ON EFFICIENT TRAINING & INFERENCE OF NEURAL DIFFERENTIAL EQUATIONS

By

AVIK PAL

B.TECH., INDIAN INSTITUTE OF TECHNOLOGY KANPUR (2017)

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2023

Authored By: Avik Pal
Department of Electrical Engineering and Computer Science
April 13, 2023

Certified By: Alan Edelman
Professor of Applied Mathematics
Thesis Supervisor

Certified By: Chris Rackauckas
Applied Mathematics Instructor
Thesis Supervisor

Accepted By: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Abstract

Democratization of machine learning requires architectures that automatically adapt to new problems. Neural Differential Equations have emerged as a popular modeling framework, enabling ML practitioners to design neural networks that can adaptively modify their depth based on the input problem. Neural Differential Equations combine differential equations with neural networks and rely on adaptive differential equation solvers for the forward process.

The flexibility of automatically adapting the depths comes with the cost of expensive training and slower predictions. Several prior works have tried to accelerate training and inference. However, almost all of them have severe tradeoffs. Either these works rely on expensive training methods to accelerate predictions or use algorithms that are harder to integrate into existing workflows.

In this thesis, we will discuss two methods to accelerate Neural Differential Equations. We propose an Infinite Time Neural ODE, which paradoxically can be trained faster than integrating a Neural ODE to a fixed time-point. Additionally, we build upon prior works on regularized Neural ODEs and propose a stochastic local regularization scheme which can be used as a drop-in replacement for Neural ODEs.

(**todo**) carry over from proposal

Originality

Statement

Publications

Open Source Software

Breakdown of Contributions

Other Works

Acknowledgement

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

(**todo**) carry over from proposal

In this thesis, we will answer the following research questions:

1. Can we design a `continuous variant of DEQs` that is equivalent to solving a Neural ODE? Why would `integrating a Neural ODE to infinity` accelerate the training?
2. Can we `accelerate training and inference of Neural Differential Equations` without constraining the models to use specific adjoint methods?

Part I

FOUNDATIONS

CHAPTER 2

Neural Ordinary Differential Equations

1 Ordinary Differential Equations

Ordinary Differential Equations (ODEs) are equations defined by a relationship to their derivative. We can generally write an ODE as:

$$\frac{dz}{dt} = f(z, p, t) \tag{2.1}$$

This effectively describes the evolution of a state $z(t)$. To compute $z(t)$, we could solve the following integral equation:

$$z(t) = \int_{t_0}^{t} f(z, p, t)\ \mathrm{d}t \tag{2.2}$$

where t_0 is the initial time. However, computing these solutions analytically is almost impossible, and hence we need to rely on numerical solvers. For this thesis, we will focus exclusively on Initial Value Problems (IVP) which specify the differential equations along with an initial condition, i.e., the value of the state at t_0: $z(t_0)$. There are other kinds of ODE Problems like Boundary Value Problems (BVP) which specify additional conditions at the end of the interval, i.e., $z(t_1)$.

Numerical Solvers for ODEs can be broadly categorized into Implicit Methods and Explicit Methods. Explicit Methods compute the state of the dynamical system at a future time-point given the current state. Implicit Methods solve for the state of the dynamical system at a future time-point given the current state *and the later one*. For example, consider two extremely simple numerical solvers for an ODE:

- Euler Method (Explicit Method) [Euler, 1824]:

$$z_{n+1} = z_n + \mathrm{d}t \cdot f(z_n, p, t_n) \tag{2.3}$$

- Backward Euler Method (Implicit Method) [Euler, 1824]:

$$z_{n+1} = z_n + \mathrm{d}t \cdot f\left(z_{n+1}, p, t_{n+1}\right) \tag{2.4}$$

Explicit methods tend to be faster than implicit methods but are not effective for solving stiff equations [Wanner and Hairer, 1996, Kim et al., 2021]. In this thesis, we will exclusively use explicit methods. A detailed discussion on implicit and explicit methods is beyond the scope of this thesis, and we refer the readers to Rackauckas [2019].

2 Explicit ODE Solvers

In this section, we will briefly discuss ODE solvers that are relevant in the context of Neural ODEs.

2.1 Tsitouras $5(4)$ Runge-Kutta (Tsit5) Method

Runge-Kutta (RK) Methods [Runge, 1895, Kutta, 1901] are widely used to approximate the solutions of ODEs numerically. Given a tableau of coefficients $\{A, c, b\}$, these methods combine s stages to obtain the estimate at $t + \mathrm{d}t$.

$$k_s = f\left(t + c_s \cdot \mathrm{d}t, z(t) + \sum_{i=1}^{s-1} a_{si} \cdot k_i\right) \tag{2.5}$$

$$z(t + dt) = z(t) + \mathrm{d}t \cdot \left(\sum_{i=1}^{s} b_i \cdot k_i\right) \tag{2.6}$$

Tsitouras [2011] presented a tableau of coefficients for a 6-stage RK method of order $5(4)$. We have found Tsit5 to be an excellent default for most ODEs (including Neural ODEs) at lower tolerances.

2.2 3^{rd} order Adams-Bashforth (VCAB3) Method

Contrary to RK Methods, Multi-step methods compute $z\left(t\right)$ by efficiently using the information from previous time steps. A linear multi-step method uses linear interpolation to compute z_{n+1}:

$$z_{n+1} = z_n + \mathrm{d}t \cdot \sum_{i=0}^{s} \beta_i \cdot f\left(z_{n+1-i}, p, t_{n+1-i}\right) \qquad \texttt{given} \quad \sum_{i=0}^{s} \beta_i = 1 \tag{2.7}$$

where s is the number of steps. If $\beta_0 = 0$, then we have an explicit method. In this thesis, we will focus on Adams methods which involve solving the following:

$$z_{n+1} = z_n + \int_{t_n}^{t_{n+1}} f\left(z(\tau), p, \tau\right) \cdot \mathrm{d}\tau \tag{2.8}$$

Adams methods approximate the integral using polynomial interpolation of the function f using evaluations at points $\{t_{n+1-s}, t_{n+2-s}, \ldots, t_n\}$. Adams-Bashforth Method approximates the function using Lagrange Interpolation:

$$f\left(z(\tau), p, \tau\right) \approx \sum_{i=0}^{s} \mathcal{L}_{n+1-i} \cdot f\left(z_{n+1-i}, p, t_{n+1-i}\right) \tag{2.9}$$

where $\mathcal{L}_{n+1-i}$'s are the Lagrange polynomials. For $s = 3$, we get the 3^{rd} order Adams-Bashforth (VCAB3) Method [Durran, 1991]:

$$z_{n+1} = z_n + \frac{\mathrm{d}t}{12} \cdot \left(23f\left(z_n, p, t_n\right) - 16f\left(z_{n-1}, p, t_{n-1}\right) + 5f\left(z_{n-2}, p, t_{n-2}\right)\right) \tag{2.10}$$

This method has a local truncation error of $O\left(\mathrm{d}t^3\right)$. Additionally, since this method evaluates f infrequently (by reusing the evaluations of f from previous time steps), it is efficient for ODEs with expensive evaluations of f like a Neural ODE.

3 Adaptive Time-Stepping in Numerical ODE Solvers

Adaptive solvers need to maximize the step size ($\mathrm{d}t$) while keeping the error estimate below the user-specified tolerances, i.e., they need to satisfy the:

$$\mathtt{E_{Est}} \leq \mathtt{atol} + \mathtt{max}\left(|z(t)|, |z(t+\mathrm{d}t)|\right) \cdot \mathtt{rtol} \tag{2.11}$$

where $\mathtt{E_{Est}}$ is the local error estimate. Adaptive solvers utilize an additional linear combiner $\tilde{b}_i$ to get an alternate solution, typically with one order less convergence [Wanner and Hairer, 1996, Fehlberg, 1968, Dormand and Prince, 1980, Tsitouras, 2011].

$$\tilde{z}(t+dt) = z(t) + \mathrm{d}t \cdot \left(\sum_{i=1}^{s} \tilde{b}_i \cdot k_i\right) \tag{2.12}$$

A classic result from Richardson extrapolation shows that $\mathtt{E_{Est}} = \|\tilde{z}(t+\mathrm{d}t) - z(t+\mathrm{d}t)\|$ is an estimate of the local truncation error [Hairer et al., 1993, Ascher and Petzold, 1998]. The new step size is determined using the following:

$$q = \left\| \frac{\mathtt{E_{Est}}}{\mathtt{atol} + \mathtt{max}\left(|z(t)|, |z(t+\mathrm{d}t)|\right) \cdot \mathtt{rtol}} \right\| \tag{2.13}$$

- If $q < 1$, $\mathrm{d}t$ is accepted.
- Otherwise, it is rejected and reduced. A standard PI controller proposes the new step size to be:

$$dt_{new} = \eta \cdot q_{n-1}^{\alpha} \cdot q_n^{\beta} \cdot dt \tag{2.14}$$

  where η is the safety factor, q_{n-1} is the error proportion of the previous step, and (α, β) are tunable PI-gain hyperparameters Wanner and Hairer [1996].

We defer the discussion of error estimation schemes for stochastic RK integrators to Rackauckas and Nie [2017b, 2020].

4 Automatic Stiffness Detection

While there is no precise definition of stiffness, the definition used in practice is "stiff equations are problems for which explicit methods don't work" [Wanner and Hairer, 1996, Shampine and Gear, 1979]. A simplified stiffness index is given by:

$$S = \max \|\mathtt{Re}(\lambda_i)\| \tag{2.15}$$

Sensitivity Algorithm	Memory Requirement	Memory Requirement with Checkpointing
Backsolve Adjoint [Chen et al., 2018]	$O(s)$	$O(s \times c)$
Interpolating Adjoint [Hindmarsh et al., 2005]	$O(s \times t)$	$O(s \times c)$
Quadrature Adjoint [Kim et al., 2021]	$O((s+p) \times t)$	-
Direct Reverse Mode Differentiation	$O(s \times t \times \text{stages})$	-

Table 2.1: Memory Requirements for various Sensitivity Algorithms for ODEs: s is the number of states, t is the number of time steps for the ODE Solve, c is the number of checkpoints ($c << t$) and stages is the number of stages of the ODE solver.

where λ_i are the eigenvalues of the local Jacobian matrix. We note that various measures of stiffness have been introduced over the years, all being variations of conditioning of the pseudo-spectra [Shampine and Thompson, 2007, Higham and Trefethen, 1993]. The difficulty in defining a stiffness metric is that in each case, stiff systems like the classic Robertson chemical kinetics or excited Van der Pol equation may violate the definition, meaning all such definitions are (useful) heuristics. In particular, it was shown that for explicit Runge-Kutta methods satisfying $c_x = c_y$ for some internal step, the term

$$\|\lambda\| \approx \left\| \frac{f\left(t + c_x \cdot \mathrm{d}t, \sum_{i=1}^{s} a_{xi}\right) - f\left(t + c_y \cdot \mathrm{d}t, \sum_{i=1}^{s} a_{yi}\right)}{\sum_{i=1}^{s} a_{xi} - \sum_{i=1}^{s} a_{yi}} \right\| \tag{2.16}$$

serves as an estimate to S [Shampine, 1977]. Since each of these terms are already required in the Runge-Kutta updates of Equation 2.6, this gives a computationally-free estimate. This estimate is thus found throughout widely used explicit Runge-Kutta implementations, such as by the dopri method (found in suites like SciPy and Octave) to exit when stiffness is detected automatically [Wanner and Hairer, 1996], and by switching methods which automatically change explicit Runge-Kutta methods to methods more suitable for stiff equations [Rackauckas and Nie, 2019].

5 Sensitivity Analysis of ODEs

5.1 Continuous Sensitivity Analysis

A detailed discussion of different continuous sensitivity algorithms is beyond the scope of this thesis. Instead, we present the algorithm of one of the popular methods (especially for training Neural ODEs) – Backsolve Adjoint. We refer the readers to Chen et al. [2018, Appendix B] for rigorous proof of this method.

