Neural Ordinary Differential Equations

MLRG Presentation By Jonathan Wilder Lavington

March 28, 2021

University of British Columbia,
Department of Computer Science

What will we talk about today

1. Implicit functions and Auto-diff
2. Deep Equilibrium Models
3. Neural ODEs

Important Note

These slides were built from my favorite parts of the Neurips 2020 tutorial found here: https://www.youtube.com/watch?v=QxtlwOV3c9M&ab_channel=ArtificialIntelligence

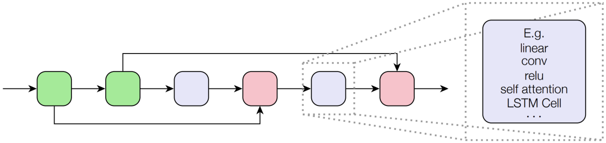
Neural Network Layers

THE UNIVERSITY OF BRITISH COLUMBIA

What is a “layer”?

A layer, for the purposes of this tutorial, is a ***differentiable parametric function***

Deep learning architectures are typically constructed by composing together many such layers, then training the complete system end-to-end via backpropagation



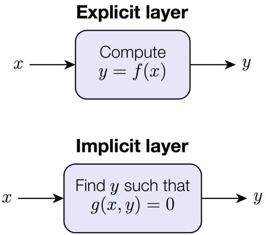
4

Explicit vs. Implicit layers

Virtually all commonly-used layers are ***explicit***, in that they provide a computation graph for computing the forward pass, and backprop through that computation graph

Implicit layers, in contrast, define a layer in terms of ***satisfying some joint condition of the input and output***

- *Many* examples: differential equations, fixed point iteration, optimization solutions, etc



Neural Network Layers

THE UNIVERSITY OF BRITISH COLUMBIA

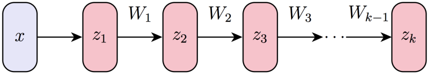
Why use implicit layers?

1. **Powerful representations:** compactly represent complex operations such as integrating differential equations, solving optimization problems, etc
2. **Memory efficiency:** no need to backpropagate through intermediate components, via implicit function theorem
3. **Simplicity:** Ease and elegance of designing architectures
4. **Abstraction:** Separate “what a layer should do” from “how to compute it”, an abstraction that has been extremely valuable in many other settings

6

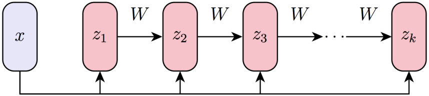
Motivating a simple implicit layer

Consider a traditional deep network applied to an input x



$$z_{i+1} = \sigma(W_i z_i + b_i)$$

We now modify this network in two ways: by re-injecting the input at each step, and by applying the *same* weight matrix at each iteration (weight tying)



$$z_{i+1} = \sigma(W z_i + x)$$

29

Iterations of deep weight-tied models

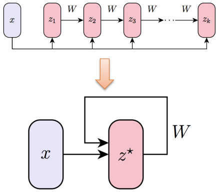
With a weight-tied model of this form, we are applying the *same* function repeatedly to the hidden units

$$z_{i+1} = \sigma(Wz_i + x)$$

In many situations, we can design the network such that this iteration will converge to some *fixed point*, or *equilibrium point*

$$z^\star = \sigma(Wz^\star + x)$$

This is precisely a recurrent backpropagation network, or a (minimal) deep equilibrium model



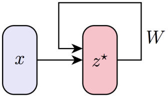
Simple instantiation: A tanh fixed point iteration

Let's consider a very simple form of such a fixed point layer, iterating:

$$z_{i+1} = \tanh(W z_i + x)$$

How do we compute the fixed point?

$$z^\star = \tanh(W z^\star + x)$$



How do we integrate such a layer with backprop? Does the derivative exist?

To answer this, let's see a quick demo

31

Auto-Diff in Implicit Layers

The implicit function theorem

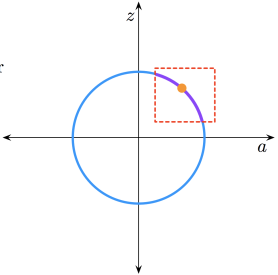
Let $f : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^n$ and $a_0 \in \mathbb{R}^p,\ z_0 \in \mathbb{R}^n$ be such that

1. $f(a_0, z_0) = 0$, and
2. f is continuously differentiable with non-singular Jacobian $\partial_1 f(a_0, z_0) \in \mathbb{R}^{n \times n}$.

