Learning Optimal Control Policies with Ordinary Differential Equations

Federico Berto

Lab Seminar

SYSTEM INTELLIGENCE LAB

Background: Continuous-depth Framework

Continuous-depth (time) framework

Analyze and develop models bridging the gap between **dynamical system theory** and **deep learning**

deep learning	$\Rightarrow$	dynamical systems
dynamical systems	$\Rightarrow$	deep learning

Differential equations are everywhere: **physics**, **engineering**, **sciences**.

A Review on Neural ODEs

Neural ODE: a Core Primitive

We seek the deep limit of neural networks

$\Rightarrow$ The input–output map is realized by the *flow* of an ODE[1]

Neural ODE [Sonoda, et al. 2017, Chen et al., 2018]

By noticing that the latent dynamics of a ResNet

$$\boxed{\mathbf{z}_{s+1} = \mathbf{z}_s + f_{\theta_s}(\mathbf{z}_s)}$$

resemble the *Euler* discretization

$$\frac{\mathbf{z}_{s+1} - \mathbf{z}_s}{\Delta s} \approx \frac{\mathrm{d}\mathbf{z}}{\mathrm{d}s} = f_{\theta_s}(\mathbf{z}_s) \quad (\Delta s = 1)$$

of the ODE

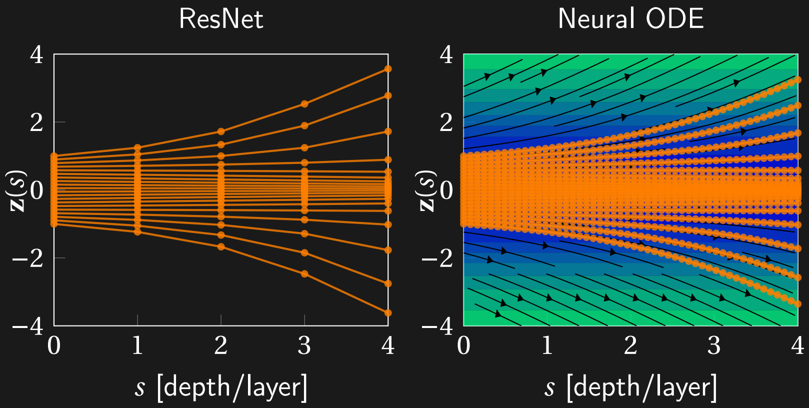
$$\boxed{\begin{aligned} &\frac{\mathrm{d}\mathbf{z}}{\mathrm{d}s} = f_{\theta_s}(s, \mathbf{z}(s)) \qquad s \in \mathcal{S} \subset \mathbb{R} \\ &\mathbf{z}(0) = \mathbf{x} \end{aligned}}$$

[1] *Ordinary Differential Equation*

Neural ODEs vs ResNets

Inference Model

$$\boxed{\hat{\mathbf{y}} = \mathbf{x} + \int_0^S f_\theta(s, \mathbf{z}(s))\,\mathrm{d}s}$$



A *General* Neural ODE Formulation

Neural Ordinary Differential Equation

$$\begin{cases} \dot{\mathbf{z}}(s) = f_{\theta(s)}(s, \mathbf{x}, \mathbf{z}(s)) \\ \mathbf{z}(0) = h_x(\mathbf{x}) \\ \hat{\mathbf{y}}(s) = h_y(\mathbf{z}(s)) \end{cases} \qquad s \in \mathcal{S}$$

Input	$\mathbf{x}$	$\mathbb{R}^{n_x}$
Output	$\hat{\mathbf{y}}$	$\mathbb{R}^{n_y}$
(Hidden) State	$\mathbf{z}$	$\mathbb{R}^{n_z}$
Parameters	$\theta(s)$	$\mathbb{R}^{n_\theta}$
Neural Vector Field	$f_{\theta(s)}$	$\mathbb{R}^{n_z}$
Input Network	h_x	$\mathbb{R}^{n_x} \to \mathbb{R}^{n_z}$
Output Network	h_y	$\mathbb{R}^{n_z} \to \mathbb{R}^{n_y}$

where f_θ, h_x, h_y are generally neural networks.