Let the gradient of the loss $\mathcal{L}$ wrt the final state of the ODE be $z(t_1)$ be $\frac{\partial L}{\partial z(t_1)}$. Define $\lambda(t)$ to be an augmented state. To compute the adjoint, we solve an augmented ODE from t_1 to t_0 with the initial condition:

$$s_0 = \left[z(t_1), \frac{\partial L}{\partial z(t_1)}, 0_{|p|}, -\left(\frac{\partial L}{\partial z(t_1)}\right)^T f\left(z(t_1), p, t_1\right) \right] \tag{2.17}$$

The augmented dynamics is given by the following:

$$\frac{dz(t)}{dt} = f\left(z(t), p, t\right) \tag{2.18}$$

$$\frac{d\lambda(t)}{dt} = -\lambda(t)^T \frac{\partial f\left(z(t), p, t\right)}{\partial z} \tag{2.19}$$

$$\frac{d\lambda_p(t)}{dt} = -\lambda(t)^T \frac{\partial f\left(z(t), p, t\right)}{\partial p} \tag{2.20}$$

$$\frac{d\lambda_t(t)}{dt} = -\lambda(t)^T \frac{\partial f\left(z(t), p, t\right)}{\partial t} \tag{2.21}$$

After solving this augmented ODE, we compute the final gradients as:

$$\frac{d\mathcal{L}}{dp} = \lambda_p(t_0) \qquad \frac{d\mathcal{L}}{dz(t_0)} = \lambda(t_0) \tag{2.22}$$

$$\frac{d\mathcal{L}}{dt_0} = \lambda_t(t_0) \qquad \frac{d\mathcal{L}}{dt_1} = \lambda_t(t_1) \tag{2.23}$$

In this algorithm, we don't need to store any of the intermediate activations in the forward solve, hence it is extremely memory efficient and has a complexity of $O\left(s\right)$ (See Table 2.1).

5.2 Discrete Sensitivity Analysis

Discrete Sensitivity Analysis is the same as running any reverse mode AD software directly through the solver. Hence, we need to store the activations at every time step the function was evaluated making the memory cost of the method extremely high. This method doesn't scale too well for larger systems, however, has certain nice properties that can be exploited to accelerate Neural ODEs (See Section 4).

5.3 Tradeoffs between Continuous and Discrete Sensitivity Analysis

Using continuous sensitivity analysis or Optimize-then-Discretize (Opt-Disc) we "optimize the continuous ODE and then discretize the optimal dynamics after training" [Onken and Ruthotto, 2020]. Alternatively in discrete sensitivity analysis or Discretize-then-Optimize (Disc-Opt) we discretize the ODE and then compute the sensitivities over that discretization. Opt-Disc has a clear memory advantage over Disc-Opt (See Table 2.1). Gholami et al. [2019] compared Disc-Opt and Opt-Disc and concluded the following for image classification tasks:

1. Opt-Disc leads to numerical instabilities in general deep neural network operations like convolutions, etc.
2. Inconsistent gradients in Opt-Disc can lead to divergence

Onken and Ruthotto [2020] extended this discussion to time-series problems and continuous normalizing flows and showed that Disc-Opt obtains similar or better results compared to Opt-Disc while having a lower computational cost.

6 Neural Ordinary Differential Equations

Neural Ordinary Differential Equations are Implicit Neural Networks that use a neural network to parameterize the dynamics of the ODE: Mathematically, this is given by:

$$\frac{dz}{dt} = f\left(z(t), \theta, t\right) \qquad z(t_0) = z_0 \tag{2.24}$$

$$z\left(t_1\right) = z\left(t_0\right) + \int_{t_0}^{t_1} f\left(z(t), \theta, t\right)\, \mathrm{d}t \tag{2.25}$$

where f is an Explicit Neural Network, θ are the parameters of the neural network, and we want to solve for the dynamics $z\left(t\right)$ in $t \in [t_0, t_1]$. Typically, z_0 is specified as the input from the previous layers. Then we can use sensitivity analysis (Section 5) to compute $v^T \frac{\partial z(t)}{\partial z_0}\Big|_{t=t_1}$ and $v^T \frac{\partial z(t)}{\partial g}\Big|_{t=t_1}$ (where v^T is obtained from back-propagation on the succeeding layers) and train the neural network end-to-end using gradient descent.

7 Common Applications of Neural ODEs

Neural ODEs have emerged as a direct alternative to explicit neural networks that can automatically adapt its depth to the problem. However, apart from being a continuous replacement to standard explicit models, Neural ODEs have certain specific modelling advantages that make them a good fit for certain problems. In this section, we discuss some of those applications of Neural ODEs.

7.1 Density Estimation: Continuous Normalizing Flows and FFJORD

Given a random variable $z \sim \mathcal{P}_z$, we can compute the probability distribution of $x = f(z) \sim \mathcal{P}_x$, where $f : \mathbb{R}^D \mapsto \mathbb{R}^D$ is an invertible function, as:

$$\log p_x(x) = \log p_z(z) - \log \det \left|\frac{\partial f(z)}{\partial z}\right| \tag{2.26}$$

Chen et al. [2018] use a neural ODE to transform a sample from a simple distribution $\mathcal{P}_{z_0}$ to the target distribution $\mathcal{P}_{z_1}$. The exact log likelihood of the sample $z(t_1)$ is given by an ODE:

$$\log p_{z_1}\left(z(t_1)\right) = \log p_{z_0}\left(z(t_0)\right) - \int_{t_0}^{t_1} \mathtt{Tr}\left|\frac{\partial f(z(t), p, t)}{\partial z(t)}\right| \mathrm{d}t \tag{2.27}$$

During training the Continuous Normalizing Flow (CNF) we want to obtain the sample z_0 that generated the data $x \in \mathbb{R}^D$, and the corresponding log probability $\log p(x)$. This is done by solving the following equations (note the backward integration from t_1 to t_0):

$$\begin{bmatrix} z(t_0) \\ \log p_{z_1}(x) - \log p_{z_0}(z(t_0)) \end{bmatrix} = \int_{t_1}^{t_0} \begin{bmatrix} f\left(z(t), p, t\right) \\ -\mathtt{Tr}\left|\frac{\partial f(z(t), p, t)}{\partial z(t)}\right| \end{bmatrix} \mathrm{d}t \tag{2.28}$$

$$\begin{bmatrix} z(t_1) \\ \log p_{z_1}(x) - \log p_{z_1}(z(t_1)) \end{bmatrix} = \begin{bmatrix} x \\ 0 \end{bmatrix} \qquad \text{initial conditions} \tag{2.29}$$

Solving this problem reduces the time complexity of $O\left((DH + D^3)L\right)$ (where H is the size of the largest hidden dimension in f and L is the number of transfomations) in Normalizing Flows to $O\left((DH + D^2)\hat{L}\right)$ (where $\hat{L}$ is the number of function evaluations for the CNF). Grathwohl et al. [2018] further improve the time complexity by using the Hutchinson Trace Estimator [Hutchinson, 1989] to estimate $\mathsf{Tr}\left|\frac{\partial f(z(t),p,t)}{\partial z(t)}\right|$.

$$\log p_{z_1}(z(t_1)) = \log p_{z_0}(z(t_0)) - \int_{t_0}^{t_1} \mathsf{Tr}\left|\frac{\partial f(z(t),p,t)}{\partial z(t)}\right| \mathrm{d}t \tag{2.30}$$

$$= \log p_{z_0}(z(t_0)) - \int_{t_0}^{t_1} \mathbb{E}_{p_\epsilon}\left[\epsilon^T \frac{\partial f(z(t),p,t)}{\partial z(t)}\epsilon\right] \mathrm{d}t \tag{2.31}$$

$$= \log p_{z_0}(z(t_0)) - \mathbb{E}_{p_\epsilon}\left[\int_{t_0}^{t_1} \epsilon^T \frac{\partial f(z(t),p,t)}{\partial z(t)}\epsilon \,\mathrm{d}t\right] \tag{2.32}$$

$\epsilon^T \frac{\partial f(z(t),p,t)}{\partial z(t)}$ is a VJP that can be directly obtained using Reverse Mode AD. This transformation further reduces the time complexity to $O\left((DH + D)\right)\hat{L}$.

7.2 Time Series Predictions

Standard Recurrent Neural Networks (RNNs) ignore the time spacing between subsequent predictions, hence when used in the context of time series predictions, they work well then the data is evenly separated. However, for irregularly spaced time series data, RNNs require discretizing the observation times or imputing the data as a preprocessing.

Chen et al. [2018] use an latent ODE model with an RNN encoder to capture the dynamics of the time series data. They use a Variational AutoEncoder (VAE) and compute the approximate posterior distribution of the latent variables z given the data sequence x:

$$q\left(z_0 \mid \{x_i, t_i\}_{i=0}^N\right) = \mathcal{N}\left(\mu_{z_0}, \sigma_{z_0}\right) \tag{2.33}$$

$$\mu_{z_0}, \sigma_{z_0} = \mathbf{RNN}\left(\{x_i, t_i\}_{i=0}^N\right) \tag{2.34}$$

The generative process works by integrating a Neural ODE from t_0 to t_N:

$$\hat{x}_i = \hat{x}_{i-1} + \int_{t_{i-1}}^{t_i} f\left(z(t), p, t\right) \mathrm{d}t \qquad \forall i \in [N] \quad x_0 \sim \mathcal{N}\left(\mu_{z_0}, \sigma_{z_0}\right) \tag{2.35}$$

Rubanova et al. [2019] extended this framework to allow encoding irregular spaced data. They used an ODE-RNN encoder, i.e. a Neural ODE that models the hidden state dynamics of the RNN. Let, h_{t_i} be the hidden state of the RNN at time t_i. Since the encoding is performed backwards, i.e. from t_N to t_0, the hidden state $h_{t_{i-1}}$ is given by:

$$h_{t_{i-1}} = h_{t_i} + \int_{t_i}^{t_{i-1}} g\left(h(t), p, t\right) \mathrm{d}t \tag{2.36}$$

Using an ODE-RNN encoder, allows the model to encode non-uniformly spaced data by simply updating the integration time-span of the ODE.

8 Accelerating Neural ODEs

Neural ODEs are competitive against explicit models in terms of accuracy and outperform them wrt memory requirements. However, their widespread adoption is bottlenecked by higher training costs and eventual slow-down over training due to emergent complicated dynamics. Hence, to make them competitive against explicit models several approaches have been proposed to accelerate Neural ODEs [Finlay et al., 2020, Kelly et al., 2020, Behl et al., 2020, Poli et al., 2020, Kidger et al., 2021, Pal et al., 2021, Xia et al., 2021, Zhuang et al., 2021, Djeumou et al., 2022, Pal et al., 2023]. In this section, we discuss some of these prior works.

8.1 Taylor Neural ODE

In Section 3, we discussed adaptive timestepping for RK schemes. A limiting factor in taking large and accurate time steps is the K^{th} order Taylor Coefficients of the solution trajectory. Kelly et al. [2020] proposed the following regularization scheme to minimize the K^{th} order Taylor Coefficients:

$$(\mathcal{R}_K)_g = \int_{t_0}^{t_1} \left\| \frac{d^K z(t)}{dt^K} \right\|_2^2 \mathrm{d}t \tag{2.37}$$

Computing the K^{th}-order gradients for $\frac{d^K z(t)}{dt^K}$ is computationally prohibitive. However, Kelly et al. [2020] used Taylor-Mode Automatic Differentiation [Griewank and Walther, 2008, Bettencourt et al., 2019] to reduce the exponential Time Complexity to $O\left(K^2\right)$ or $O\left(K \log K\right)$ (based on the operations).

8.2 STEER

Neural ODEs tend to learn more complex dynamics as the training progresses. Since the complexity of the dynamics and time taken by the solver are intrinsically related, the training and inference time grows over training. Behl et al. [2020] stochastically perturb the ending integration time-point of the ODE to allow the Neural ODE to learn simpler dynamics. During training, they reformulate the problem as:

$$z\left(t_1\right) = z\left(t_0\right) + \int_{t_0}^{T} f\left(z(t), p, t\right)\, \mathrm{d}t \tag{2.38}$$

$$\texttt{where}\quad T \sim \mathcal{U}(t_1 - b, t_1 + b) \tag{2.39}$$

$$b < t_1 - t_0 \tag{2.40}$$

8.3 Hypersolvers for Neural ODEs

Poli et al. [2020] proposed Hypersolvers to speed up inference of Neural ODEs. They trained a hypersolver to augment a base ODE solver to match the truncation error of a more accurate ODE solver. Let,

$$z_{n+1} = z_n + \mathrm{d}t \cdot \psi(z_n, p, t_n) \tag{2.41}$$

where ψ is the update from an Explicit ODE Integrator. We can represent the steps of a γ^{th} order Hypersolved Neural ODE to be:

$$z_{n+1} = z_n + \mathrm{d}t \cdot \psi(z_n, p, t_n) + \mathrm{d}t^{\gamma+1} \cdot g_\omega(z_n, z_0, \mathrm{d}t, t_n) \tag{2.42}$$

where g_ω is a neural network that is trained to match the residual of the base ODE solver and a more accurate solver. The typical training process involves:

- Train a Neural ODE with an accurate ODE solver
- Fixed the weights of the Neural ODE
- Train the Hypersolver to match the residual of the Neural ODE solved using the base ODE integrator with hypersolver and the more accurate ODE solver.