Then there exist open sets $S_{a_0} \subset \mathbb{R}^p$ and $S_{z_0} \subset \mathbb{R}^n$ containing a_0 and z_0, respectively, and a unique continuous function $z^* : S_{a_0} \to S_{z_0}$ such that

1. $z_0 = z^*(a_0)$,
2. $f(a, z^*(a)) = 0 \quad \forall a \in S_{a_0}$, and
3. z^* is differentiable on S_{a_0}.

$f(a, z) = a^2 + z^2 - 1 = 0$



40

Auto-Diff in Implicit Layers

THE UNIVERSITY OF BRITISH COLUMBIA

The implicit function theorem

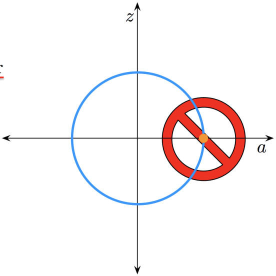
Let $f : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^n$ and $a_0 \in \mathbb{R}^p,\ z_0 \in \mathbb{R}^n$ be such that

1. $f(a_0, z_0) = 0$, and
2. f is continuously differentiable with non-singular Jacobian $\partial_1 f(a_0, z_0) \in \mathbb{R}^{n \times n}$.

Then there exist open sets $S_{a_0} \subset \mathbb{R}^p$ and $S_{z_0} \subset \mathbb{R}^n$ containing a_0 and z_0, respectively, and a unique continuous function $z^* : S_{a_0} \to S_{z_0}$ such that

1. $z_0 = z^*(a_0)$,
2. $f(a, z^*(a)) = 0 \quad \forall a \in S_{a_0}$, and
3. z^* is differentiable on S_{a_0}.

$f(a, z) = a^2 + z^2 - 1 = 0$



41

The implicit function theorem: derivative expression

$$f(a, z^*(a)) = 0 \quad \forall a \in S_{a_0}$$

$$\partial_0 f(a, z^*(a)) + \partial_1 f(a, z^*(a)) \partial z^*(a) = 0 \quad \forall a \in S_{a_0}$$

$$\partial_0 f(a_0, z_0) + \partial_1 f(a_0, z_0) \partial z^*(a_0) = 0$$

$$\partial z^*(a_0) = -[\partial_1 f(a_0, z_0)]^{-1} \partial_0 f(a_0, z_0)$$

Punchline: can express Jacobian matrix of solution mapping z^* in terms of Jacobian matrices of f at solution point (a_0, z_0).

45

Differentiation of fixed point solution mappings

$$z_0 = f(z_0, a_0)$$

$$z^*(a) = f(z^*(a), a)$$

$$\partial z^*(a_0) = \partial_0 f(z_0, a_0)\partial z^*(a_0) + \partial_1 f(z_0, a_0)$$

$$\partial z^*(a_0) = [I - \partial_0 f(z_0, a_0)]^{-1}\partial_1 f(z_0, a_0)$$

47

Connecting to automatic differentiation

1. Jacobian-vector products: $v \mapsto \partial f(x)\, v$

   JVP / push-forward / forward-mode

   build Jacobian one column at a time

2. vector-Jacobian products: $w \mapsto w^\top \partial f(x)$

   VJP / pull-back / reverse-mode

   build Jacobian one row at a time

48

Auto-Diff in Fixed Point Equations

THE UNIVERSITY OF BRITISH COLUMBIA

VJPs for fixed point solution mappings

$$\partial z^*(a_0) = [I - \partial_0 f(z_0, a_0)]^{-1} \partial_1 f(z_0, a_0)$$

$$w^\top \partial z^*(a_0) = w^\top [I - \partial_0 f(z_0, a_0)]^{-1} \partial_1 f(z_0, a_0)$$

$$= \boxed{u^\top \partial_1 f(z_0, a_0)}$$

VJPs!

$$\text{where } u^\top = w^\top + \boxed{u^\top \partial_0 f(z_0, a_0)}$$

Punchline: backward pass solve is a (linear) fixed point in terms of VJPs!

