Higher-Order Optimization of Neural ODEs via Optimal Control Principle

Guan-Horng Liu

Center of Machine Learning
Georgia Institute of Technology
ghliu@gatech.edu

12/13 DataSig | Rough Path Interest Group
Second-Order Neural ODE Optimizer \textbf{(NeurIPS'21 spotlight)}

A new optimizer for deep continuous-time models (e.g., Neural ODEs) that enjoys

- Strong empirical results
  - superior convergence & test-time performance
  - hyperparameter robustness
  - architecture optimization

- Solid theoretical analysis
  - continuous-time optimal control theory
  - generalization of first-order adjoint method to higher-order at the \textit{same} $O(1)$ memory (in depth)
Training Process of Neural ODEs

Adjoint-based optimization

\[
\min_{\theta} \mathcal{L}(x(t_1)), \quad \text{s.t.} \quad \frac{dx(t)}{dt} = F(t, x(t), \theta), \quad x(t_0) = x_{t_0}
\]

\textbf{Forward ODE} \quad \textbf{Backward Adjoint ODE}

\[
\begin{align*}
&x(t_0) \quad \xrightarrow{\text{ODESolve}} \quad x(t_1) \\
&F(\cdot, \cdot, \theta)
\end{align*}
\]

\[
\begin{align*}
&x(t_0) \quad \xrightarrow{\text{ODESolve}} \quad x(t_1) \\
&\frac{\partial \mathcal{L}}{\partial x(t_0)} \quad \frac{\partial \mathcal{L}}{\partial \theta}
\end{align*}
\]

\text{Backward Vector Field}

\text{Computation flow}

\text{Query time derivative}
Training Process of Neural ODEs

Application to higher-order optimization (prior attempts)

Backward Adjoint ODE

\[
\begin{bmatrix}
    x(t_0) \\
    \frac{\partial L}{\partial x(t_0)} \\
    \frac{\partial L}{\partial \theta}
\end{bmatrix} \xleftarrow{\text{ODESolve}} \begin{bmatrix}
    x(t_1) \\
    \frac{\partial L}{\partial x(t_1)} \\
    0
\end{bmatrix}
\]

\[\frac{\partial}{\partial \theta} L = \text{grad}(L, \theta)\]

→ Computation flow
← → Query time derivative
Training Process of Neural ODEs

Application to higher-order optimization (prior attempts)

\[ \frac{\partial^n L}{\partial x^n(t_0)} \]
\[ \frac{\partial^n L}{\partial x^n(t_1)} \]

\[ \frac{\partial^n L}{\partial \theta^n} = \text{grad}(\frac{\partial^{n-1} L}{\partial \theta^{n-1}}, \theta) \]

 spel intersection error

backward recursive adjoint ODE

Table: Numerical errors of recursive adjoint method

<table>
<thead>
<tr>
<th></th>
<th>rk4</th>
<th>implicit adams</th>
<th>dopri5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{\partial L}{\partial \theta} )</td>
<td>7.63 × 10^{-5}</td>
<td>2.11 × 10^{-3}</td>
<td>3.44 × 10^{-4}</td>
</tr>
<tr>
<td>( \frac{\partial^2 L}{\partial \theta^2} )</td>
<td>6.83 × 10^{-3}</td>
<td>2.50 × 10^{-1}</td>
<td>44.10</td>
</tr>
</tbody>
</table>
Training Process of Neural ODEs

Application to higher-order optimization (prior attempts)

**Backward Recursive Adjoint ODE**

\[
\begin{bmatrix}
x(t_0) \\
\frac{\partial^n L}{\partial x^n(t_0)} \\
\frac{\partial^n L}{\partial \theta^n}
\end{bmatrix}
\rightarrow
\text{ODESolve}
\rightarrow
\begin{bmatrix}
x(t_1) \\
\frac{\partial^n L}{\partial x^n(t_1)} \\
0
\end{bmatrix}
\]

\[
\frac{\partial^n L}{\partial \theta^n} = \text{grad}(\frac{\partial^{n-1} L}{\partial \theta^{n-1}}, \theta)
\]