Poli et al. [2020] demonstrated that Hypersolvers can speedup inference of Neural ODEs by 8x over Dopri5 [Dormand and Prince, 1980] for certain image classification tasks.

9 Conclusion

In this chapter, we have have briefly covered numerical methods to solve ODEs specifically in the context of Neural ODEs. Additionally, we have described some algorithms for sensitivity analysis of ODEs that allow us to compute gradients and perform gradient based optimization of Neural ODEs. We covered various applications of Neural ODEs like time series modelling and generative modelling. Finally, we have discussed prior works that have accelerated the training and inference of Neural ODEs.

However, most of these prior works have focused on either accelerating the training or the inference. Additionally, speeding up inference often comes at the price of slower training or tools that are not easily applicable. In this thesis, we will describe methods that can accelerate both training and inference of Neural ODEs while being easily composable with existing methods.

CHAPTER 3

Deep Equilibrium Models

1 Steady State Problems

Steady State Problems involve determining the equilibrium state of a system, i.e., the state of the system where the rate of change of the system is zero. Let, our steady state problem be defined as follows:

$$\frac{dz}{dt} = f_\theta\left(z, t\right) - z \tag{3.1}$$

In the case of continuous dynamical systems, the steady state would be determined by the partial derivative w.r.t. time being zero:

$$\frac{dz}{dt} = 0 \tag{3.2}$$

In case of discrete dynamical systems, the steady state would be defined by states remaining constant over time:

$$z_{n+1} = z_n \implies z^* = f_\theta\left(z^*, \infty\right) \tag{3.3}$$

There are two ways to solve steady-state problems:

1. **Treating it as a Nonlinear Problem**:
2. **Treating it as an ODE**:
3. **Using Pseudo-Transient Methods**:

2 Sensitivity Analysis of Steady State Problems

Let, z^* be the steady state solution of the system, i.e., $z^* = f_\theta(z^*, \infty)$. For the sake of brevity, let us drop the $t = \infty$ term, i.e., $z^* = f_\theta(z^*)$. Differentiating w.r.t. θ we get:

$$\frac{\partial z^*}{\partial \theta} = \frac{\partial f_\theta(z^*)}{\partial z^*} \times \frac{\partial z^*}{\partial \theta} + \frac{\partial f_\theta(z^*)}{\partial \theta} \tag{3.4}$$

$$\implies \left(I - \frac{\partial f_\theta(z^*)}{\partial z^*}\right) \times \frac{\partial z^*}{\partial \theta} = \frac{\partial f_\theta(z^*)}{\partial \theta} \tag{3.5}$$

Let, the cost function we are optimizing be $g(z^*, \theta)$. Taking the total derivative of $g(z^*, \theta)$ w.r.t. the parameters θ we get:

$$\frac{dg(z^*, \theta)}{d\theta} = \frac{\partial g(z^*, \theta)}{\partial z^*} \times \frac{\partial z^*}{\partial \theta} + \frac{\partial g(z^*, \theta)}{\partial \theta} \tag{3.6}$$

$$= \frac{\partial g(z^*, \theta)}{\partial z^*} \times \left(I - \frac{\partial f_\theta(z^*)}{\partial z^*}\right)^{-1} \times \frac{\partial f_\theta(z^*)}{\partial \theta} + \frac{\partial g(z^*, \theta)}{\partial \theta} \tag{3.7}$$

The RHS term involves the inverse of the jacobian $\frac{f_\theta(z^*)}{\partial z^*}$, computing which is both computationally and memory-wise inefficient. Instead of directly computing the RHS, we can rewrite Equation 3.7 as:

$$\lambda^T = \frac{\partial g(z^*, \theta)}{\partial z^*} \times \left(I - \frac{\partial f_\theta(z^*)}{\partial z^*}\right)^{-1} \tag{3.8}$$

$$\implies \lambda^T \times \left(I - \frac{\partial f_\theta(z^*)}{\partial z^*}\right) = \frac{\partial g(z^*, \theta)}{\partial z^*} \tag{3.9}$$

$$\implies \left(I - \frac{\partial f_\theta(z^*)}{\partial z^*}\right)^T \times \lambda = \left(\frac{\partial g(z^*, \theta)}{\partial z^*}\right)^T \tag{3.10}$$

Substituting λ^T in the Equation 3.7, we get:

$$\frac{dg(z^*, \theta)}{d\theta} = \lambda^T \times \frac{\partial f_\theta(z^*)}{\partial \theta} + \frac{\partial g(z^*, \theta)}{\partial \theta} \tag{3.11}$$

For small systems we can compute $\frac{\partial f_\theta(z^*)}{\partial z^*}$ using forward mode automatic differentiation and solve Equation 3.10 using any linear solver. However, as the scale of the problem increases, these methods are inefficient and often practically infeasible. We defer the discussion on how to deal with large scale steady state problems to Section 3.3. Hence, sensitivity analysis of a (small) steady state problem boils down to solving a linear system of equations and a matrix-vector product.

3 Deep Equilibrium Networks

Deep Equilibrium Networks (DEQs) [Bai et al., 2019] are implicit models where the output space represents a steady-state solution. Intuitively, this represents infinitely deep neural networks with input injection, i.e., an infinite composition of explicit layers $z_{n+1} = f_\theta(z_n, x)$ with $z_0 = 0$ and $n \to \infty$. In practice, it is equivalent to evaluating a dynamical system until it reaches a steady state $z^* = f_\theta(z^*, x)$.

Evaluating DEQs requires solving a steady-state equation involving multiple evaluations of the explicit layer slowing down the forward pass. However, driving the solution to steady-state makes the backward pass very efficient [Johnson, 2006] (See Section 2). Despite a potentially infinite number of evaluations of f_θ in the forward

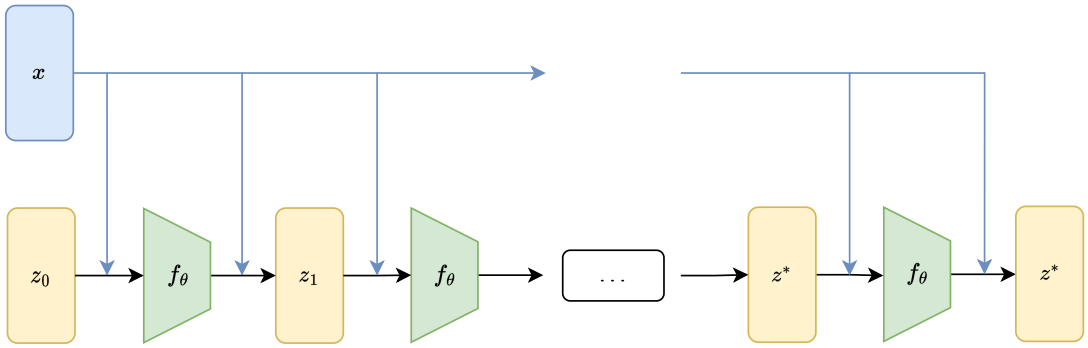

Figure 3.1: Discrete DEQ Formulation: Discrete DEQ Block where the input x is injected at every iteration till the system (with initial state z_0) converges to a steady z^*.

pass, backpropagation only requires solving a linear equation.

3.1 Nonlinear Solvers

In this section, we will exclusively discuss Nonlinear Solvers for solving large steady state problems (i.e., systems with thousands of states).

3.1.1 Broyden's Method

Newton's method is a widely used iterative method for solving nonlinear systems of equations. It is an iterative method that uses the Jacobian matrix to update the solution vector in each iteration.

$$x^{(k+1)} = x^{(k)} - \left(\frac{\partial f\left(x^{(k)}\right)}{\partial x}\right)^{-1} f\left(x^{(k)}\right) \tag{3.12}$$

However, computing the Jacobian matrix is computationally expensive (cubic time complexity) and memory-wise inefficient. Broyden's method [Broyden, 1965] is a quasi-Newton method that approximates the Jacobian matrix using the updates to the solution vector from previous iterations. Specifically, let $f(x)$ be a nonlinear function of the vector x, and let $x^{(k)}$ denote the solution vector at the k-th iteration. The approximation to the inverse of the Jacobian matrix at iteration k is given by:

$$B^{(k)} = B^{(k-1)} + \left(\frac{\Delta x^{(k)} - B^{(k-1)}\Delta f^{(k)}}{\Delta x^{(k)^T} B^{(k-1)} \Delta f^{(k)}}\right)\left(\Delta x^{(k)^T} B^{(k-1)}\right) \tag{3.13}$$

where $\Delta x^{(k)} = x^{(k)} - x^{(k-1)}$ is the step vector, and $\Delta f^{(k)} = f\left(x^{(k)}\right) - f\left(x^{(k-1)}\right)$ is the difference in the function values at the current and previous iterations. $B^{(k-1)}$ is the approximation to the Jacobian matrix at the previous iteration.

The solution vector at the k-th iteration is then updated using the following equation:

$$x^{(k+1)} = x^{(k)} - B^{(k)} f(x^{(k)}) \tag{3.14}$$

Broyden's method has several advantages over Newton's method, including a lower computational cost per iteration, making it feasible for solving large nonlinear system of equations (like deep equilibrium models).

3.1.2 Anderson Acceleration

(**todo**) verify the equations once

Anderson acceleration [Anderson, 1965] is an iterative method for solving steady state problems of the form $f(x) = x$ that uses a combination of previous iterates to improve the convergence rate of fixed-point iterations. Define the residual $g(x) = f(x) - x$. For notational brevity let, $f^{(k)} = f\left(x^{(k)}\right)$ and $g^{(k)} = g\left(x^{(k)}\right)$. The basic idea of Anderson acceleration is to construct a sequence $\left\{x^{(0)}, x^{(1)}, x^{(2)}, \ldots, x^{(K)}\right\}$ to accelerate the convergence of a fixed-point sequence. Given an initial guess $x^{(0)}$ and an integer mixing parameter $m \geq 1$, this method performs the following for each iteration k:

1. Compute $m^{(k)} \leftarrow \texttt{min}\,(m, k)$
2. Let, $G^{(k)} \leftarrow \left[g^{(k-m^{(k)})} \ldots g^{(k)}\right]$
3. Let, $A^{(k)} \leftarrow \left\{\alpha = \left(\alpha_0, \ldots, \alpha_{m^{(k)}}\right) \in \mathbb{R}^{m^{(k)}+1} : \sum_{i=0}^{m^{(k)}} \alpha_i = 1\right\}$
4. $\alpha^{(k)} \leftarrow \underset{\alpha \in A^{(k)}}{\texttt{argmin}} \left\|G^{(k)}\alpha\right\|_2$
5. $x^{(k+1)} \leftarrow \sum_{i=0}^{m^{(k)}} \alpha_i^{(k)} f^{(k-m^{(k)}+i)}$

Anderson acceleration can be very effective for accelerating the convergence of fixed-point iterations for non-linear problems, especially for problems with slow convergence or oscillatory behavior.

3.1.3 Limited Memory Broyden's Method

As described in Section 3.1.1, we can avoid the computational complexity of inverting a Jacobian Matrix by using Broyden's method. However, as pointed out in Bai et al. [2020], even storing the Broyden matrix B for a Nonlinear function $g_\theta : R^{32\times32\times80} \mapsto R^{32\times32\times80}$ requires nearly $25GB$ of storage. To circumvent this issue Bai et al. [2020] propose a limited memory variant of Broyden's method. The idea is to write the low rank approximation matrix of the inverted jacobian $J_{g_\theta}^{-1}(B)$ as the sum of low rank updates:

$$B^{(i+1)} = B^{(0)} + \sum_{k=i}^{i+1} \mathrm{u}^{(k)} \mathrm{v}^{(k)^T} \tag{3.15}$$

$$B^{(i+1)} = B^{(0)} + UV^T \tag{3.16}$$

$$\texttt{where}\quad B^{(0)} = -I \tag{3.17}$$

u and v come from Sherman-Morrison Formula [Sherman and Morrison, 1950]. In the limited memory version, Bai et al. [2020] store the last m low-rank updates for u and v, and use a first-in-first-out approach to update U and V.