49

Auto-Diff in Fixed Point Equations

THE UNIVERSITY OF BRITISH COLUMBIA

Deep Equilibrium Models

The simple recurrent backpropagation cell we used previously was quite limited, in practice we want to find an equilibrium point of a more complex "cell", and use this as our *entire* model (plus one additional linear layer)

$$z^\star = \sigma(Wz^\star + x) \quad \Rightarrow \quad z^\star = f(z^\star, x, \theta)$$

Residual block, Transformer block, LSTM cell, etc ($\theta \equiv$ parameters of layers)

As motivated by the discussion on implicit differentiation, we additionally do not care *how* we solve for the equilibrium point, and can use any non-linear root finding algorithm to do so (and also to solve the backward pass)

[Bai, Kolter, Koltun "Deep Equilibrium Models", NeurIPS 2019]

51

How to train your DEQ

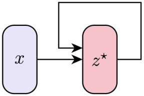
Forward pass:

- Given (x, y), compute equilibrium point $z^\star$
  $$z^\star = f(z^\star, x, \theta)$$
- Compute loss as some function of $z^\star$, $\ell(z^\star, y)$

Backward pass: Compute gradients using implicit function theorem:

$$\partial\ell(\theta) = \partial_0\ell(z^\star, y)\underbrace{\left(I - \partial_0 f(z^\star, x, \theta)\right)^{-1}\partial_2 f(z^\star, x, \theta)}$$

Implicit differentiation-based solution, solve via indirect method



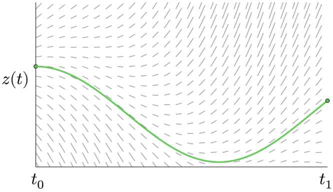
$$z^\star = f(z^\star, x, \theta)$$

52

Neural ODEs

THE UNIVERSITY OF BRITISH COLUMBIA

Ordinary Differential Equations



If a vector z follows dynamics f:

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

Can find $z(t_1)$ by starting at $z(t_0)$ and integrating until time t_1:

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

An implicit layer: $y = \mathrm{odeint}(f, x, t_0, t_1, \theta)$

For continuously differentiable and Lipshitz f, gradients always exist. (no `relu`, but `tanh` fine)

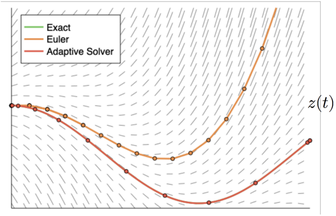
61

How to Solve ODEs?

Simplest way: Euler's method. Take steps of size h in direction of f

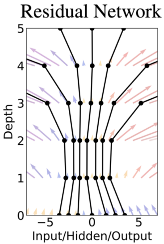
$$z_{i+1} = z_i + hf(z_i, t_i, \theta)$$

Looks just like a residual network!



From ResNets to ODE-Nets

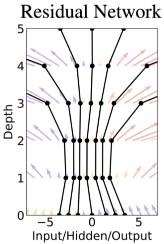
```
def f(z, t, θ):
  return nnet(z, θ[t])

def resnet(z, θ):
  for t in [1:T]:
    z = z + f(z, t, θ)
  return z
```



Neural ODEs Vs ResNets

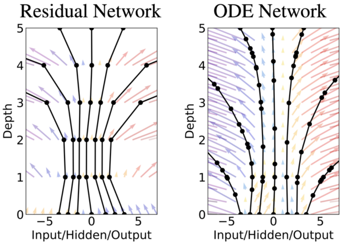
From ResNets to ODE-Nets

```
def f(z, t, θ):
  return nnet([z, t], θ)

def resnet(z, θ):
  for t in [1:T]:
    z = z + f(z, t, θ)
  return z
```



From ResNets to ODE-Nets

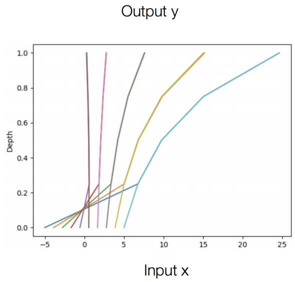
```
def f(z, t, θ):
  return nnet([z, t], θ)

def ODEnet(z, θ):
  return ODESolve(f, z, 0, 1, θ)
```



Residual Networks vs ODE solutions

Example: Fit $y = x^2$

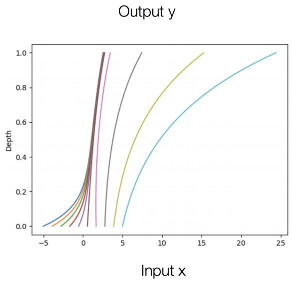
ResNet can learn non-bijective transformations.



