method to compute gradients of a scalar-valued loss with respect to all inputs of any ODE solver, without backpropagating through the operations of the solver
- **Adaptive computation** : Use SOTA ODE Solvers instead of Euler.
- **Scalable and invertible normalizing flows**
- **Continuous time-series models**

Scope and Limitations (Conclusion)

- **Minibatching** : Use of mini-batches is less straightforward than for standard neural networks.
- **Uniqueness** : solution to an initial value problem exists and is unique if the differential equation is uniformly Lipschitz continuous in z and continuous in t. Holds if the neural network has finite weights and uses Lipshitz nonlinearities, such as tanh or relu.

Reference

- Chen, R. T., Rubanova, Y., Bettencourt, J., Duvenaud, D. (2018). Neural ordinary differential equations. arXiv preprint arXiv:1806.07366.