3.2 Jacobian Free Newton-Krylov Methods (JNFK) for solving Linear Systems

JFNK Methods are used to solve Linear System of Equations without actually computing the Jacobian Matrix. These methods require the ability to compute matrix-vector products. As we will observe in Section 3.3, ability to avoid the computation of the Jacobian matrix is crucial for large scale steady state problems. JNFK methods use Krylov subspace K_j of dimension k to solve linear equations of the form $Ax = b$ where $A \in \mathbb{R}^{n\times n}$ is a sparse,

non-symmetric matrix, $b \in \mathbb{R}^n$ is a known vector, and $x \in \mathbb{R}^n$ is the solution vector.

$$K_j = \texttt{span}\left(r_0, Ar_0, A^2 r_0, \ldots, A^{j-1} r_0\right) \tag{3.18}$$

$$\texttt{where} \quad r_0 = b - Ax_0 \tag{3.19}$$

3.3 Adjoint Equations

In Section 2, we derived the following linear system of equations:

$$\left(I - \frac{\partial f_\theta\left(z^*\right)}{\partial z^*}\right)^T \times \lambda = \left(\frac{\partial g\left(z^*, \theta\right)}{\partial z^*}\right)^T \tag{3.20}$$

For DEQs, the state space is too large to compute the entire jacobian matrix $\frac{\partial f_\theta(z^*)}{\partial z^*}$. Instead of computing $A = \left(I - \frac{\partial f_\theta(z^*)}{\partial z^*}\right)^T$, we can use Matrix-Free Methods discussed in Section 3.2 to solve Equation 3.10. To use JFNK solvers we need to be efficiently compute:

$$A \times \lambda = \lambda - \left(\frac{\partial f_\theta\left(z^*\right)}{\partial z^*}\right)^T \times \lambda \tag{3.21}$$

$$= \lambda - \left(\lambda^T \times \frac{\partial f_\theta\left(z^*\right)}{\partial z^*}\right)^T \tag{3.22}$$

The second term is the Vector-Jacobian Product (VJP) which can be efficiently computed by any reverse-mode automatic differentiation framework (without constructing the entire Jacobian). Additionally, in Equation 3.11 we can compute $\lambda^T \times \frac{\partial f_\theta(z^*)}{\partial \theta}$ using the VJP trick using any reverse-mode automatic differentiation framework.

4 Common Applications of DEQs

5 Accelerating DEQs

DEQs share the benefits of implicit neural networks in reducing the memory requirements for training. Specifically, DEQs reduce the memory complexity from $O\left(SL\right)$ where S is the dimensions of the output and L is the number of layers to $O\left(S\right)$. However, a major concern is the high cost of forward pass which requires solving a steady state problem. An expensive forward pass is not only a bottleneck for training but also hinders deployment, that rely on fast inference. In this section, we discuss some prior works that accelerate the training and inference of DEQs.

5.1 Jacobian Regularization of DEQs

Bai et al. [2021b]

5.2 Jacobian-Free Backpropagation

Fung et al. [2022]

5.3 Neural Deep Equilibrium Solvers

Bai et al. [2021a]

Part II

ACCELERATING NEURAL DIFFERENTIAL EQUATIONS

CHAPTER 4

Continuous Deep Equilibrium Networks: Training Neural ODEs Faster by Integrating Them to Infinity

Implicit layer methods, such as Neural ODEs and Deep Equilibrium models [Chen et al., 2018, Bai et al., 2019, Ghaoui et al., 2020], have gained popularity due to their ability to automatically adapt model depth based on the "complexity" of new problems and inputs. The forward pass of these methods involves solving steady-state problems, convex optimization problems, differential equations, etc., all defined by neural networks, which can be expensive. However, training these more generalized models has empirically been shown to take significantly more time than traditional explicit models such as recurrent neural networks and transformers. *Nothing within the problem's structure requires expensive training methods, so we asked, can we reformulate continuous implicit models so that this is not the case*?

Grathwohl et al. [2018], Dupont et al. [2019], Kelly et al. [2020], Finlay et al. [2020] have identified several problems with training implicit networks. These models grow in complexity as training progresses, and a single forward pass can take over 100 iterations [Kelly et al., 2020] even for simple problems like MNIST. Deep Equilibrium Models [Bai et al., 2019, 2020] have better scaling in the backward pass but are still bottlenecked by slow steady-state convergence. Bai et al. [2021b] quantified several convergence and stability problems with DEQs. They proposed a regularization technique by exploiting the "implicitness" of DEQs to stabilize their training. *We marry the idea of faster backward pass for DEQs and continuous modeling from Neural ODEs to create Infinite Time Neural ODEs which scale significantly better in the backward pass and drastically reduce the training time.*

Our main contributions include[1]:

[1]Our code is publicly available at `https://github.com/SciML/DeepEquilibriumNetworks.jl`

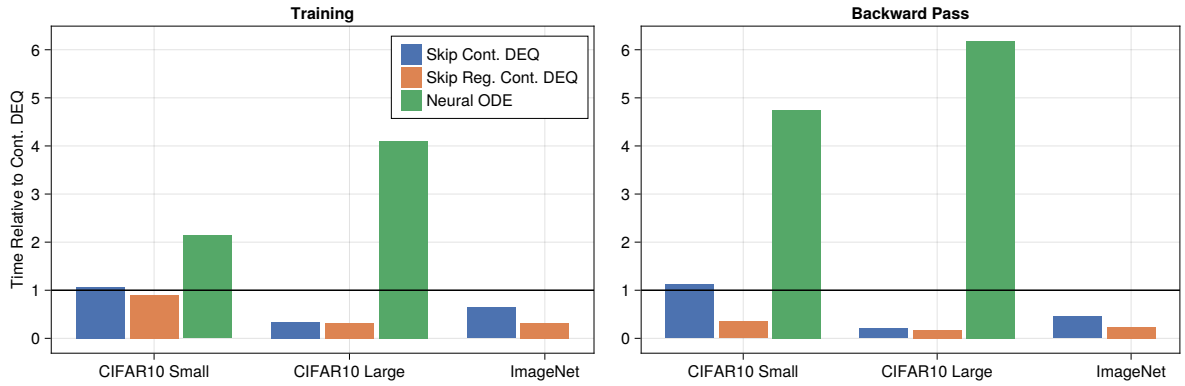

Figure 4.1: Relative Training and Backward Pass Timings against Continuous DEQs *(lower is better)*: In all scenarios, Neural ODEs take $4.7 - 6.182\times$ more time in the backward pass compared to Vanilla Continuous DEQs. Whereas combining Skip (Reg.) with Continuous DEQs accelerates the backward pass by $2.8 - 5.9\times$.

1. An improved DEQ architecture (Skip-DEQ) that uses an additional neural network to predict better initial conditions.
2. A regularization scheme (Skip Regularized DEQ) incentivizes the DEQ to learn simpler dynamics and leads to faster training and prediction. Notably, this does not require nested automatic differentiation and thus is considerably less computationally expensive than other published techniques.
3. A continuous formulation for DEQs as an infinite time neural ODE, which paradoxically accelerates the backward pass over standard neural ODEs by replacing the continuous adjoints with a simple linear system.
4. We demonstrate the seamless combination of Continuous DEQs with Skip DEQs to create a drop-in replacement for Neural ODEs without incurring a high training cost.

The contents of this chapter has appeared in the pre-print: Pal, A., Edelman, A. and Rackauckas, C., 2022. Continuous Deep Equilibrium Models: Training Neural ODEs Faster by Integrating Them to Infinity. arXiv preprint arXiv:2201.12240. [Pal et al., 2022]

1 Continuous Deep Equilibrium Networks

Deep Equilibrium Models have traditionally been formulated as steady-state problems for a discrete dynamical system. However, discrete dynamical systems come with a variety of shortcomings. Consider the following linear discrete dynamical system (See Figure 4.2):

$$u_{n+1} = \alpha \cdot u_n \tag{4.1}$$

$$\texttt{where } \|\alpha\| < 1 \texttt{ and } u_0 = 1 \tag{4.2}$$

This system converges to a steady state of $u_\infty = 0$. However, in many cases, this convergence can be relatively slow. If $\alpha = 0.9$, then after 10 steps, the value is $u_{10} = 0.35$ because a small amount only reduces each successive step. Thus convergence could only be accelerated by taking many steps together. Even further, if $\alpha = -0.9$, the

value ping-pongs over the steady state $u_1 = -0.9$, meaning that if we could take some fractional step $u_{\delta t}$ then it would be possible to approach the steady state much faster.

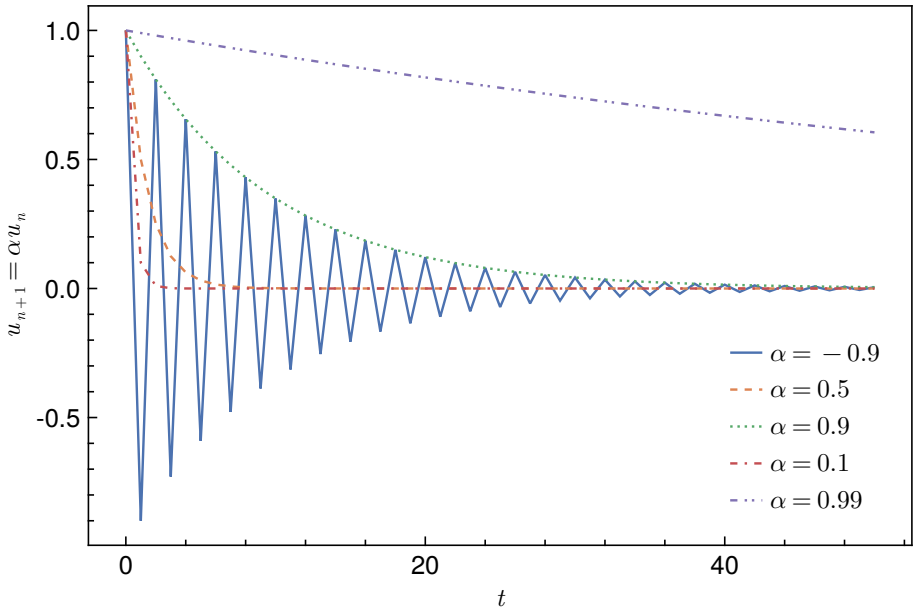

Figure 4.2: Slow Convergence of Simple Linear Discrete Dynamical Systems

Rico-Martinez et al. [1992], Bulsari [1995] describe several other shortcomings of using discrete steady-state dynamics over continuous steady-state dynamics. These issues combined motivate changing from a discrete description of the system (the fixed point or Broyden's method approach) to a continuous description of the system that allows adaptivity to change the stepping behavior and accelerate convergence.

To this end, we propose an alternate formulation for DEQs by modeling a continuous dynamical system (Continuous DEQ) where the forward pass is represented by an ODE which is solved from $t_0 = 0$ to $t_1 = \infty$:

$$\frac{dz}{dt} = f_\theta(z, x) - z \tag{4.3}$$

where f_θ is an explicit neural network. Continuous DEQs leverage fast adaptive ODE solvers, which terminate automatically once the solution is close to a steady state, i.e., $\frac{dz^*}{dt} = 0$, which then satisfies $f_\theta(z^*, x) = z^*$ and is thus the solution to the same implicit system as before.

The Continuous DEQ can be considered an infinite-time neural ODE in this form. However, almost paradoxically, the infinite time version is cheaper to train than the finite time version as its solution is the solution to the nonlinear system, meaning the same implicit differentiation formula of the original DEQ holds for the derivative. This means that no backpropagation through the steps is required for the Continuous DEQ, and only a linear system must be solved. In Section 4, we empirically demonstrate that Continuous DEQs outperform Neural ODEs in terms of training time while achieving similar accuracies.

2 Skip Deep Equilibrium Networks

Bai et al. [2019, 2020] set the initial condition $u_0 = 0$ while solving a DEQ. Assuming the existence of a steady state, the solvers will converge given enough iterations. However, each iteration is expensive, and a poor guess of the initial condition makes the convergence slower. To counteract these issues, we introduce an alternate architecture for DEQ (Skip DEQ), where we use an explicit model g_ϕ to predict the initial condition for the steady-state problem $u_0 = g_\phi(x)$[2]. We jointly optimize for $\{\theta, \phi\}$ by adding an auxiliary loss function:

$$\mathcal{L}_{\texttt{skip}} = \lambda_{\texttt{skip}} \cdot \|f_\theta(z^*, x) - g_\phi(x)\| \tag{4.4}$$

Intuitively, our explicit model g_ϕ better predicts a value closer to the steady-state (over the training iterations), and hence we need to perform fewer iterations during the forward pass. Given that its prediction is relatively free compared to the cost of the DEQ, this technique could decrease the cost of the DEQ by reducing the total number of iterations required. However, this prediction-correction approach still uses the result of the DEQ for its final predictions and thus should achieve robustness properties equal.