- 😞 linear runtime dependency
- 😞 accumulated integration errors

### Higher-order computational framework from an optimization viewpoint.

This Talk
Optimal Control Perspective (OC)

- Treat the *propagation of each layer* as a distinct *time step of a nonlinear dynamical system*.
- Interpret *layer parameter* as the *time-varying control* (Weinan et al., 2018; Liu & Theodorou, 2019).
- New optimization theory & robustifying prior methods with OC optimality (Liu et al., 2021a,b).
Neural ODEs, by construction, aim to represent continuous-time dynamical systems. Backward adjoint ODE originates from optimality conditions in Optimal Control (Pontryagin et al., 1962).
Training Neural ODE by Solving OCP

**Original Training Process**

\[
\begin{align*}
\min_{\theta} & \quad \mathcal{L}(x(t_1)) , \\
\text{s.t.} & \quad \frac{dx(t)}{dt} = F(t, x(t), \theta), \quad x(t_0) = x_{t_0}, \quad \text{Neural ODE}
\end{align*}
\]

**Optimal Control Programming (OCP)**

\[
\begin{align*}
(\mathcal{L}, 0) & := (\Phi, \ell) \\
\min_{\theta} & \quad \left[ \Phi(x_{t_f}) + \int_{t_0}^{t_1} \ell(t, x_t, u_t) dt \right] , \\
\text{s.t.} & \quad \begin{cases}
\frac{dx(t)}{dt} = F(t, x(t), u(t)), \\
\frac{du(t)}{dt} = 0,
\end{cases} \\
& \quad x(t_0) = x_{t_0}, \\
& \quad u(t_0) = \theta, \\
& \quad \text{ODE with time-invariant control}
\end{align*}
\]

Define accumulated loss:

\[
Q(t, x_t, u_t) := \Phi(x_{t_1}) + \int_{t}^{t_1} \ell(\tau, x_\tau, u_\tau) d\tau
\]

**Adjoint-based derivatives**

\[
\frac{\partial \mathcal{L}}{\partial \theta}, \quad \frac{\partial^2 \mathcal{L}}{\partial \theta \partial \theta}
\]

\[
(\frac{\partial \mathcal{L}}{\partial \theta}, \frac{\partial^2 \mathcal{L}}{\partial \theta \partial \theta}) := (Q_u(t_0), Q_{uu}(t_0))
\]

**OCP-based derivatives**

\[
\begin{align*}
\frac{\partial Q(t_0, x_{t_0}, u_{t_0})}{\partial u_{t_0}} & = Q_u(t_0) \\
\frac{\partial^2 Q(t_0, x_{t_0}, u_{t_0})}{\partial u_{t_0} \partial u_{t_0}} & = Q_{uu}(t_0)
\end{align*}
\]
Generalization of Adjoint Process

Our goal is to solve derivatives of $Q(t, x_t, u_t)$ w.r.t. the control $u$ at $t_0$:

**Theorem 1 (Second-Order Differential Programming)**

The derivatives of $Q$ expanded along a solution path $(x_t, u_t)$, which solves the forward ODE, obey the following set of **coupled** backward ODEs:

\[
\begin{align*}
- \frac{dQ_x}{dt} &= \ell_x + F_x^T Q_x \\
- \frac{dQ_{xx}}{dt} &= \ell_{xx} + F_x^T Q_{xx} + Q_{xx} F_x \\
- \frac{dQ_{uu}}{dt} &= \ell_{uu} + F_u^T Q_{uu} + Q_{ux} F_u \\
- \frac{dQ_u}{dt} &= \ell_u + F_u^T Q_x \\
- \frac{dQ_{xu}}{dt} &= \ell_{xu} + F_x^T Q_{xu} + Q_{xx} F_u \\
- \frac{dQ_{ux}}{dt} &= \ell_{ux} + F_u^T Q_{xx} + Q_{ux} F_x
\end{align*}
\]

