Discretize-Optimize Methods for Neural ODEs in Continuous Normalizing Flows

SIAM MDS 2020

Derek Onken
Department of Computer Science
Emory University
donken@emory.edu
derekonken.com
Collaborators and Acknowledgments

Samy Wu Fung
UCLA

Xingjian Li
Emory

Lars Ruthotto
Emory

Emory Funding:  
- DMS 1751636
- US-Israel BSF 2018209

UCLA Funding:  
- AFOSR MURI FA9550-18-1-0502 and FA9550-18-1-0167
- ONR N00014-18-1-2527

Special thanks:  
Organizers and staff of IPAM Long Program MLP 2019.
Overview

- Continuous ResNet and Neural ODEs
  - Discrete neural networks viewed in continuous framework
- Continuous Normalizing Flows (CNFs)
  - Discrete normalizing flows similarly moved to the continuous framework
- Discretize-Optimize vs Optimize-Discretize
  - Comparing approaches in solving CNFs
- OT-Flow
  - Incorporating optimal transport with Discretize-Optimize for fast and accurate CNFs

Motivation: Existing CNFs approaches are prohibitively slow and expensive.

- DO and L Ruthotto
  Discretize-Optimize vs. Optimize-Discretize for Time-Series Regression and CNFs
- DO, S Wu Fung, X Li, L Ruthotto
  OT-Flow: Fast and Accurate CNFs via OT
Neural ODE

A neural ODE is an ordinary differential equation (ODE) with neural network components.

For input $x \in \mathbb{R}^d$, neural network $f: \mathbb{R}^d \to \mathbb{R}^d$ models the solution to

$$\partial_t z(x, t) = v(z(x, t), t; \theta), \quad z(x, 0) = x$$

where

- time $t \in [0, T]$
- $v: \mathbb{R}^d \times [0, T] \to \mathbb{R}^d$ is a neural network layer with parameters $\theta$
- $z(x, t)$ are the features for initial $x$ at time $t$
- $f(x) = z(x, T)$
Background

Historically

- Residual Neural Networks (ResNets)\(^1\) perform well on image classification and more.
- ResNets are merely Forward Euler of some Continuous ResNet (1).\(^2,3\)

---

\(^3\)Haber and Ruthotto. “Stable Architectures for Deep Neural Networks”. 2017.
We introduce a new family of deep neural network models. Instead of specifying a discrete sequence of hidden layers, we parameterize the derivative of the hidden state using a neural network. The output of the network is computed using a black-box differential equation solver. These continuous-depth models have constant memory cost, adapt their evaluation strategy to each input, and can explicitly trade numerical precision for speed. We demonstrate these properties in continuous-depth residual networks and continuous-time latent variable models. We also construct continuous normalizing flows, a generative model that can train by maximum likelihood, without partitioning or ordering the data dimensions. For training, we show how to scalably backpropagate through any ODE solver, without access to its internal operations. This allows end-to-end training of ODEs within larger models.

1 Introduction

Residual Network ODE Network

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.

Models such as residual networks, recurrent neural network decoders, and normalizing flows build complicated transformations by composing a sequence of transformations to a hidden state:

\[
\begin{align*}
    h_{t+1} &= h_t + f(h_t, \theta_t) \\
    t &\in \{0, ..., T\}
\end{align*}
\]  

where \( t \in \{0, ..., T\} \) and \( h_t \in \mathbb{R}^D \). These iterative updates can be seen as an Euler discretization of a continuous transformation (Lu et al., 2017; Haber and Ruthotto, 2017; Ruthotto and Haber, 2018).

What happens as we add more layers and take smaller steps? In the limit, we parameterize the continuous dynamics of hidden units using an ordinary differential equation (ODE) specified by a neural network:

\[
\frac{dh(t)}{dt} = f(h(t), t, \theta)
\]

Starting from the input layer \( h^{(0)} \), we can define the output layer \( h^{(T)} \) to be the solution to this ODE initial value problem at some time \( T \). This value can be computed by a black-box differential equation solver, which evaluates the hidden unit dynamics \( f \) wherever necessary to determine the solution with the desired accuracy. Figure 1 contrasts these two approaches.

Defining and evaluating models using ODE solvers has several benefits:

- Memory efficiency

In Section 2, we show how 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. Not storing any intermediate quantities of the forward pass allows us to train our models with nearly constant memory cost as a function of depth, a major bottleneck of training deep models.