3 Skip Regularized DEQ: Regularization Scheme without Extra Parameters

One of the primary benefits of DEQs is the low memory footprint of these models. Introducing an explicit model g_ϕ increases the memory requirements for training. To alleviate this problem, we propose a regularization term to minimize the L1 distance between the first prediction of f_θ and the steady-state solution:

$$\mathcal{L}_{\texttt{skip\_reg}} = \lambda_{\texttt{skip}} \cdot \|f_\theta(z^*, x) - f_\theta(0, x)\| \tag{4.5}$$

This technique follows the same principle as the Skip DEQ where the DEQ's internal neural network is now treated as the prediction model. We hypothesize that this introduces an inductive bias in the model to learn simpler training dynamics.

4 Experiments

In this section, we consider the effectiveness of our proposed methods – Continuous DEQs and Skip DEQs – on the training and prediction timings. We consider the following baselines:

1. Discrete DEQs with L-Broyden Solver.
2. Jacobian Regularization of DEQs.[3]
3. Multi-Scale Neural ODEs with Input Injection: A modified Continuous Multiscale DEQ without the steady state convergence constaint.

Our primary metrics are classification accuracy, the number of function evaluations (NFEs), total training time,

[2]We note that the concurrent work Bai et al. [2021a] introduced a similar formulation as a part of HyperDEQ

[3]We note that due to limitations of our Automatic Differentiation system, we cannot perform Jacobian Regularization for Convolutional Models. However, our preliminary analysis suggests that the Skip DEQ and Continuous DEQ approaches are fully composable with Jacobian Regularization and provide better performance compared to using only Jacobian Regularization (See Table 4.1).

Model	Jacobian Reg.	# of Params	Test Accuracy (%)	Testing NFE	Training Time (min)	Prediction Time (s / batch)
Vanilla DEQ	✗	138K	97.926 ± 0.107	18.345 ± 0.732	5.197 ± 1.106	0.038 ± 0.009
	✓		98.123 ± 0.025	5.034 ± 0.059	7.321 ± 0.454	0.011 ± 0.005
Skip DEQ	✗	151K	97.759 ± 0.080	4.001 ± 0.001	1.711 ± 0.202	0.010 ± 0.001
	✓		97.749 ± 0.141	4.001 ± 0.000	6.019 ± 0.234	0.012 ± 0.001
Skip Reg. DEQ	✗	138K	97.973 ± 0.134	4.001 ± 0.000	1.295 ± 0.222	0.010 ± 0.001
	✓		98.016 ± 0.049	4.001 ± 0.000	5.128 ± 0.241	0.012 ± 0.000

Table 4.1: MNIST Classification with Fully Connected Layers: Skip Reg. Continuous DEQ without Jacobian Regularization takes $4\times$ *less training time* and *speeds up prediction time by* $4\times$ compared to Continuous DEQ. Continuous DEQ with Jacobian Regularization has a similar prediction time but takes $6\times$ *more training time* than Skip Reg. Continuous DEQ. Using Skip variants *speeds up training by* $1.42 \times -4\times$.

time for the backward pass, and prediction time per batch. We showcase the performance of our methods on – MNIST [LeCun et al., 1998], CIFAR-10 [Krizhevsky et al., 2009], SVHN [Netzer et al., 2011], & ImageNet [Deng et al., 2009]. We use perform our experiments in Julia [Bezanson et al., 2017] using Lux.jl [Pal, 2023] and DifferentialEquations.jl [Rackauckas and Nie, 2017a, Rackauckas et al., 2018, 2020].

4.1 MNIST Image Classification

Training Details: Following Kelly et al. [2020], our Fully Connected Model consists of 3 layers – a downsampling layer $\mathbb{R}^{784} \mapsto \mathbb{R}^{128}$, continuous DEQ layer $f_\theta : \mathbb{R}^{128} \mapsto \mathbb{R}^{128}$, and a linear classifier $\mathbb{R}^{128} \mapsto \mathbb{R}^{10}$.

For regularization, we use $\lambda_{\texttt{skip}} = 0.01$ and train the models for 25 epochs with a batch size of 32. We use Tsit5 [Tsitouras, 2011] with a relative tolerance for convergence of 0.005. For optimization, we use Adam [Kingma and Ba, 2014a] with a constant learning rate of 0.001.

Baselines: We use continuous DEQ and continuous DEQ with Jacobian Stabilization as our baselines. We additionally compose Skip DEQs with Jacobian Stabilization in our benchmarks. For all experiments, we keep $\lambda_{\texttt{jac}} = 1.0$.

Results: We summarize our results in Table 4.1. Without Jacobian Stabilization, Skip Reg. Continuous DEQ has the highest testing accuracy of 97.973% and has the *lowest training and prediction timings overall.* Using Jacobian Regularization, DEQ outperforms Skip DEQ models by $< 0.4\%$, however, jacobian regularization increases training time by $1.4 - 4\times$. Skip DEQ models can obtain the lowest prediction time per batch of ~ 0.01s.

4.2 CIFAR10 Image Classification

For all the baselines in this section, Vanilla DEQ is trained with the same training hyperparameters as the corresponding Skip DEQs (taken from Bai et al. [2020]). Multiscale Neural ODE with Input Injection is trained with the same hyperparameters as the corresponding Continuous DEQs.

Model	Continuous	# of Params	Test Accuracy (%)	Training Time (s / batch)	Backward Pass (s / batch)	Prediction Time (s / batch)
Vanilla DEQ	✗	163546	81.233 ± 0.097	0.651 ± 0.009	0.075 ± 0.001	0.282 ± 0.005
	✓		80.807 ± 0.631	0.753 ± 0.017	0.261 ± 0.010	0.136 ± 0.010
Skip DEQ	✗	200122	82.013 ± 0.306	0.717 ± 0.022	0.115 ± 0.004	0.274 ± 0.005
	✓		80.807 ± 0.230	0.806 ± 0.010	0.293 ± 0.004	0.154 ± 0.002
Skip Reg. DEQ	✗	163546	81.170 ± 0.356	0.709 ± 0.005	0.114 ± 0.002	0.283 ± 0.007
	✓		82.513 ± 0.177	0.679 ± 0.015	0.143 ± 0.017	0.154 ± 0.003
Neural ODE	✓	163546	83.543 ± 0.393	1.608 ± 0.026	1.240 ± 0.021	0.207 ± 0.006

Table 4.2: CIFAR10 Classification with Small Neural Network: Skip Reg. Continuous DEQ achieves the *highest test accuracy among DEQs*. Continuous DEQs are faster than Neural ODEs during training by a factor of $2\times - 2.36\times$, with a speedup of $4.2\times - 8.67\times$ in the backward pass. We also observe a prediction speed-up for Continuous DEQs of $1.77\times - 2.07\times$ against Discrete DEQs and $1.34\times - 1.52\times$ against Neural ODE.

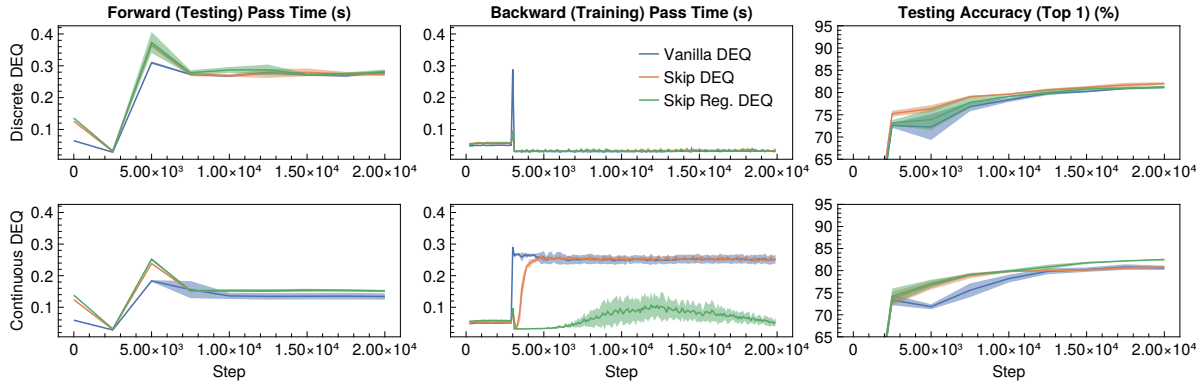

Figure 4.3: CIFAR10 Classification with Small Neural Network

4.2.1 Architecture with 200K parameters

Training Details: Our Multiscale DEQ architecture is the same as MDEQ-small architecture used in Bai et al. [2020]. For the explicit network in Skip DEQ, we use the residual block and downsampling blocks from Bai et al. [2020] which account for the additional 58K trainable parameters.

We use a fixed regularization weight of $\lambda_{\texttt{skip}} = 0.01$ and the models are trained for 20000 steps. We use a batch size of 128. For continuous models, we use VCAB3 [Wanner and Hairer, 1996] with a relative tolerance for convergence of 0.05. We use AdamW [Loshchilov and Hutter, 2017] optimizer with a cosine scheduling on the learning rate – starting from 10^{-3} and terminating at 10^{-6} – and a weight decay of 2.5×10^{-6}.

Results: We summarize our results in Table 4.2 and Figure 4.3. Continuous DEQs are faster than Neural ODEs during training by a factor of $2\times - 2.36\times$, with a speedup of $4.2\times - 8.67\times$ in the backward pass.

Model	Continuous	# of Params	Test Accuracy (%)	Training Time (s / batch)	Backward Pass (s / batch)	Prediction Time (s / batch)
Vanilla DEQ	✗	10.63M	88.913 ± 0.287	0.625 ± 0.165	0.111 ± 0.021	0.414 ± 0.222
	✓		89.367 ± 0.832	1.284 ± 0.011	0.739 ± 0.003	0.606 ± 0.010
Skip DEQ	✗	11.19M	88.783 ± 0.178	0.588 ± 0.042	0.112 ± 0.006	0.314 ± 0.017
	✓		89.600 ± 0.947	0.697 ± 0.012	0.150 ± 0.013	0.625 ± 0.004
Skip Reg. DEQ	✗	10.63M	88.773 ± 0.115	0.613 ± 0.048	0.109 ± 0.008	0.268 ± 0.031
	✓		90.107 ± 0.837	0.660 ± 0.019	0.125 ± 0.003	0.634 ± 0.019
Neural ODE	✓	10.63M	89.047 ± 0.116	5.267 ± 0.078	4.569 ± 0.077	0.573 ± 0.010

Table 4.3: CIFAR10 Classification with Large Neural Network: Skip Reg. Continuous DEQ achieves the *highest test accuracy*. Continuous DEQs are faster than Neural ODEs during training by a factor of $4.1\times - 7.98\times$, with a speedup of $6.18\times - 36.552\times$ in the backward pass. However, we observe a prediction slowdown for Continuous DEQs of $1.4\times - 2.36\times$ against Discrete DEQs and $0.90\times - 0.95\times$ against Neural ODE.

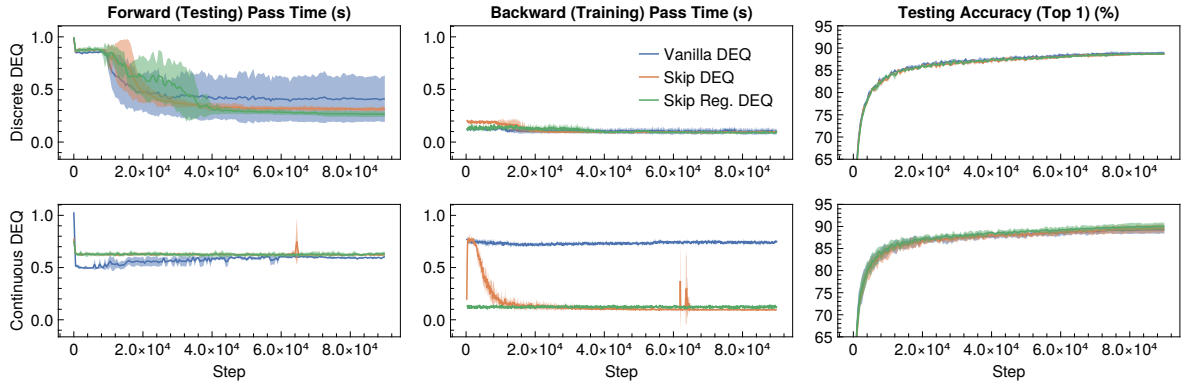

Figure 4.4: CIFAR10 Classification with Large Neural Network

4.2.2 Architecture with 11M parameters

Training Details: Our Multiscale DEQ architecture is the same as MDEQ-large architecture used in Bai et al. [2020]. For the explicit network in Skip DEQ, we use the residual block and downsampling blocks from Bai et al. [2020] which account for the additional 58K trainable parameters.