Inference of general Neural ODEs

$$\hat{\mathbf{y}}(S) = h_y\left(h_x(\mathbf{x}) + \int_0^{S^*_{\mathbf{x}}} f_{\theta(s)}(s, \mathbf{x}, \mathbf{z}(s))\mathrm{d}s\right)$$

An *Optimal Control* Perspective to Training

Loss Function

$$\ell(\theta, \mathbf{x}) := \overbrace{L(\mathbf{z}(S))}^{\texttt{Terminal Loss}} + \underbrace{\int_0^S l(s, \mathbf{z}(s)) \mathrm{d}s}_{\texttt{Integral Loss}}$$

The training can be then cast into the *optimal control* problem

$$\begin{aligned} \min_{\theta \in \mathcal{W}} \quad & \mathbb{E}_{\mathbf{x}} \left[\ell(\theta, \mathbf{x}) \right] \\ \text{subject to} \quad & \dot{\mathbf{z}}(s) = f_{\theta(s)}(s, \mathbf{x}, \mathbf{z}(s)) \quad s \in \mathcal{S} \\ & \mathbf{z}(0) = h_x(\mathbf{x}) \\ & \hat{\mathbf{y}}(s) = h_y(\mathbf{z}(s)) \end{aligned}$$

which we wish to solve via stochastic gradient descent.

Computing Gradients

Two options:

- backpropagate trough the discrete steps of the solver
- backpropagate through the solution of the ODE

Can we compute gradients in closed–form?

Adjoint Method – Intuition

To retrieve the gradient, we have to "unroll" the integral
$\Rightarrow$ solve an other differential equation *backward* in $\mathcal{S}$

Let θ be *constant* in s [Chen T.Q. et al., 2018]

Proposition: Generalized Adjoint Gradients

Consider the loss function $\ell = L(\mathbf{z}(S)) + \int_{\mathcal{S}} l(s, \mathbf{z}(s))\mathrm{d}s$. Then,

$$\frac{\mathrm{d}\ell}{\mathrm{d}\theta} = \int_{\mathcal{S}} \mathbf{a}^\top(s)\frac{\partial f_\theta}{\partial \theta}\mathrm{d}s \quad \text{where } \mathbf{a}(s) \text{ satisfies} \quad \boxed{\begin{cases} \dot{\mathbf{a}}^\top(s) = -\mathbf{a}^\top(s)\dfrac{\partial f_\theta}{\partial \mathbf{z}} - \dfrac{\partial l}{\partial \mathbf{z}} \\ \mathbf{a}^\top(S) = \dfrac{\partial L}{\partial \mathbf{z}(S)} \end{cases}}$$

- This is the exact gradient;
- We do not need to store activations $\Rightarrow$ $\mathcal{O}(1)$ memory efficiency

What about for a general class of $\theta(s)$?[2]

[2] Massaroli, Poli et al., Dissecting Neural ODEs, NeurIPS 2020

Why bother with the framework?

Advantages	**Disadvantages**
$\mathcal{O}(1)$ memory gradients	Expressivity limitations
Learning and Control	Compute requirements
Cheap Normalizing Flows	Novelty
Uncertainty Estimation	Different *tricks*
Stability and Constraints	Less accessible

Hypersolvers for Neural ODEs

Solving the Differential Equation

Solving differential equations is **costly**. Can be an issue for deployment.

Can we do better?

Yes. Analyze interplay between numerical solvers and Neural ODE.[3]
Ties in with pretraining strategies (NLP) and compression techniques.

Hot area at the moment. Regularization techniques [4]also help by controlling the stiffness of learned ODEs, but they are not applicable in general.

[3]Poli, Massaroli et al., Hypersolvers: Toward Fast Continuous-Depth Models, NeurIPS 2020

[4]Finlay et al., How to train your neural ODE: the world of Jacobian and kinetic regularization, ICML 2020. Kelly et al., Learning Differential Equations that are Fast to Solve, NeurIPS2020

Quick Refresher on Explicit ODE Solvers

Iterate to solve:

$$\boxed{\mathbf{z}_{k+1} = \mathbf{z}_k + \epsilon\psi(s_k, \mathbf{x}, \mathbf{z}_k)}$$

- ODE solvers differ in how the map ψ is designed
- *Higher–order* (explicit) solvers compute ψ iteratively in p steps (p denotes the order of the solver)