Residual Networks vs ODE solutions

Example: Fit $y = x^2$

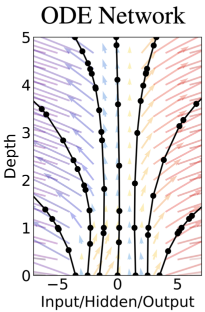
Ode-net can only learn bijective transformations.



Adaptive ODE Solvers

Adaptive solvers:

- Usually fit a local polynomial to dynamics
- Try to estimate extrapolation error
- Need fewer evaluations of dynamics function f when dynamics are simple / well-approximated
- Can adjust tolerance / precision of solver at any time

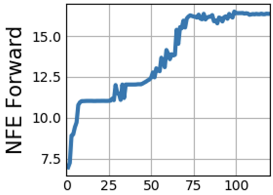


Dynamics Become Increasingly Complex in Training

Dynamics become more demanding to compute during training.

Adapts computation time according to complexity of dynamics.

Also happens in DEQs



(d) Training Epoch

Training Neural ODEs

How to train an ODE net?

Can backprop through solver operations, but high memory cost.

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

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

Continuous-time Backpropagation

Standard Backprop:

$$\frac{\partial L}{\partial \mathbf{z}_t} = \frac{\partial L}{\partial \mathbf{z}_{t+1}} \frac{\partial f(\mathbf{z}_t, \theta)}{\partial \mathbf{z}_t}$$

$$\frac{\partial L}{\partial \theta} = \sum_t \frac{\partial L}{\partial \mathbf{z}_t} \frac{\partial f(\mathbf{z}_t, \theta)}{\partial \theta}$$

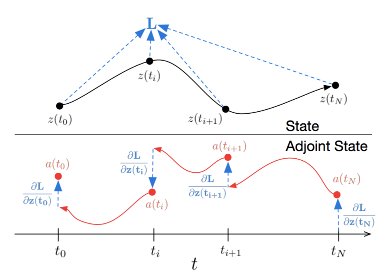
Adjoint sensitivities:
(Pontryagin et al., 1962):

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

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

The Adjoint Sensitivity Method

$$L(\mathbf{z}(t_1)) = L\left(\mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)dt\right) = L\left(\text{ODESolve}(\mathbf{z}(t_0), f, t_0, t_1, \theta)\right) \quad (3)$$



$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^{\mathsf{T}} \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}} \quad (4)$$

$$\frac{dL}{d\theta} = -\int_{t_1}^{t_0} \mathbf{a}(t)^{\mathsf{T}} \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \theta} dt \quad (5)$$

Where $a(t) = \partial L / \partial z(t)$

The Adjoint Sensitivity Method

THE UNIVERSITY OF BRITISH COLUMBIA

Continuous-time Backpropagation

Can build adjoint dynamics with autodiff, compute all gradients with another ODE solve:

```
def f_and_a([z, a, d], t):
  return [f, -a*df/da, -a*df/dθ)

[z0, dL/dx, dL/dθ] =
   ODESolve(f_and_a,
[z(t1), dL/dz(t), 0], t1, t0)
```

Adjoint sensitivities: (Pontryagin et al., 1962):

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

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

The Adjoint Sensitivity Method

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

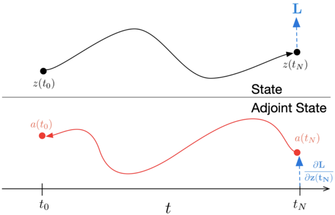
return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}$ ▷ Return gradients

O(1) Memory Gradients

No need to store activations, just run dynamics backwards from output.

Can do similar trick with Reversible ResNets (Gomez et al., 2018), but must restrict architecture.