We use a fixed regularization weight of $\lambda_{\texttt{skip}} = 0.01$ and the models are trained for 90000 steps. We use a batch size of 128. For continuous models, we use VCAB3 [Wanner and Hairer, 1996] with a relative tolerance for convergence of 0.05. We use Adam [Kingma and Ba, 2014a] optimizer with a cosine scheduling on the learning rate – starting from 10^{-3} and terminating at 10^{-6}.

Results: We summarize our results in Table 4.3 and Figure 4.4. Continuous DEQs are faster than Neural ODEs during training by a factor of $4.1\times - 7.98\times$, with a speedup of $6.18\times - 36.552\times$ in the backward pass.

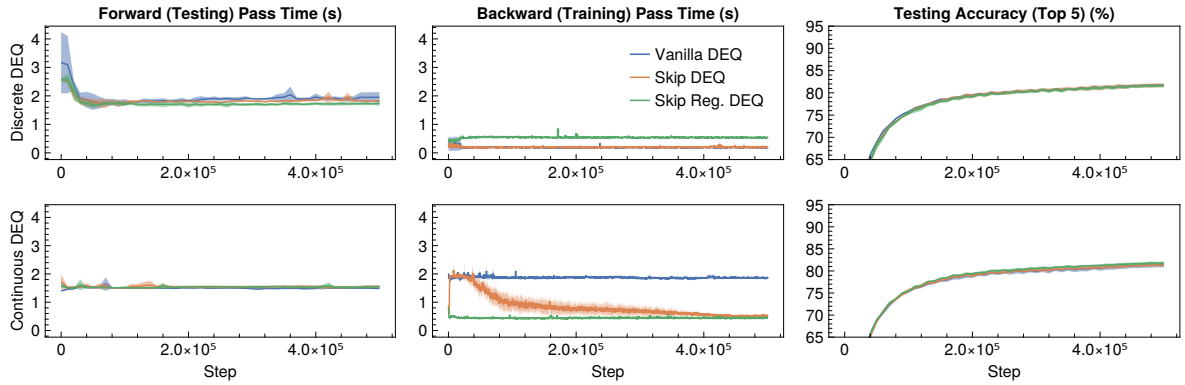

Figure 4.5: ImageNet Classification

Model	Continuous	# of Params	Test Accuracy (Top 5) (%)	Training Time (s / batch)	Backward Pass (s / batch)	Prediction Time (s / batch)
Vanilla DEQ	✗	17.91M	81.809 ± 0.115	2.057 ± 0.138	0.195 ± 0.007	1.963 ± 0.189
	✓		81.329 ± 0.516	3.131 ± 0.027	1.873 ± 0.015	1.506 ± 0.027
Skip DEQ	✗	18.47M	81.717 ± 0.452	1.956 ± 0.012	0.194 ± 0.001	1.843 ± 0.025
	✓		81.334 ± 0.322	2.016 ± 0.129	0.845 ± 0.127	1.575 ± 0.053
Skip Reg. DEQ	✗	17.91M	81.611 ± 0.369	1.996 ± 0.035	0.539 ± 0.023	1.752 ± 0.093
	✓		81.813 ± 0.350	1.607 ± 0.044	0.444 ± 0.026	1.560 ± 0.021

Table 4.4: ImageNet Classification: All the variants attain comparable evaluation accuracies. Skip (Reg.) accelerates the training of Continuous DEQ by $1.57\times - 1.96\times$, with a reduction of $2.2\times - 4.2\times$ in the backward pass timings. However, we observe a marginal increase of 4% in prediction timings for Skip (Reg.) Continuous DEQ compared against Continuous DEQ. For Discrete DEQs, Skip (Reg.) variants reduce the prediction timings by $6.5\% - 12\%$.

4.3 ImageNet Image Classification

Training Details: Our Multiscale DEQ architecture is the same as MDEQ-small architecture used in Bai et al. [2020]. For the explicit network in Skip DEQ, we use the residual block and downsampling blocks from Bai et al. [2020] which account for the additional 58K trainable parameters.

We use a fixed regularization weight of $\lambda_{\texttt{skip}} = 0.01$, and the models are trained for 500000 steps. We use a batch size of 64. For continuous models, we use VCAB3 [Wanner and Hairer, 1996] with a relative tolerance for convergence of 0.05. We use SGD with a momentum of 0.9 and weight decay of 10^{-6}. We use a step LR scheduling reducing the learning rate from 0.05 by a multiplicative factor of 0.1 at steps 100000, 150000, and 250000.

Baselines: Vanilla DEQ is trained with the same training hyperparameters as the corresponding Skip DEQs

(taken from [Bai et al., 2020])[4].

Results: We summarize our results in Table 4.4 and Figure 4.5. Skip (Reg.) variants accelerate the training of Continuous DEQ by $1.57\times - 1.96\times$, with a reduction of $2.2\times - 4.2\times$ in the backward pass timings.

5 Discussion

We have empirically shown the effectiveness of Continuous DEQs as a faster alternative for Neural ODEs. Consistent with the ablation studies in Bai et al. [2021a], we see that Skip DEQ in itself doesn't significantly improve the prediction or training timings for Discrete DEQs. Skip Reg. DEQ does, however, speeds up the inference for larger Discrete DEQs. However, combining Skip DEQ and Skip Reg. DEQ with Continuous DEQs, enable a speedup in backward pass by over $2.8 - 5.9\times$. We hypothesize that this improvement is due to reduction in the condition number, which results in faster convergence of GMRES in the backward pass, however, acertaining this would require furthur investigation. We have demonstrated that our improvements to DEQs and Neural ODEs enable the drop-in replacement of Skip Continuous DEQs in any classical deep learning problem where continuous implicit models were previously employed.

5.1 Limitations

We observe the following limitations for our proposed methods:

- Reformulating a Neural ODE as a Continuous DEQ is valid, when the actual dynamics of the system doesn't matter. This holds true for all applications of Neural ODEs to classical Deep Learning problems.
- Continuous DEQs are slower than their Discrete counterparts for larger models (without any significant improvement to accuracy), hence the authors recommend their usage only for cases where a continuous model is truly needed.

[4]When training MultiScale Neural ODE with the same configuration as Continuous DEQ, we observed a $8\times$ slower backward pass which made the training of the baseline infeasible.

CHAPTER 5

Opening the Blackbox: Global Regularization using Local Error & Stiffness Estimates

How many hidden layers should you choose in your recurrent neural network? Chen et al. [2018] showed that the answer could be found automatically by using a continuous reformulation, the neural ordinary differential equation, and allowing an adaptive ODE solver to effectively choose the number of steps to take. Since then the idea was generalized to other domains such as stochastic differential equations [Liu et al., 2019, Rackauckas et al., 2020] but one fact remained: *solving a neural differential equation is expensive, & training a neural differential equation is even more so.*

In this thesis, we present a generally applicable method to force the neural differential equation training process to choose the least expensive option. We open the blackbox and show how using the numerical heuristics baked inside of these sophisticated differential equation solver codes allows for identifying the cheapest equations without requiring extra computation. However, "opening the blackbox" has several downsides – they are *harder to integrate into existing code-bases* and are *more memory intensive.* Hence, we describe methods that exploit random sampling to leverage the benefits of "opening the blackbox" without actually requiring specialized training methods and the associated memory overhead.

Our main contributions include:

- We introduce a novel regularization scheme for neural differential equations based on the local error estimates and stiffness estimates. We observe that by white-boxing differential equation solvers to leverage pre-computed statistics about the neural differential equations, we can obtain faster training and prediction time while having a minimal effect on testing metrics.
- We compare our method with various regularization schemes [Kelly et al., 2020, Behl et al., 2020], which often use higher order derivatives and are difficult to incorporate within existing systems. We empirically

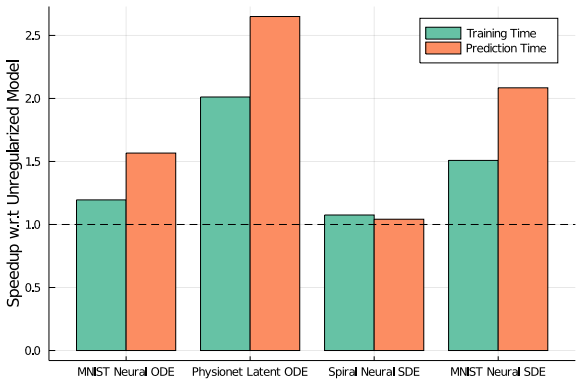

Figure 5.1: Training and Prediction Performance of Regularized NDEs We obtain an average training and prediction speedup of 1.45x and 1.84x respectively for our best model on supervised classification and time series problems.

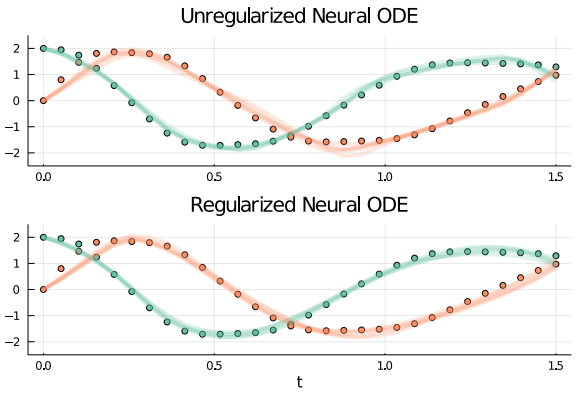

Figure 5.2: Error and Stiffness Regularization Keeps Accuracy. We show the fits of the unregularized/regularized Neural ODE variants on the Sprial equation. However, the unregularized variant requires 1083.0 ± 57.55 NFEs while the one regularized using the stiffness and error estimates requires only 676.2 ± 68.20 NFEs, reducing prediction time by nearly 50%.

show that regularization using cheap statistics can lead to as efficient predictions as the ones requiring higher order automatic differentiation [Kelly et al., 2020, Finlay et al., 2020] without the increased training time.

- We release our code[1], implemented using the Julia Programming Language [Bezanson et al., 2017] and SciML Software Suite [Rackauckas et al., 2019], with the intention of wider adoption of the proposed methods in the community.

The contents of this chapter has appeared previously in the publication: Pal, A., Ma, Y., Shah, V. and Rackauckas, C.V., 2021, July. Opening the Blackbox: Accelerating Neural Differential Equations by Regularizing Internal Solver Heuristics. In International Conference on Machine Learning (pp. 8325-8335). PMLR. [Pal et al., 2021]

1 Opening the Blackbox: Global Regularization using Local Error & Stiffness Estimates

Section 3 describes how larger local error estimates $\mathsf{E}_{\mathtt{Est}}$ lead to reduced step sizes and thus a higher overall cost in the neural ODE training and predictions. Given this, we propose regularizing the neural ODE training process by the total local error in order to learn neural ODEs with as large step sizes as possible. Thus we define the regularizing term:

$$(\mathcal{R}_E)_g = \sum_j (\mathsf{E}_{\mathtt{Est}})_j \cdot |\mathrm{d}t_j| \tag{5.1}$$

summing over j the time steps of the solution. This was done by accumulating the $(\mathsf{E}_{\mathtt{Est}})_j$ from the internals of the time stepping process at the end of each step. We note that this is similar to the regularization proposed in

[1] https://github.com/avik-pal/RegNeuralDE.jl

Kelly et al. [2020], namely:

$$(\mathcal{R}_K)_g = \int_{t_0}^{t_1} \left\| \frac{d^K z(t)}{dt^K} \right\| \mathrm{d}t \tag{5.2}$$

where integrating over the K^{th} derivatives is proportional to the principle (largest) truncation error term of the Runge-Kutta method [Hairer et al., 1993]. However, this formulation requires high order automatic differentiation (which then is layered with reverse-mode automatic differentiation) which can be an expensive computation [Zhang et al., 2008] while Equation 5.1 requires no differentiation.

Similarly, the stiffness estimates (Section 4) at each step can be summed as:

$$(\mathcal{R}_S)_g = \sum_j (\mathsf{S_{Est}})_j \cdot |\mathrm{d}t_j| \tag{5.3}$$

giving a computational heuristic for the total stiffness of the equation. Notably, both of these estimates $(\mathsf{E_{Est}})_j$ and $(\mathsf{S_{Est}})_j$ are already computed during the course of a standard explicit Runge-Kutta solution, making the forward pass calculation of the regularization term computationally free.