Popularized

- Incorporate a black-box solver and coin the term neural ODE.\(^4\)
- Applied to normalizing flows.\(^5\)

Discrete Normalizing Flows

A normalizing flow\textsuperscript{6,7} is an invertible mapping $f : \mathbb{R}^d \to \mathbb{R}^d$ between an arbitrary probability distribution and a standard normal distribution whose densities we denote by $\rho_0$ and $\rho_1$, respectively.

By the change of variables formula, the flow satisfies

$$\log \rho_0(x) = \log \rho_1(f(x)) + \log |\det \nabla f(x)| \quad \text{for all} \quad x \in \mathbb{R}^d. \quad (2)$$

\textsuperscript{6}Rezende and Mohamed. “Variational Inference with Normalizing Flows”. 2015.
Gaussian Mixture Toy Example

Data $\mathbf{x}$

Estimate $\rho_0$

Generation $f^{-1}(\mathbf{y})$
Continuous Normalizing Flows (CNFs)

Replace the log-det with a trace

**Issue:**
- log-determinants cost $O(d^3)$ FLOPS in general.

**Solutions:**
- Use specific neural network architectures for $v$ so the log-det computation is manageable.
- Replace the log-det with a trace computation.

Using the neural ODE $f$ (1) and Jacobi’s formula\(^8\), we can rewrite (2) as

$$\ell(x, T) := \log \rho_0(x) - \log \rho_1(f(x)) = \int_0^T \text{tr} \left( \nabla v(z(x, t), t; \theta) \right) dt. \quad (3)$$

\(^8\)Chen et al. 2018.
For expected negative log-likelihood \(^9,^{10}\)

\[
C(x, T) := \frac{1}{2} \| z(x, T) \|^2 - \ell(x, T) + \frac{d}{2} \log(2\pi),
\]

we optimize

\[
\min_{\theta} \mathbb{E}_{\rho_0(x)} C(x, T)
\]

where for a given \(\theta\), the trajectory \(z\) satisfies the CNF \(^{11}\)

\[
\begin{bmatrix}
\partial_t z(x, t) \\
\partial_t \ell(x, t)
\end{bmatrix}
= \begin{bmatrix}
v(z(x, t), t; \theta) \\
\text{tr} \left( \nabla v(z(x, t), t; \theta) \right)
\end{bmatrix},
\]

\[
\begin{bmatrix}
z(x, 0) \\
\ell(x, 0)
\end{bmatrix}
= \begin{bmatrix}
x \\
0
\end{bmatrix}.
\]

In optimal control, \(z\) is the state and \(\theta\) is the control.

\(^9\)Rezende and Mohamed. 2015.
\(^{10}\)Papamakarios et al. 2019.
\(^{11}\)Grathwohl et al. 2019.
Solving ODE-constrained Optimization Problems

**Two Predominant Approaches:**

- **Discretize-Optimize (DO)**
  - Discretize the ODE, then optimize on that discretization.
  - Typical machine learning approach: set up architecture with \( N \) layers, then optimize on that discretization (propagate forward, calculate loss, backpropagate)
  - ANODE\(^{12}\)

- **Optimize-Discretize (OD)**
  - Optimize in the continuous space, then discretize.
  - Use the Karush-Kuhn-Tucker (KKT) conditions or the adjoint equations to optimize, then choose a discretization.
  - Neural ODEs paper\(^{13}\) and FFJORD\(^{14}\)

---


\(^{13}\)Chen et al. 2018.

\(^{14}\)Grathwohl et al. 2019.
Solving ODE-constrained Optimization Problems

Popular methods use Optimize-Discretize.

- Adaptive solver `dopri5` for the forward propagation
- Adjoint-based backpropagation recomputes the intermediate gradients
- **Drawback:** inaccurate gradients when the adjoint equation is not solved well enough.\(^\text{15,16}\)

We choose Discretize-Optimize.

- Same discretization for the forward and backpropagation.
  - Use automatic differentiation (AD) for the backpropagation.
  - The gradients are accurate.
- We use Runge-Kutta 4 with a fixed step size.
- **Drawback:** have to tune a sufficiently small step size for the solver

\(^{15}\) Li et al. “Maximum principle based algorithms for deep learning”. 2017.
\(^{16}\) Gholaminejad, Keutzer, and Biros. 2019.
For all five data sets, DO and FFJORD\textsuperscript{17} (OD) achieve similar results with different training time.

**DO has an average speed-up of 6.4x** even with slower training on HEPMASS.

**Reasons**

- Fewer function evaluations (RK4 instead of dopri5).
- Intermediate gradients are stored (with AD) rather than recomputed.
- DO has more accurate gradients.