- no recursive dependency! (everything's solved in **one single backward pass**)
- can be extended to computing higher-order derivatives (e.g., 3rd-order tensors)
Derivation Roadmap

**Derivation**

- **1st-order adjoint method**
- Higher-order adjoint process (Theorem 1)

---

**Forward ODE**

\[ x(t_0) = \text{ODESolve}(t_0, t_1, x(t_0), F) \]

**Backward Adjoint ODE (O(1) memory)**

\[
\begin{bmatrix}
    x(t_0), 
    \frac{\partial L}{\partial x(t_0)}, 
    \frac{\partial L}{\partial \theta}
\end{bmatrix}^T
\]

\[ = \text{ODESolve}(t_1, t_0, [x(t_1), \frac{\partial L}{\partial x(t_1)}, 0]^T, G) \]
Derivation Roadmap

**Derivation**

- **1st-order adjoint method**

- **Higher-order adjoint process (Theorem 1)**

**Forward ODE**

\[
\begin{align*}
\mathbf{x}(t_0) &= \text{ODESolve}(t_0, t_1, \mathbf{x}(t_0), F) \\
\end{align*}
\]

**Backward Adjoint ODE (O(1) memory)**

\[
\begin{align*}
&\begin{bmatrix} x(t_0), \frac{\partial L}{\partial x(t_0)}, \frac{\partial L}{\partial \theta} \end{bmatrix}^T \\
&= \text{ODESolve}(t_1, t_0, [x(t_1), \frac{\partial L}{\partial x(t_1)}, 0]^T, \mathcal{G})
\end{align*}
\]

**Higher-Order Backward Adjoint ODE (O(1) memory)**

\[
\begin{align*}
&\begin{bmatrix} x(t_0), Q_x(t_0), Q_u(t_0), Q_{xx}(t_0), Q_{xxu}(t_0), Q_{uux}(t_0), Q_{uu}(t_0) \end{bmatrix}^T \\
&= \text{ODESolve}(t_1, t_0, [x(t_1), \Phi_x, 0, 0, 0, 0]^T, \mathcal{G})
\end{align*}
\]

![Coupled matrix ODEs are too expensive...](image)

(from Theorem 1)

**1st-order**

**2nd-order**
Efficient Higher-Order Computation: Step 1

- Disentangle the ***coupled matrix ODEs*** in Theorem 1 into a set of ***independent vector ODEs***.

**Proposition 2 (Low-rank representation)**

Suppose $\ell := 0$ and let $Q_{xx}(t_1) = \sum_{i=1}^{R} y_i \otimes y_i$ be a symmetric matrix of rank $R$. Then, we have the following decompositions:

\[
Q_{xx}(t) = \sum_{i=1}^{R} q_i(t) \otimes q_i(t), \quad Q_{xu}(t) = \sum_{i=1}^{R} q_i(t) \otimes p_i(t), \quad Q_{uu}(t) = \sum_{i=1}^{R} p_i(t) \otimes p_i(t)
\]

where the vectors $(q_i(t), p_i(t))$ obey the following backward ODEs:

\[
- \frac{dq_i(t)}{dt} = F_x^T q_i(t), \quad - \frac{dp_i(t)}{dt} = F_u^T p_i(t)
\]

with the terminal conditions given by $(q_i(t_1), p_i(t_1)) := (y_i, 0)$. 
Efficient Higher-Order Computation: Step 1

**Derivation**

1. **1st-order adjoint method**

2. **Higher-order adjoint process (Theorem 1)**

3. **Low-rank representation (Proposition 2)**

**Higher-Order Backward Adjoint ODE (matrix form)**

\[
\begin{bmatrix} x(t_0), Q_x(t_0), Q_u(t_0), \{Q_{xx}(t_0)\}_{i=1}^R, \{Q_{ux}(t_0)\}_{i=1}^R, Q_{uu}(t_0) \end{bmatrix}^T = \text{ODESolve}(t_1, t_0, [x(t_1), \Phi_x, 0, \Phi_{xx}, 0, 0]^T, \tilde{G})
\]