2 Adjoints of Internal Solver Heuristics

Notice that $(\mathsf{E_{Est}})_j = \sum_{i=1}^{s} \left(b_i - \tilde{b}_i\right) \cdot k_i$ cannot be constructed directly from the $z\left(t_j\right)$ trajectory of the ODE's solution. More precisely, the k_i terms are not defined by the continuous ODE but instead by the chosen steps of the solver method. Continuous adjoint methods for neural ODEs [Chen et al., 2018, Zhuang et al., 2021] only define derivatives in terms of the ODE quantities. This is required in order exploit properties such as allowing different steps in reverse and reversibility for reduced memory, and in constructing solvers requiring fewer NFEs [Kidger et al., 2021]. Indeed, computing the adjoint of each stage variable k_i can be done, but is known as discrete sensitivity analysis and is known to be equivalent to automatic differentiation of the solver [Zhang and Sandu, 2014]. Thus to calculate the derivative of the solution simultaneously to the derivatives of the solver states, we used direct automatic differentiation of the differential equation solvers for performing the experiments [Innes, 2018a]. We note that discrete adjoints are known to be more stable than continuous adjoints [Zhang and Sandu, 2014] and in the context of neural ODEs have been shown to stabilize the training process leading to better fits [Gholami et al., 2019, Onken and Ruthotto, 2020]. While more memory intensive than some forms of the continuous adjoint, we note that checkpointing methods can be used to reduce the peak memory [Dauvergne and Hascoët, 2006]. We note that this is equivalent to backpropagation of a fixed time step discretization if the step sizes are chosen in advance, and verify in the example code that no additional overhead is introduced.

3 Experimental Results

In this section, we consider the effectiveness of regularizing Neural Differential Equations (NDEs) on their training and prediction timings. We consider the following baselines while evaluating our models:

1. **Vanilla Neural (O/S)DE** with discrete sensitivities.
2. **STEER**: Temporal Regularization for Neural ODE models by stochastic sampling of the end time during training [Behl et al., 2020].

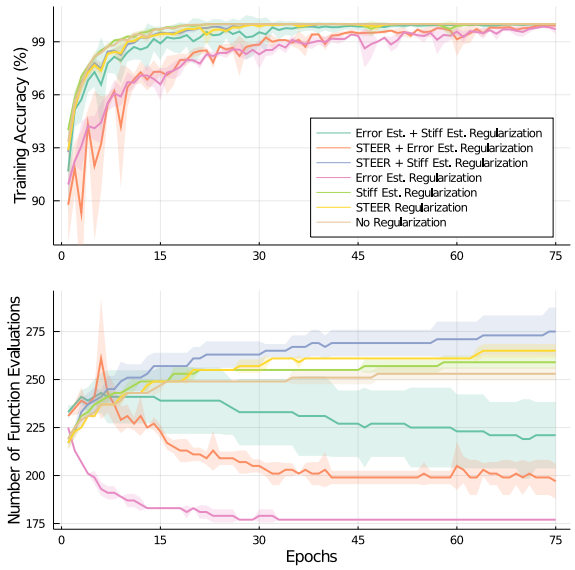

Figure 5.3: Number of Function Evaluations and Training Accuracy for Supervised MNIST Classification Regularizing using ERNODE is the most consistent way to reduce the overall number of function evaluations. Using SRNODE alongside ERNODE stabilizes the training at the cost of increased prediction time.

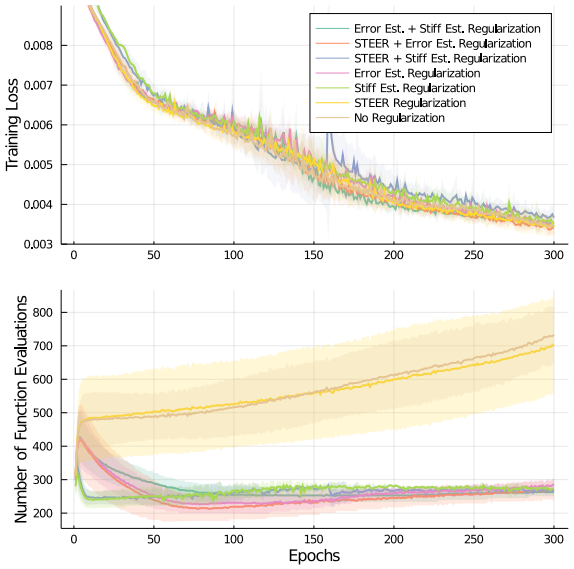

Figure 5.4: Number of Function Evaluations and Training Loss for Physionet Time Series Interpolation Regularized and Unregularized variants of the model have very similar trajectories for the training loss. We do notice a significant difference in the NFE plot. Using either Error Estimate Regularization or Stiffness Regularization is able to bound the NFE to < 300, compared to ~ 700 for STEER or unregularized Latent ODE.

3. **TayNODE**: Regularizing the K^{th} order derivatives of the Neural ODEs [Kelly et al., 2020][2].

We test our regularization on four tasks – supervised image classification (Section 3.1.1) and time series interpolation (Section 3.1.2) using Neural ODE, and fitting Neural SDE (Section 3.2.1) and supervised image classification using Neural SDE (Section 3.2.2). We use DiffEqFlux [Rackauckas et al., 2019] and Flux [Innes et al., 2018] for our experiments.

3.1 Neural Ordinary Differential Equations

In the following experiments, we use a Runge Kutta 5(4) solver [Tsitouras, 2011] with absolute and relative tolerances of 1.4×10^{-8} to solve the ODEs. To measure the prediction time, we use a test batch size equal to the training batch size.

Method	Train Accuracy (%)	Test Accuracy (%)	Train Time (hr)	Prediction Time (s)	NFE
Vanilla NODE	100.0 ± 0.00	97.94 ± 0.02	0.98 ± 0.03	0.094 ± 0.010	253.0 ± 3.46
STEER	100.0 ± 0.00	97.94 ± 0.03	1.31 ± 0.07	0.092 ± 0.002	265.0 ± 3.46
TayNODE	98.98 ± 0.06	97.89 ± 0.00	1.19 ± 0.07	0.079 ± 0.007	080.3 ± 0.43
SRNODE (Ours)	100.0 ± 0.00	98.08 ± 0.15	1.24 ± 0.06	0.094 ± 0.003	259.0 ± 3.46
ERNODE (Ours)	99.71 ± 0.28	97.32 ± 0.06	0.82 ± 0.02	0.060 ± 0.001	177.0 ± 0.00
STEER + *SRNODE*	100.0 ± 0.00	97.88 ± 0.06	1.55 ± 0.27	0.101 ± 0.009	275.0 ± 12.5
STEER + *ERNODE*	99.91 ± 0.02	97.61 ± 0.11	1.37 ± 0.11	0.086 ± 0.018	197.0 ± 9.17
SRNODE + ERNODE	99.98 ± 0.03	97.77 ± 0.05	1.37 ± 0.04	0.081 ± 0.006	221.0 ± 17.3

Table 5.1: MNIST Image Classification using Neural ODE Using ERNODE obtains a training and prediction speedup of 16.33% and 37.78% respectively, at only 0.6% reduced prediction accuracy. SRNODE doesn't help in isolation but is effective when combined with ERNODE to reduce the prediction time by 14.44% while incurring a reduced test accuracy of only 0.17%.

3.1.1 Supervised Classification

Training Details We train a Neural ODE and a Linear Classifier to map flattened MNIST Images to their corresponding labels. Our model uses a two layered neural network f_{θ_1}, as the ODE dynamics, followed by a linear classifier g_{θ_2}, identical to the architecture used in Kelly et al. [2020].

$$z_{\theta_1}(x,t) = \tanh(W_1[x;t] + B_1) \tag{5.4}$$

$$f_{\theta_1}(x,t) = \tanh(W_2[z_{\theta_1}(x,t);t] + B_2) \tag{5.5}$$

$$g_{\theta_2}(x,t) = \sigma(W_3 x + B_3) \tag{5.6}$$

where the parameters $W_1 \in \mathbb{R}^{100\times785}$, $B_1 \in \mathbb{R}^{100}$, $W_2 \in \mathbb{R}^{784\times101}$, $B_2 \in \mathbb{R}^{784}$, $W_3 \in \mathbb{R}^{10\times784}$, and $B_3 \in \mathbb{R}^{10}$. We use a batch size of 512 and train the model for 75 epochs using Momentum [Qian, 1999] with learning rate of 0.1 and mass of 0.9, and a learning rate inverse decay of 10^{-5} per iteration. For Error Estimate Regularization, we perform exponential annealing of the regularization coefficient from 100.0 to 10.0 over 75 epochs. For Stiffness Regularization, we use a constant coefficient of 0.0285.

Baselines For the STEER baseline, we train the models by stochastically sampling the end time point from $\mathcal{U}(T-b, T+b)$ where $T = 1.0$ and $b = 0.5$[3]. We observe no training improvement but there is a minor improvement in prediction time. For the TayNODE baseline, we train the model with a reduced batch size of

[2]We use the original code formulation of the TayNODE in order to ensure usage of the specially-optimized Taylor-mode automatic differentiation technique [Bettencourt et al., 2019] in the training process. Given the large size of the neural networks, most of the compute time lies in optimized BLAS kernels which are the same in both implementations, meaning we do not suspect library to be a major factor in timing differences beyond the AD specifics.

[3]$b = 0.25$ was also considered but final results were comparable

Method	Train Loss ($\times 10^{-3}$)	Test Loss ($\times 10^{-3}$)	Train Time (hr)	Prediction Time (s)	NFE
Vanilla NODE	3.48 ± 0.00	3.55 ± 0.00	1.75 ± 0.39	0.53 ± 0.12	733.0 ± 84.29
STEER	3.43 ± 0.02	3.48 ± 0.01	1.62 ± 0.26	0.54 ± 0.06	699.0 ± 141.1
TayNODE	4.21 ± 0.02	4.21 ± 0.01	12.3 ± 0.32	0.22 ± 0.02	167.3 ± 11.93
SRNODE (Ours)	3.52 ± 1.44	3.58 ± 0.05	0.87 ± 0.09	0.20 ± 0.01	273.0 ± 0.000
ERNODE (Ours)	3.51 ± 0.00	3.57 ± 0.00	0.94 ± 0.13	0.21 ± 0.02	287.0 ± 17.32
STEER + *SRNODE*	3.67 ± 0.02	3.73 ± 0.02	0.89 ± 0.08	0.20 ± 0.01	271.0 ± 12.49
STEER + *ERNODE*	3.41 ± 0.02	3.48 ± 0.01	1.03 ± 0.25	0.24 ± 0.05	269.0 ± 33.05
SRNODE + *ERNODE*	3.48 ± 0.11	3.56 ± 0.03	1.12 ± 0.08	0.21 ± 0.01	263.0 ± 12.49

Table 5.2: Physionet Time Series Interpolation All the regularized variants of Latent ODE (except STEER) have comparable prediction times. Additionally, the training time is reduced by $36\% - 50\%$ on using one of our proposed regularizers, while TayNODE increases the training time by 7x. Overall, SRNODE has the best training and prediction timings while incurring an increased 0.85% test loss.

100[4], $\lambda = 3.02 \times 10^{-3}$, and regularizing 3^{rd} order derivatives.

Results Figure 5.3 visualizes the training accuracy and number of function evaluations over training. Table 5.1 summarizes the metrics from the trained baseline and proposed models – Error Estimate Regularized Neural ODE (*ERNODE*) and Stiffness Regularized Neural ODE (*SRNODE*). Additionally, we perform ablation studies by composing various regularization strategies.

3.1.2 Time Series Interpolation

Training Details We use the Latent ODE [Chen et al., 2018] model with RNN encoder to learn the trajectories for ICU Patients for Physionet Challenge 2012 Dataset [Silva et al., 2012]. We use the preprocessed data provided by Kelly et al. [2020] to ensure consistency in results. For every independent run, we perform an $80:20$ split of the data for training and evaluation.

Our model architecture is similar to the encoder-decoder models used in Rubanova et al. [2019]. We use a 20-dimensional latent state and a 40-dimensional hidden state for the recognition model. Our ODE dynamics is given by a 4-layered neural network with 50 units and tanh activation. We train our models for 300 epochs with a batchsize of 512 and using Adamax [Kingma and Ba, 2014b] with a learning rate of 0.01 and an inverse decay of 10^{-5}. We minimize the negative log likelihood of the predictions and perform KL annealing with a coefficient of 0.99.