\textsuperscript{17}Grathwohl et al. 2019.

\textit{DO and L Ruthotto}

\textit{Discretize-Optimize vs. Optimize-Discretize for Time-Series Regression and CNFs}

Can we solve even faster?

\[
\begin{align*}
  \partial_t \left[ \begin{array}{c} z(x, t) \\ \ell(x, t) \end{array} \right] &= \left[ \begin{array}{c} v(z(x, t), t; \theta) \\ \text{tr} \left( \nabla v(z(x, t), t; \theta) \right) \end{array} \right], \\
  \left[ \begin{array}{c} z(x, 0) \\ \ell(x, 0) \end{array} \right] &= \left[ \begin{array}{c} x \\ 0 \end{array} \right]
\end{align*}
\]

What makes CNFs slow?

- Trajectories can be complicated leading to high number of function evaluations.
- Trace computation costs \( O(d^2) \) FLOPS in general.
  - FFJORD uses Hutchinson’s estimator for \( O(d) \) FLOPS in training.
Straight Trajectories

Include some optimal transport (OT)

In OT, a unique mapping exists.

We regularize the optimization problem

$$\min_{\theta} \mathbb{E}_{\rho_0(x)} \left\{ C(x, T) + L(x, T) \right\}$$ (4)

subject to (1).

The $L_2$ transport costs are given by

$$L(x, T) = \int_0^T \frac{1}{2} \| v(z(x, t), t; \theta) \|^2 dt.$$
Apply the Pontryagin maximum principle\textsuperscript{18} to (4)

There exists a scalar potential function

$$\Phi: \mathbb{R}^d \times [0, T] \rightarrow \mathbb{R}$$

such that

$$v(x, t; \theta) = -\nabla \Phi(x, t; \theta).$$

(5)

Analogous to classical physics, samples move in a manner to minimize their potential.

We parametrize potential $\Phi$ instead of $v$.

\textsuperscript{18}Evans. \textit{An Introduction to Mathematical Optimal Control Theory Version 0.2.} 2013.
The optimality conditions of (4) lead to another regularizer.

Potential $\Phi$ satisfies the Hamilton-Jacobi-Bellman (HJB) equation

$$-\partial_t \Phi(x, t) + \frac{1}{2} \| \nabla \Phi(z(x, t), t) \|^2 = 0,$$

where

$$\Phi(x, T) = G(x)$$

Terminal condition $G$ derives from the variational derivative or the Kullback-Leibler (KL) divergence.

$\Phi_s$ satisfies the Hamilton-Jacobi-Bellman (HJB) equation

$$-\partial_t \Phi_s(x(t), t) + \frac{1}{2} \| \nabla \Phi_s(z(x(t), t), t) \|^2 = 0,$$

where

$$\Phi_s(x, T) = G(x)$$

Terminal condition $G$ derives from the variational derivative or the Kullback-Leibler (KL) divergence.

---

More OT

HJB regularizer $R$

Penalize deviations from the HJB equation

We add another regularizer, so the optimization problem is

$$\min_{\theta} \mathbb{E}_{\rho_0(x)} \left\{ C(x, T) + L(x, T) + R(x, T) \right\}$$

subject to (1).

The HJB regularizer is computed as

$$R(x, T) = \int_0^T \left| \partial_t \Phi(z(x, t), t) - \frac{1}{2} \| \nabla \Phi(z(x, t), t) \|^2 \right| \, dt.$$
OT-Flow Formulation

We incorporate the time integration in the ODE solver.

\[
\min_{\theta} \mathbb{E}_{\rho_0(x)} \left\{ C(x, T) + L(x, T) + R(x, T) \right\}
\]

subject to

\[
\partial_t \begin{pmatrix} z(x, t) \\ \ell(x, t) \\ L(x, t) \\ R(x, t) \end{pmatrix} = \begin{pmatrix} -\nabla \Phi(z(x, t), t; \theta) \\ -\text{tr}(\nabla^2 \Phi(z(x, t), t; \theta)) \\ \frac{1}{2} \| \nabla \Phi(z(x, t), t; \theta) \|^2 \\ |\partial_t \Phi(z(x, t), t; \theta) - \frac{1}{2} \| \nabla \Phi(z(x, t), t; \theta) \|^2| \end{pmatrix}
\]

with initial conditions

\[
z(x, 0) = x \quad \text{and} \quad \ell(x, 0) = L(x, 0) = R(x, 0) = 0
\]
Other OT approaches in CNFs