**Higher-Order Backward Adjoint ODE (low rank)**

\[
\begin{bmatrix} x(t_0), Q_x(t_0), Q_u(t_0), \{q_i(t_0)\}_{i=1}^R, \{p_i(t_0)\}_{i=1}^R \end{bmatrix}^T = \text{ODESolve}(t_1, t_0, [x(t_1), \Phi_x, 0, \{y\}_{i=1}^R, 0]^T, \tilde{G})
\]

\[\implies \text{recover precondition matrix: } \mathcal{L}_{\theta\theta} \equiv Q_{uu}(t_0) = \sum_{i=1}^{R} p_i(t_0) \otimes p_i(t_0)\]

\[\implies \text{compute preconditioned update: } \mathcal{L}_{\theta} \leftarrow \mathcal{L}_{\theta\theta}^{-1} \mathcal{L}_{\theta}\]

Efficient backward pass! Preconditioning is still quite expensive...
Efficient Higher-Order Computation: Step 2

Derivation

1\textsuperscript{st}-order adjoint method

Higher-order adjoint process (Theorem 1)

Low-rank representation (Proposition 2)

Kronecker-factored preconditioned update

\[
\mathcal{L}_{\theta\theta}^{-1} \mathcal{L}_\theta \approx \mathcal{L}_{\theta\theta k}^{-1} \mathcal{L}_{\theta k} = \text{vec}(B_k^{-1} \mathcal{L}_{\theta k} A_k^{-T})
\]

\[
\theta_1 \quad \theta_k \quad \vdots \quad \theta_l
\]
Efficient Higher-Order Computation: Step 2

Derivation

1st-order adjoint method

Higher-order adjoint process (Theorem 1)

Low-rank representation (Proposition 2)

Kronecker-factored preconditioned update

(i) expand the precondition matrix with the ODE in Proposition 2

\[ \mathcal{L}_{\theta\theta} = \sum_{i=1}^{R} \mathbf{p}_i(t_0) \otimes \mathbf{p}_i(t_0) = \sum_{i=1}^{R} \left( \int_{t_1}^{t_0} F_{\theta}^T \mathbf{q}_i dt \right) \otimes \left( \int_{t_1}^{t_0} F_{\theta}^T \mathbf{q}_i dt \right), \]
Efficient Higher-Order Computation: Step 2

1st-order adjoint method

Higher-order adjoint process (Theorem 1)

Low-rank representation (Proposition 2)

Kronecker-factored preconditioned update

(ii) layer-wise precondition matrix.

\[ \mathcal{L}_{\theta\theta} = \sum_{i=1}^{R} p_i(t_0) \otimes p_i(t_0) = \sum_{i=1}^{R} \left( \int_{t_0}^{t_1} F_{\theta}^T q_i dt \right) \otimes \left( \int_{t_1}^{t_0} F_{\theta}^T q_i dt \right) \]

\[ \mathcal{L}_{\theta_k\theta_k} = \sum_{i=1}^{R} \left( \int_{t_1}^{t_0} F_{\theta_k}^T q_i dt \right) \otimes \left( \int_{t_1}^{t_0} F_{\theta_k}^T q_i dt \right) \]
Efficient Higher-Order Computation: Step 2

Derivation

1\textsuperscript{st}-order adjoint method

Higher-order adjoint process (Theorem 1)

Low-rank representation (Proposition 2)

Kronecker-factored preconditioned update

(iii) exploit the Kronecker factorization of JVP.