[example] p-th order *Runge-Kutta* method:

$$\begin{aligned}
\mathbf{r}_i &= f_{\theta(s_k)}(s_k + \mathbf{c}_i\epsilon,\ \mathbf{x},\ \mathbf{z}_k + \tilde{\mathbf{z}}_k^i) && i = 1, \dots, p \\
\tilde{\mathbf{z}}_k^i &= \epsilon \textstyle\sum_{j=1}^{p} \mathbf{a}_{ij}\mathbf{r}_j && i = 1, \dots, p \\
\psi &= \textstyle\sum_{j=1}^{p} \mathbf{b}_j \mathbf{r}_j
\end{aligned}$$

where $\mathbf{a} \in \mathbb{R}^{p \times p}$, $\mathbf{b} \in \mathbb{R}^p$, $\mathbf{c} \in \mathbb{R}^p$ fully characterize the method

Hypersolvers: main idea

Consider a pth order explicit ODE solver.

Discretized Neural ODE Solution

$$\boxed{\begin{cases} \mathbf{z}_{k+1} = \mathbf{z}_k + \epsilon\psi(s_k, \mathbf{x}, \mathbf{z}_k) \\ \mathbf{z}_0 = h_x(\mathbf{x}) & \quad k = 0, 1, \dots, K-1 \\ \hat{\mathbf{y}}_k = h_y(\mathbf{z}_k) \end{cases}}$$

ϵ: step size, ψ: solver's step.

Local Truncation Error $\boxed{e_k := \|\mathbf{z}(s_{k+1}) - \mathbf{z}(s_k) - \epsilon\psi(s_k, \mathbf{x}, \mathbf{z}(s_k))\|_2 = \mathcal{O}\left(\epsilon^{p+1}\right)}$

Hypersolver

$$\mathbf{z}_{k+1} = \mathbf{z}_k + \underbrace{\epsilon\psi(s_k, \mathbf{x}, \mathbf{z}_k)}_{\text{base solver step}} + \epsilon^{p+1} \overbrace{g_\omega(\epsilon, s_k, \mathbf{x}, \mathbf{z}_k)}^{\text{hypersolver net}}$$

Improvement on Solver Error

Given some *nominal* trajectories $\{(s_k, \mathbf{z}(s_k))\}_k$, consider the *residual*

$$\mathcal{R}_k = \frac{1}{\epsilon^{p+1}} \left[\mathbf{z}(s_{k+1}) - \mathbf{z}(s_k) - \epsilon \psi(\mathbf{x}, s_k, \mathbf{z}(s_k)) \right]$$

Residual Training

Consider also the loss function

$$\ell = \frac{1}{K} \sum_{k=0}^{K-1} \| \mathcal{R}(s_k, \mathbf{z}(s_k), \mathbf{z}(s_{k+1})) - g_\omega(\epsilon, \mathbf{x}, s_k, \mathbf{z}(s_k)) \|_2$$

We have that,

$$\boxed{\forall k \quad \|\mathcal{R}_k - g_\omega\|_2 \leq \mathcal{O}(\delta) \quad \Rightarrow \quad e_k = \mathcal{O}(\delta \epsilon^{p+1})}$$

At training, may use a lower–order solver to achieve the same accuracy!
$\Rightarrow$ Accelerate inference of Neural ODEs

Hypersolvers and Optimal Control

Hypersolvers for Optimal Control

Key points:

- We need accurate trajectories for training a controller
- If the optimization is done *online* it is even more crucial to speed up the simulation

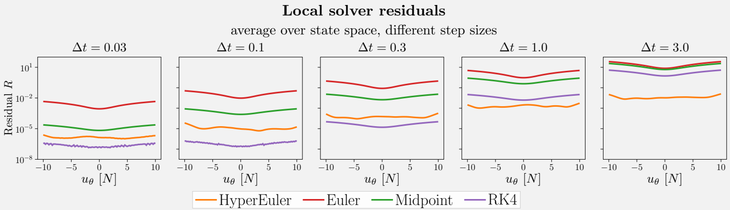
$\implies$ Hypersolvers are helpful to obtain Pareto-optimal forward simulations!
We need to train the hypersolver on a reasonable set of the explored state-space and introduce the control input as an important feature

Training Hypersolvers for Controlled Systems

Residual Training for Controlled Systems

The control value u becomes an important input for the hypersolver

$$\ell = \frac{1}{K}\sum_{k=0}^{K-1}\left\| R\left(t_k, x(t_k), x(t_{k+1})\right) - g_\omega\left(t_k, x(t_k), \dot{x}(t_k), u(t_k)\right)\right\|_2$$