This introduces extra numerical error. if mismatch is detected, can use checkpointing to force a better match.



ODE-Nets Performance

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

Figure 9: Data-space trajectories decoded from varying one dimension of $\mathbf{z}_{t_0}$. Color indicates progression through time, starting at purple and ending at red. Note that the trajectories on the left are counter-clockwise, while the trajectories on the right are clockwise.

Continuous Time Normalizing Flows

Continuous(-time) Normalizing Flows

Change of variables theorem:

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

Determinant is $O(D^3)$ cost

Must design architectures to have structured Jacobian

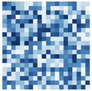
(Low rank) (Sparse) (Lower triangular)

Instantaneous change of variables:

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

Trace is always $O(D)$ cost.

Trace allows flows at **linear cost**.



(Arbitrary)

THE UNIVERSITY OF BRITISH COLUMBIA

Stochastic Estimation for CNFs

Divergence of a neural network can be be computationally expensive

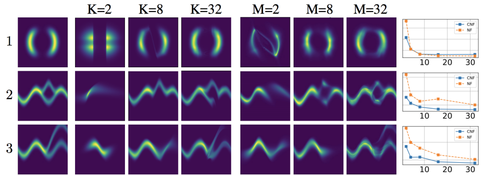
$$\begin{aligned}\log p(x) &= \log p(z) + \int_0^T \operatorname{div} f \; dt \\ &= \log p(z) + \int_0^T \operatorname{tr}(J_f) \; dt \\ &= \log p(z) + \mathbb{E}_{v\sim\mathcal{N}(0,1)} \left[\int_0^T v^T J_f v \; dt \right]\end{aligned}$$

vector-Jacobian products are cheap

$$\operatorname{tr}(A) = E_{v\sim\mathcal{N}(0,1)}[v^T A v]$$

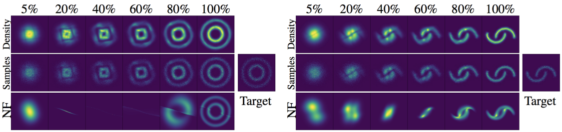
(Hutchinson's trace estimator)

Continuous Time Normalizing Flows Results



(a) Target (b) NF (c) CNF (d) Loss vs. K/M

Figure 4: Comparison of normalizing flows versus continuous normalizing flows. The model capacity of normalizing flows is determined by their depth (K), while continuous normalizing flows can also increase capacity by increasing width (M), making them easier to train.



(a) Two Circles (b) Two Moons

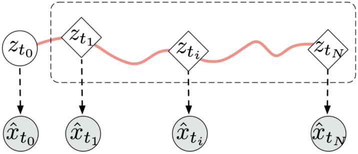
Figure 5: **Visualizing the transformation from noise to data.** Continuous-time normalizing flows are reversible, so we can train on a density estimation task and still be able to sample from the learned density efficiently.

Continuous-time Time-series Model

Continuous-time Time Series Models

Can deal with data collected at irregular intervals natively.

Can jointly train dynamics, likelihood, and recognition network as a VAE.



Latent ODEs for Irregularly-Sampled Time Series. Rubanova, Chen, Duvenaud (2020)
Neural Controlled Differential Equations for Irregular Time Series. Kidger, Morrill, Foster, Lyons (2020)
GRU-ODE-Bayes: Continuous modeling of sporadically-observed time series. de Brouwer, Simm, Arany, Moreau. (2020)

Continuous-time Time-series Model

$$\mathbf{z}_{t_0} \sim p(\mathbf{z}_{t_0}) \tag{11}$$

$$\mathbf{z}_{t_1}, \mathbf{z}_{t_2}, \ldots, \mathbf{z}_{t_N} = \mathrm{ODESolve}(\mathbf{z}_{t_0}, f, \theta_f, t_0, \ldots, t_N) \tag{12}$$

$$\text{each} \quad \mathbf{x}_{t_i} \sim p(\mathbf{x}|\mathbf{z}_{t_i}, \theta_{\mathbf{x}}) \tag{13}$$