\[
\mathcal{L}_{\theta \theta} = \sum_{i=1}^{R} \mathbf{p}_i(t_0) \otimes \mathbf{p}_i(t_0) = \sum_{i=1}^{R} \left( \int_{t_1}^{t_0} F_{\theta}^T \mathbf{q}_i dt \right) \otimes \left( \int_{t_1}^{t_0} F_{\theta}^T \mathbf{q}_i dt \right),
\]

\[
\mathcal{L}_{\theta_k \theta_k} = \sum_{i=1}^{R} \left( \int_{t_1}^{t_0} F_{\theta_k}^T \mathbf{q}_i dt \right) \otimes \left( \int_{t_1}^{t_0} F_{\theta_k}^T \mathbf{q}_i dt \right)
\]

\[
= \sum_{i=1}^{R} \left( \int_{t_1}^{t_0} \left( \mathbf{z}_k \otimes \left( \frac{\partial F^T}{\partial \mathbf{h}_k} \mathbf{q}_i \right) \right) dt \right) \otimes \left( \int_{t_1}^{t_0} \left( \mathbf{z}_k \otimes \left( \frac{\partial F^T}{\partial \mathbf{h}_k} \mathbf{q}_i \right) \right) dt \right)
\]
Efficient Higher-Order Computation: Step 2

Derivation

1\textsuperscript{st}-order adjoint method

Higher-order adjoint process (Theorem 1)

Low-rank representation (Proposition 2)

Kronecker-factored preconditioned update

(iv) adopt formula of Kronecker product and few approximations.

\[
\mathcal{L}_{\theta \theta} = \sum_{i=1}^{R} p_i(t_0) \otimes p_i(t_0) = \sum_{i=1}^{R} \left( \int_{t_1}^{t_0} F_{\theta}^T q_i dt \right) \otimes \left( \int_{t_1}^{t_0} F_{\theta}^T q_i dt \right),
\]

\[
\mathcal{L}_{\theta_k \theta_k} = \sum_{i=1}^{R} \left( \int_{t_1}^{t_0} F_{\theta_k}^T q_i dt \right) \otimes \left( \int_{t_1}^{t_0} F_{\theta_k}^T q_i dt \right)
\]

\[
= \sum_{i=1}^{R} \left( \int_{t_1}^{t_0} \left( z_k \otimes \left( \frac{\partial F^T}{\partial h_k} q_i \right) \right) dt \right) \otimes \left( \int_{t_1}^{t_0} \left( z_k \otimes \left( \frac{\partial F^T}{\partial h_k} q_i \right) \right) dt \right)
\]

\[
\approx \int_{t_1}^{t_0} (z_k \otimes z_k) dt \otimes \int_{t_1}^{t_0} \sum_{i=1}^{R} \left( \frac{\partial F^T}{\partial h_k} q_i \otimes \frac{\partial F^T}{\partial h_k} q_i \right) dt
\]

\[
\underbrace{A_k(t)}_{A_k(t)} \otimes \underbrace{B_k(t)}_{B_k(t)}
\]

\[\Rightarrow\] layer-wise precondition matrix: \( \mathcal{L}_{\theta_k \theta_k} \approx \bar{A}_k \otimes \bar{B}_k, \) where \[
\begin{align*}
\bar{A}_k &= \sum_j A_k(t_j) \cdot \Delta t \\
\bar{B}_k &= \sum_j B_k(t_j) \cdot \Delta t
\end{align*}
\]
Efficient Higher-Order Computation: Step 2

Derivation

1\textsuperscript{st}-order adjoint method

Higher-order adjoint process (Theorem 1)

Low-rank representation (Proposition 2)

Kronecker-factored preconditioned update

**Higher-Order Adjoint ODE (low rank)**

\[
[x(t_0), Q_x(t_0), Q_u(t_0), \{q_i(t_0)\}_{i=1}^R, \{p_i(t_0)\}_{i=1}^R]^T = \text{ODESolve}(t_1, t_0, [x(t_1), \Phi_x, 0, \{y\}_{i=1}^R, 0]^T, \tilde{G})
\]

**Higher-Order Adjoint ODE (low rank + Kronecker)**

\[
[x(t_0), Q_x(t_0), Q_u(t_0), \{q_i(t_0)\}_{i=1}^R]^T = \text{ODESolve}(t_1, t_0, [x(t_1), \Phi_x, 0, \{y\}_{i=1}^R]^T, \tilde{G})
\]
Computation Comparison to 1st-Order Baseline