### Table: Comparison of flow formulations.

<table>
<thead>
<tr>
<th>Model</th>
<th>ODE (1)</th>
<th>$\Phi$</th>
<th>$L_2$ cost</th>
<th>HJB reg.</th>
<th>$|\nabla v|^2_F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFJORD$^{20}$</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>RNODE$^{21}$</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Monge-Ampère Flows$^{22}$</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Potential Flow Gen.$^{23}$</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OT-Flow</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

$^{20}$ Grathwohl et al. 2019.
Improving the Trace Computation

**General Trace Computation:** $O(d^2)$ FLOPS

**Trace Estimators used in state-of-the-art:** $O(d)$ FLOPS

**Our Exact Trace in OT-Flow** $O(d)$ FLOPS

In runtime, our exact trace is competitive with the estimators

(a) MINIBOONE, $d=43$

(b) BSDS300, $d=63$

(c) MNIST, $d=784$

(d) Accuracy of Estimators
**Exact Trace Computation**

Our model

**Neural Network**

\[
\Phi(s; \theta) = w^\top N(s; \theta_N) + \frac{1}{2} s^\top (A^\top A) s + b^\top s + c,
\]

where \( \theta = (w, \theta_N, A, b, c) \)

**Gradient**

\[
\nabla_s \Phi(s; \theta) = \nabla_s N(s; \theta_N) w + (A^\top A) s + b
\]

where

- space-time inputs \( s = (x, t) \in \mathbb{R}^{d+1} \)
- neural network \( N(s; \theta_N) : \mathbb{R}^{d+1} \rightarrow \mathbb{R}^m \) (we choose ResNet)
- \( \theta \) consists of all the trainable weights:
  \( w \in \mathbb{R}^m, \theta_N \in \mathbb{R}^p, A \in \mathbb{R}^{r \times (d+1)}, b \in \mathbb{R}^{d+1}, c \in \mathbb{R} \)
  where \( r = \min(10, d) \)
$N$ is an $(M + 1)$-layer ResNet

**Forward propagation**
Compute $N(s; \theta_N) = u_M$.

\[
\begin{align*}
    u_0 &= \sigma(K_0 s + b_0) \\
    u_1 &= u_0 + h\sigma(K_1 u_0 + b_1) \\
    &\vdots \\
    u_M &= u_{M-1} + h\sigma(K_M u_{M-1} + b_M)
\end{align*}
\]

where

- fixed step size $h > 0$
- ResNet weights $\theta_N$ are
  \> $K_0 \in \mathbb{R}^{m \times (d+1)}$
  \> $K_1, \ldots, K_M \in \mathbb{R}^{m \times m}$
  \> $b_0, \ldots, b_M \in \mathbb{R}^m$
- $\sigma(x) = \log(\exp(x) + \exp(-x))$
  \> the antiderivative of hyperbolic tangent
  \> so, $\sigma'(x) = \tanh(x)$
Exact Trace Computation
Analytic Gradient Computation

$N$ is an $(M + 1)$-layer ResNet

**Forward propagation**
Compute $N(s; \theta_N) = u_M$.

$u_0 = \sigma(K_0s + b_0)$
$u_1 = u_0 + h\sigma(K_1u_0 + b_1)$
  ...
$u_M = u_{M-1} + h\sigma(K_Mu_{M-1} + b_M)$

**Backpropagation**
Compute $\nabla_s N(s; \theta_N)w = z_0$

$z_{M+1} = w$
$z_M = z_{M+1}$
  $+ hK_M^\top \text{diag}(\sigma'(K_Mu_{M-1} + b_M))z_{M+1}$
$z_1 = z_2 + hK_1^\top \text{diag}(\sigma'(K_1u_0 + b_1))z_2$
$z_0 = K_0^\top \text{diag}(\sigma'(K_0s + b_0))z_1$
Exact Trace Computation

Laplacian of the Potential

\[
\text{tr} \left( \nabla^2 \Phi(s; \theta) \right) = \text{tr} \left( E^\top \left( \nabla_s^2 (N(s; \theta_N)w) + A^\top A \right) E \right) \quad \text{for} \quad E = \text{eye}(d+1, d)
\]
Exact Trace Computation

Laplacian of the Potential

\[ \text{tr} \left( \nabla^2 \Phi(s; \theta) \right) = \text{tr} \left( E^\top (\nabla_s^2 N(s; \theta_N) \omega) + A^\top A \right) E \] for \( E = \text{eye}(d+1, d) \)