Direct Optimal Control

We want to obtain a control policy by *directly* optimizing the cost function over the complete trajectories

$$
\begin{aligned}
\min_{u} \quad & J_u \\
\text{subject to} \quad & \dot{x}(t) = f(t, x(t), u(t)) \\
& x \in \mathbb{X}; \;\; u \in \mathbb{U}
\end{aligned}
$$

where u is the control policy and cost function J is

$$J_u = x^\top(t_f)\mathbf{P}x(t_f) + \int_{t_0}^{t_f} \left[x^\top(t)\mathbf{Q}x(t) + u^\top(t)\mathbf{R}u(t)\right] dt$$

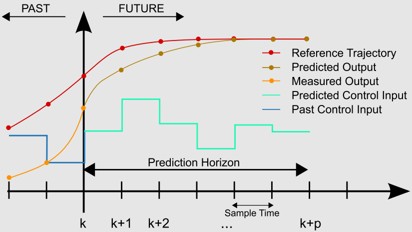
$\implies$ Hypersolvers make the simulation of whole trajectories more accurate and efficient compared to classical solvers!

Model Predictive Control (MPC)

The following optimization problem is solved *online* at each time step:

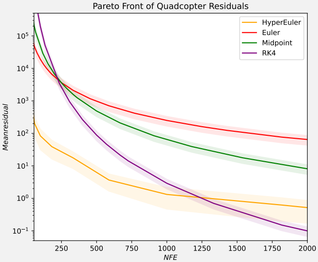
$$\begin{aligned} \min_{u} \quad & \sum_{i=1}^{T} J_u(x(t_i)) \\ \text{subject to} \quad & \dot{x}(t) = f(t, x(t), u(t)) \\ & x \in \mathbb{X};\ u \in \mathbb{U} \end{aligned}$$

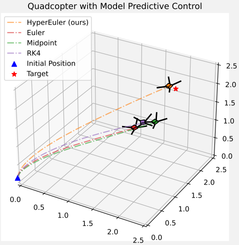
u: *control policy*; T: *receding horizon* in which future trajectories are predicted



Accelerating Model Predictive Control

MPC is an online optimization algorithm, it is even more crucial to obtain *fast and accurate* trajectory predictions





Results on a quadcopter model (17 states and 4 control variables)

Multi-stage Hypersolvers

If the dynamic model does not match perfectly the real one, we can use an **additional first-order term** to correct the vector field while the **second-order term** can be used to further improve accuracy

Multi-stage HyperEuler Step

$$x_{k+1} = x_k + \overbrace{\epsilon f(t_k, x_k, u_k)}^{\text{Euler step}} + \epsilon \overbrace{g'_w(t_k, x_k, u_k)}^{\text{1st order residual approximator}} + \epsilon^2 \underbrace{g''_\omega(t_k, x_k, u_k)}_{\text{2nd order residual approximator}}$$

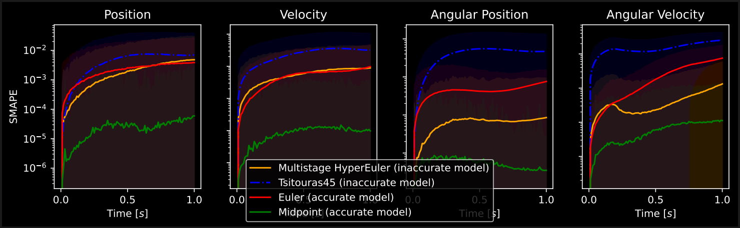
Training Multiple Stages

If the *nominal* system model is unknown, we can not know what the residual directly: we can train directly on the known nominal trajectories

Training multi-stage hypersolvers

$$\ell = \frac{1}{K}\sum_{k=0}^{K-1} MSE\left(x(t_{k+1}), x_{k+1}\right)$$

where $x(t_{k+1})$ is obtained by the real (nominal) system and x_{k+1} is one step of the Multi-stage HyperEuler



Extra: `torchdyn`

The new `torchdyn` 1.0 is out! Check it out at
https://github.com/DiffEqML/torchdyn

PyTorch library dedicated to neural differential equations and implicit models. Maintained by DiffEqML.

Thank you for your Attention!