Derivation

1st-order adjoint method

Higher-order adjoint process (Theorem 1)

Low-rank representation (Proposition 2)

Second-Order Neural ODE Optimizer (SNOpt)

\[ \begin{bmatrix} x(t_0), \frac{\partial L}{\partial x(t_0)}, \frac{\partial L}{\partial \theta} \end{bmatrix}^T = \text{ODESolve} \left( t_1, t_0, [x(t_1), \frac{\partial L}{\partial x(t_1)}, 0]^T, G \right) \]

\[ \begin{bmatrix} x(t_1) \end{bmatrix} \]

ODE solution path

query time derivatives

collect sampled matrices

sampled time grid \{t_j\}

\[ \begin{bmatrix} x(t_0), Q_x(t_0), Q_u(t_0), \{q_i(t_0)\}_{i=1}^R \end{bmatrix}^T = \text{ODESolve} \left( t_1, t_0, [x(t_1), \Phi_x, 0, \{y\}_{i=1}^R]^T, \hat{G} \right) \]

\[ \begin{bmatrix} x(t_1) \end{bmatrix} \]

\[ \begin{bmatrix} Q_x(t_1) \\ Q_u(t_0) \end{bmatrix} \]

\[ \begin{bmatrix} \{y_i\}_{i=1}^R \end{bmatrix} \]

\[ \begin{align*}
\hat{A}_k &= \sum_j A_k(t_j) \cdot \Delta t \\
\hat{B}_k &= \sum_j B_k(t_j) \cdot \Delta t
\end{align*} \]

\[
L_{\theta_\epsilon}^{-1} L_{\theta_b} = \text{vec}(\hat{B}_k^{-1} Q_u(t_0) \hat{A}_k^{-T})
\]
### Computation Comparison at Each Stage

**Derivation**

- **1\(^{st}\)-order adjoint method**
  - Higher-order adjoint process (Theorem 1)
  - Low-rank representation (Proposition 2)
  - Second-Order Neural ODE Optimizer (SNOpt)

<table>
<thead>
<tr>
<th></th>
<th>1(^{st})-order</th>
<th>Theorem 1</th>
<th>Proposition 2</th>
<th>SNOpt</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Adjoint backward</strong></td>
<td>( \mathcal{O}(1) )</td>
<td>( \mathcal{O}(1) )</td>
<td>( \mathcal{O}(1) )</td>
<td>( \mathcal{O}(1) )</td>
</tr>
<tr>
<td></td>
<td>( \mathcal{O}(n + m) )</td>
<td>( \mathcal{O}((n + m)^2) )</td>
<td>( \mathcal{O}(Rn + Rm) )</td>
<td>( \mathcal{O}(2n + Rm) )</td>
</tr>
<tr>
<td><strong>Param. update</strong></td>
<td>( \mathcal{O}(n) )</td>
<td>( \mathcal{O}(n^2) )</td>
<td>( \mathcal{O}(n^2) )</td>
<td>( \mathcal{O}(2n) )</td>
</tr>
</tbody>
</table>

- 10-40% additional constant memory compared to 1\(^{st}\)-order Adjoint. (less than 1GB on all experiments)
Results: Computational Efficiency

- Per-iteration runtime (seconds) w.r.t. Adam

<table>
<thead>
<tr>
<th></th>
<th>Image Classification</th>
<th>Time-series Prediction</th>
<th>Continuous NF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MNIST</td>
<td>SVHN</td>
<td>CIFAR10</td>
</tr>
<tr>
<td>Ours</td>
<td>1.00</td>
<td>0.87</td>
<td>1.16</td>
</tr>
<tr>
<td>Adam</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

🚨 run nearly as fast as Adam!
😀 may run faster (e.g., SVHN)!