Focus on the nontrivial part (the ResNet)

\[ \text{tr} \left( E^\top \nabla_s^2 (N(s; \theta_N) \omega) E \right) = t_0 + h \sum_{i=1}^{M} t_i, \]
Exact Trace Computation
Laplacian of the Potential

$$\text{tr} \left( \nabla^2 \Phi(s; \theta) \right) = \text{tr} \left( E^\top \left( \nabla^2_s (N(s; \theta_N)w) + A^\top A \right) E \right) \quad \text{for} \quad E = \text{eye}(d+1, d)$$

Focus on the nontrivial part (the ResNet)

$$\text{tr} \left( E^\top \nabla^2_s (N(s; \theta_N)w) E \right) = t_0 + h \sum_{i=1}^{M} t_i,$$

With one pass, we calculate the trace of each layer

$$t_0 = \text{tr} \left( J_{i-1}^\top \nabla_s (K_i^\top \text{diag}(\sigma''(K_i u_{i-1}(s) + b_i)) z_{i+1}) J_{i-1} \right)$$

$$= (\sigma''(K_0 s + b_0) \odot z_1)^\top ((K_0 E) \odot (K_0 E)) \mathbf{1} \quad \text{and}$$

$$t_i = \text{tr} \left( J_{i-1}^\top \nabla_s (K_i^\top \text{diag}(\sigma''(K_i u_{i-1}(s) + b_i)) z_{i+1}) J_{i-1} \right)$$

$$= (\sigma''(K_i u_{i-1} + b_i) \odot z_{i+1})^\top ((K_i J_{i-1}) \odot (K_i J_{i-1})) \mathbf{1}.$$

where Jacobian $J_{i-1} = \nabla_s u_{i-1}^\top \in \mathbb{R}^{m \times d}$, $\odot$ denotes Hadamard product, and $\mathbf{1} = \text{ones}(d, 1)$.
Exact Trace Computation

Efficiency

Update and overwrite $J_{i-1} = \nabla_s u_{i-1}^\top \in \mathbb{R}^{m \times d}$ in the forward pass as

$$\nabla_s u_i^\top = \nabla_s u_{i-1} + h \sigma'(K_i u_{i-1} + b_i) K_i^\top \nabla_s u_{i-1}$$

$$J \leftarrow J + h \sigma'(K_i u_{i-1} + b_i) K_i^\top J$$

Overall Cost is $\mathcal{O}(m^2 \cdot d)$ FLOPS.

Recall: $K_0 \in \mathbb{R}^{m \times (d+1)}$ and $K_1, \ldots, K_M \in \mathbb{R}^{m \times m}$
Results

Fast Training

OT-Flow has 19x training speed-up on average

Reasons

- OT-inspired regularization leads to straight trajectories that are inexpensive to integrate.
- The trace computation is efficient and exact.
- The potential flows approach results in fewer weights and a smaller model.

DO, S Wu Fung, X Li, L Ruthotto
OT-Flow: Fast and Accurate CNFs via OT
Results

Fast Inference

OT-Flow has 28x testing speed-up on average

Inference-Specific Reasons

- Inference uses exact trace (no estimates)
  - State-of-the-art approaches use AD to obtain exact trace with $O(d^2)$
  - Meanwhile, our exact trace is $O(d)$

DO, S Wu Fung, X Li, L Ruthotto

*OT-Flow: Fast and Accurate CNFs via OT*

More Results
Details in paper

Samples
\( x \sim \rho_0(x) \)

OT-Flow
\( f(x) \)

FFJORD
\( f(x) \)

Two of the 43 dimensions in the MINIBOONE CNF.

MNIST synthetic generation. Original images boxed in red.

DO, S Wu Fung, X Li, L Ruthotto
OT-Flow: Fast and Accurate CNFs via OT
Conclusions

**Discretize-Optimize**
- DO often converges faster than OD when used in neural ODEs
- For CNFs, DO provides 6x training speedup

**OT-Flow**
- OT regularization \(\Rightarrow\) well-posed and efficient time integration
- Potential flow \(\Rightarrow\) smaller model
- OT-Flow achieves 19x training speedup and 28x inference speedup over same baseline

---

DO and L Ruthotto
*DO vs. OD for Time-Series Regression and CNFs*

DO, S Wu Fung, X Li, L Ruthotto
*OT-Flow: Fast and Accurate CNFs via OT*

Code: [github.com/EmoryMLIP/OT-Flow](https://github.com/EmoryMLIP/OT-Flow)