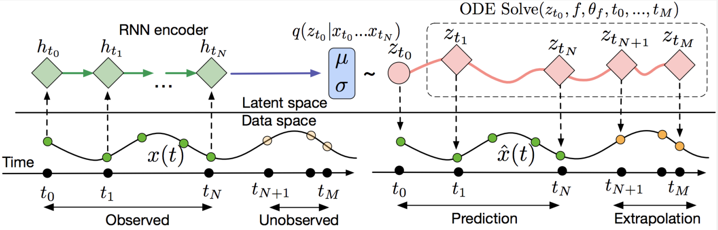
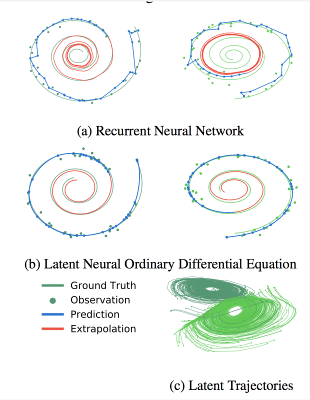


Figure 6: Computation graph of the latent ODE model.

Continuous-time Time-series Model Results

THE UNIVERSITY OF BRITISH COLUMBIA



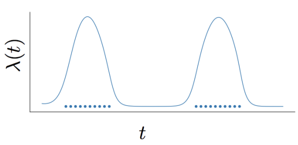


Figure 7: Fitting a latent ODE dynamics model with a Poisson process likelihood. Dots show event times. The line is the learned intensity $\lambda(t)$ of the Poisson process.

Conclusions and Future Work

Open Problems and Future Directions

1. Regularizing DEQs and Neural ODEs to be faster to solve
2. Re-architecting models to take advantage of memory advantages
3. Scaling and application of latent SDEs
4. Partial differential equation (PDE) solutions as a layer

106

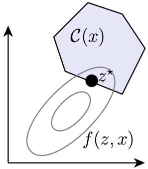
Differentiable optimization

DEQs and Neural ODEs both impose substantial structure on the nature of the layer, in order to gain substantial representational power

Other common strategy for imposing a different (but related) kind of structure is that of differentiable optimization

Layer of the form

$$z^\star = \operatorname*{argmin}_{z \in \mathcal{C}(x)} f(z, x)$$



99

Extra: Differentiable Optimization

THE UNIVERSITY OF BRITISH COLUMBIA

Differentiating optimization problems

How do we differentiate through a layer?

$$z^\star = \operatorname*{argmin}_{z \in \mathcal{C}(x)} f(z, x)$$

Finding a solution to constrained optimization is equivalent to finding the solution of a of a set of nonlinear equations called KKT conditions

$$\begin{aligned} z^\star = \quad & \operatorname{argmin} \ \frac{1}{2} z^T Q(x) z + p(x)^T z \\ & \text{subject to} \ A(x) z = b(x), \\ & \qquad\qquad\quad G(x) z \leq h(x) \end{aligned}$$

$$\begin{aligned} & \mathit{Find}\ (z^\star, \nu^\star, \lambda^\star)\ s.t. \\ & 1.\, Az^\star = b \\ & 2.\, Gz^\star \leq h \\ & 3.\, \lambda^\star \geq 0 \\ & 4.\, \lambda^\star \circ (Gz^\star - h) = 0 \\ & 5.\, Qz^\star + p + A^T \nu^\star + G^T \lambda^\star = 0 \end{aligned}$$

100

Extra: Differentiable Optimization

Differentiating through optimization problems

Alternatively, we can view virtually any optimization procedure as a fixed point iteration; e.g. for projected gradient descent

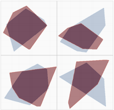
$$z_{k+1} = \mathrm{Proj}_{C(x)}[z_k - \alpha \partial_0 f(z_k, x)]$$

(But also true of much more sophisticated optimization approaches)

Therefore, can use differentiation of fixed point iteration to differentiate through optimization problems!

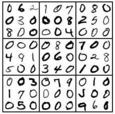
101

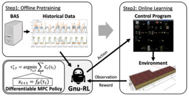
Some example applications



Learning a convex polytope from data
[Amos and Kolter., 2018]

Solving Sudoku (w/ MNIST digits) using
differentiable SDP solver [Wang et al., 2019]





Controlling HVAC systems with differentiable
MPC controllers [Chen et al., 2019]

102