- Preconditioned updates may lead to implicit regularization.

|                   | # of function evaluation (NFE) | Regularization \( (\int ||\nabla_x F||^2 + \int ||F||^2) \) |
|-------------------|--------------------------------|-------------------------------------------------|
|                   | forward pass | backward pass | Adam | Ours | 32.0 | 42.1 | 323.9 (100%) |
|                   |              |                | 32.0 | 26.0 | 42.1 | 32.6 | 199.1 (61.5%) |
Results: Computational Efficiency

- Superior convergence in wall-clock time (Adam, SGD, SNOpt (ours))
Results: Improvement & Robustness

- Improves test-time performance (accuracy for Image/Time-series, NLL for CNF)

<table>
<thead>
<tr>
<th></th>
<th>MNIST</th>
<th>SVHN</th>
<th>CIFAR10</th>
<th>SpoAD</th>
<th>ArtWR</th>
<th>CharT</th>
<th>Circle</th>
<th>Gas</th>
<th>Miniboone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>98.83</td>
<td>91.92</td>
<td>77.41</td>
<td>94.64</td>
<td>84.14</td>
<td>93.29</td>
<td>0.90</td>
<td>-6.42</td>
<td>13.10</td>
</tr>
<tr>
<td>SGD</td>
<td>98.68</td>
<td>93.34</td>
<td>76.42</td>
<td><strong>97.70</strong></td>
<td>85.82</td>
<td>95.93</td>
<td>0.94</td>
<td>-4.58</td>
<td>13.75</td>
</tr>
<tr>
<td>SNOpt</td>
<td><strong>98.99</strong></td>
<td><strong>95.77</strong></td>
<td><strong>79.11</strong></td>
<td>97.41</td>
<td>90.23</td>
<td>96.63</td>
<td>0.86</td>
<td><strong>-7.55</strong></td>
<td><strong>12.50</strong></td>
</tr>
</tbody>
</table>

- Robustifies hyper-parameter sensitivity.
Results: Comparison to Recursive Baseline

- SNOpt is around 2~5x faster than recursive adjoint baseline.
- Improve test-time accuracy by 5~15%.
Consider an extension of $Q(t, x_t, u_t)$ that includes the terminal horizon $t_1$.

\[
Q(t, x_t, u_t) := \Phi(x_t) + \int_t^{t_1} \ell(\tau, x_\tau, u_\tau) d\tau
\]

\[
\tilde{\Phi} := \Phi + \text{penalty on } T
\]

\[
\tilde{Q}(t, x_t, u_t, T) := \tilde{\Phi}(T, x_T) + \int_t^T \ell(\tau, x_\tau, u_\tau) d\tau
\]

iterative \textit{feedback} update rule

\[
T \leftarrow T - \delta T(\delta \theta), \text{ where }
\delta T(\delta \theta) = [\tilde{Q}_{TT}(t_0)]^{-1} (\tilde{Q}_T(t_0) + \tilde{Q}_{Tu}(t_0)\delta \theta)
\]

self-adaptive policy accounting for parameter update

😀 fast convergence to desired $t_1$ without divergence (as in baseline)

😀 reduce runtime by 20% without hindering test-time accuracy
Conclusion

A new second-order optimizer for training Neural ODEs that

- grounded on optimal control theory
- generalizes first-order adjoint method while retaining the same constant $O(1)$ memory (in depth)
- achieves strong empirical results (e.g., convergence & test-time performance)
- opens up new applications and questions (e.g., architecture optimization, implicit regularization)

* Code is available at [https://github.com/ghliu/snopt](https://github.com/ghliu/snopt)
Weinan et al., 2018, “A mean-field optimal control formulation of deep learning.”
Liu et al., 2021a, “DDPNOpt: Differential dynamic programming neural optimizer.”
Liu et al., 2021b, “Dynamic game-theoretic neural optimizer.”
Pontryagin et al., 1962, "The mathematical theory of optimal processes."
Finlay et al., 2020, "How to train your neural ode: the world of jacobian and kinetic regularization."