For Error Estimate Regularization, we perform exponential annealing of the regularization coefficient from 1000.0 to 100.0 over 300 epochs. We note that using $(\mathcal{R}_E)_g = \sum_j (\mathtt{E_{Est}})_j^2$, instead of $(\mathcal{R}_E)_g = \sum_j (\mathtt{E_{Est}})_j \cdot |\mathrm{d}t_j|$, yields similar results with a constant regularization coefficient of 100.0. For Stiffness Regularization, we use a constant coefficient of 0.285.

[4]Batch Size was reduced to ensure we reach a comparable train/test accuracy as the other trained models.

Method	Mean Squared Loss	Train Time (s)	Prediction Time (s)	NFE
Vanilla NSDE	0.0217 ± 0.0088	178.95 ± 20.22	0.07553 ± 0.0186	528.67 ± 6.11
SRNSDE (Ours)	0.0204 ± 0.0091	166.42 ± 14.51	0.07250 ± 0.0017	502.00 ± 4.00
ERNSDE (Ours)	0.0227 ± 0.0090	173.43 ± 04.18	0.07552 ± 0.0008	502.00 ± 4.00

Table 5.3: Spiral SDE The ERNSDE attains a relative loss of 4% compared to vanilla Neural SDE while reducing the training time and number of function evaluations. Using SRNSDE reduces both the training and prediction times by 7% and 4% respectively.

Baselines For STEER Baseline, we stochastically sample the timestep to evaluate the difference between interpolated and ground truth data. Essentially for the interval (t_i, t_{i+1}), we evaluate the model at $\mathcal{U}(t_{i+1} - \frac{t_{i+1}-t_i}{2}, t_{i+1} + \frac{t_{i+1}-t_i}{2})$ and compare with the truth at t_{i+1}. We sample end points after every iteration of the model. STEER reduces the training time but has no significant effect on the prediction time. TayNODE was trained by regularizing the 2^{nd} order derivatives and a coefficient of 0.01 for 300 epochs and a batchsize of 512. TayNODE had an exceptionally high training time $\sim 7\times$ compared to the unregularized baseline.

Results Figure 5.4 shows the training MSE loss and the NFE counts for the considered models. Table 5.2 summarizes the metrics and wall clock timings for the baselines, proposed regularizers and their compositions with previously proposed regularizers. We observe that SRNODE provides the most significant speedup while ERNODE attains similar losses at slightly higher training and prediction times.

3.2 Neural Stochastic Differential Equations

In these experiments, we use SOSRI/SOSRI2 [Rackauckas and Nie, 2020] to solve the Neural SDEs. The wall clock timings represent runs on a CPU.

3.2.1 Fitting Spiral Differential Equation

Training Details In this experiment, we consider training a Neural SDE to mimic the dynamics of the Spiral Stochastic Differential Equation with Diagonal Noise (DSDE). Spiral DSDE is prescribed by the following equations:

$$\begin{aligned} du_1 &= -\alpha u_1^3 dt + \beta u_2^3 dt + \gamma u_1 dW \\ du_2 &= -\beta u_1^3 dt - \alpha u_2^3 dt + \gamma u_2 dW \end{aligned} \tag{5.7}$$

where $\alpha = 0.1$, $\beta = 2.0$, and $\gamma = 0.2$. We generate data across 10000 trajectories at 30 uniformly spaced points between $t \in [0, 1]$ (Figure 5.5). We parameterize our drift and diffusion functions using neural networks f_θ and g_ϕ via:

$$\begin{aligned} f_\theta(x, t) &= W_2 \tanh(W_1 x^3 + B_1) + B_2 \\ g_\phi(x, t) &= W_3 x + B_3 \end{aligned} \tag{5.8}$$

where the parameters $W_1 \in \mathbb{R}^{50\times 2}$, $B_1 \in \mathbb{R}^{50}$, $W_2 \in \mathbb{R}^{2\times 50}$, $B_2 \in \mathbb{R}^2$, $W_3 \in \mathbb{R}^{2\times 2}$, and $B_3 \in \mathbb{R}^2$. For fitting the

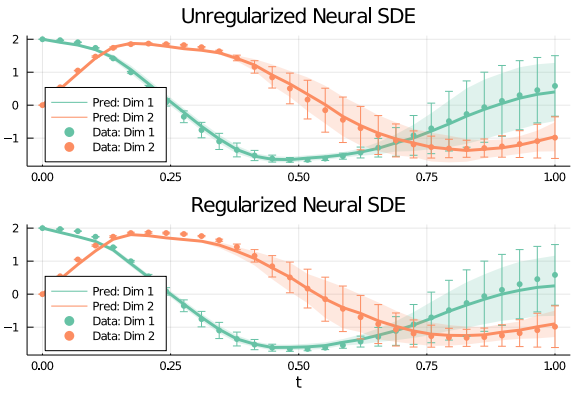

Figure 5.5: Fitting a Neural SDE on Spiral SDE Data. Regularizing has minimal effect on the learned dynamics with reduced training and prediction cost.

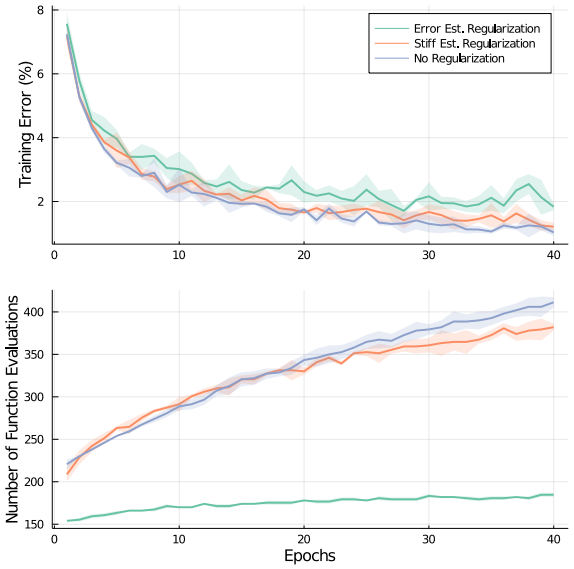

Figure 5.6: Number of Function Evaluations and Training Error for Supervised MNIST Classification using Neural SDE ERNSDE reduces the NFE below 300 with minimal error change while the unregularized version has NFE $\sim$ 400.

drift and diffusion functions to the simulated data, we used a generalized method of moments loss function [Lück and Wolf, 2016, Jeisman, 2006]. Our objective is to train these parameters to minimize the L_2 distance between the mean (μ) and variance (σ^2) of predicted and real data. Let, $\hat{\mu}_i$'s and $\hat{\sigma}_i^2$'s denote the means and variances respectively of the multiple predicted trajectories.

$$\mathcal{L}(u_0; \theta, \phi) = \sum_{i=1}^{30} \left[(\mu_i - \hat{\mu}_i)^2 + (\sigma_i^2 - \hat{\sigma}_i^2)^2 \right] + \lambda_r R_E \tag{5.9}$$

The models were trained using AdaBelief Optimizer [Zhuang et al., 2020] with a learning rate of 0.01 for 250 iterations. We generate 100 trajectories for each iteration to compute the $\hat{\mu}_i$s and $\hat{\sigma}_i^2$s.

Results Table 5.3 summarizes the final results for the trained models for 3 different random seeds. We notice that even for this "toy" problem, we can marginally improve training time while incurring a minimal penalty on the final loss.

3.2.2 Supervised Classification

Training Details We train a Neural SDE model to map flattened MNIST Images to their corresponding labels. Our diffusion function uses a two layered neural network f_{θ_2} and the drift function is a linear map g_{θ_3}. We use two additional linear maps – a_{θ_1} mapping the flattened image to the hidden dimension and b_{θ_4} mapping the

Method	Train Accuracy (%)	Test Accuracy (%)	Train Time (hr)	Prediction Time (s)	NFE
Vanilla NSDE	98.97 ± 0.11	96.95 ± 0.11	6.32 ± 0.19	15.07 ± 0.93	411.33 ± 6.11
SRNSDE (Ours)	98.79 ± 0.12	96.80 ± 0.07	8.54 ± 0.37	14.50 ± 0.40	382.00 ± 4.00
ERNSDE (Ours)	98.16 ± 0.11	96.27 ± 0.35	4.19 ± 0.04	07.23 ± 0.14	184.67 ± 2.31

Table 5.4: MNIST Image Classification using Neural SDE ERNSDE obtains a training and prediction speedup of 33.7% and 52.02% respectively, at only 0.7% reduced prediction accuracy.

output of the Neural SDE to the logits.

$$a_{\theta_1}(x,t) = W_1 x + B_1 \tag{5.10}$$

$$f_{\theta_2}(x,t) = W_3 \ \tanh(W_2\, x + B_2) + B_3 \tag{5.11}$$

$$g_{\theta_3}(x,t) = W_4 \ x + B_4 \tag{5.12}$$

$$b_{\theta_4}(x,t) = W_5 \ x + B_5 \tag{5.13}$$

where the parameters $W_1 \in \mathbb{R}^{32\times 784}$, $B_1 \in \mathbb{R}^{32}$, $W_2 \in \mathbb{R}^{32\times 64}$, $B_2 \in \mathbb{R}^{64}$, $W_3 \in \mathbb{R}^{32\times 64}$, $B_3 \in \mathbb{R}^{32}$, $W_4 \in \mathbb{R}^{10\times 32}$, and $B_3 \in \mathbb{R}^{10}$. We use a batch size of 512 and train the model for 40 epochs using Adam [Kingma and Ba, 2014b] with learning rate of 0.01, and an inverse decay of 10^{-5} per iteration. While making predictions we use the mean logits across 10 trajectories. For Error Estimate and Stiffness Regularization, we use constant coefficients 10.0 and 0.1 respectively.

Results Figure 5.6 shows the variation in NFE and Training Error during training. Table 5.4 summarizes the final metrics and timings for all the trained models. We observe that SRNSDE doesn't improve the training/prediction time, similar to the MNIST Neural ODE Experiment 3.1.1. However, ERNSDE gives us a training and prediction speedup of 33.7% and 52.02% respectively, at the cost of 0.7% reduced test accuracy.

4 Discussion

Numerical analysis has had over a century of theoretical developments leading to efficient adaptive methods for solving many common nonlinear equations such as differential equations. Here we demonstrate that by using the knowledge embedded within the heuristics of these methods we can accelerate the training process of neural ODEs.

We note that on the larger sized PhysioNet and MNIST examples we saw significant speedups while on the smaller differential equation examples we saw only minor performance improvements. This showcases how the NFE becomes a better estimate of the total compute time as the cost of the ODE f (and SDE g) increase when the model size increases.

This result motivates efforts in differentiable programming [Wang et al., 2018, Abadi and Plotkin, 2019, Rackauckas et al.] which enables direct differentiation of solvers since utilizing the solver's heuristics may be crucial in the development of advanced techniques. This idea could be straightforwardly extended not only to other forms of differential equations, but also to other "implicit layer" machine learning methods. For example, Deep Equilibrium Models (DEQ) [Bai et al., 2019] model the system as the solution to an implicit function via a non-

linear solver like Bryoden or Newton's method. Heuristics like the ratio of the residuals have commonly been used as a convergence criterion and as a work estimate for the difficulty of solving a particular nonlinear equation [Wanner and Hairer, 1996], and thus could similarly be used to regularize for learning DEQs whose forward passes are faster to solve. Similarly, optimization techniques such as BFGS [Kelley, 1999] contain internal estimates of the Hessian which can be used to regularize the stiffness of "optimization as layers" machine learning architectures like OptNet [Amos and Kolter, 2017]. However, in these cases we note that continuous adjoint techniques have a significant computational advantage over discrete adjoint methods because the continuous adjoint method can be computed directly at the point of the solution while discrete adjoints would require differentiating through the iteration process. Thus while a similar regularization would exist in these contexts, in the case of differential equations the continuous and discrete adjoints share the same computational complexity which is not the case in methods which iterate to convergence. Further study of these applications would be required in order to ascertain the effectiveness in accelerating the training process, though by extrapolation one may guess that at least the forward pass would be accelerated.

4.1 Limitations

While these experiments have demonstrated major performance improvements, it is pertinent to point out the limitations of the method. One major point to note is that this only applies to learning neural ODEs for maps $z(0) \mapsto z(1)$ as is used in machine learning applications of the architecture [Chen et al., 2018]. Indeed, a neural ODE as an "implicit layer" for predictions in machine learning does not require identification of dynamical mechanisms. However, if the purpose is to learn the true governing dynamics a physical system from timeseries data, this form of regularization would bias the result, dampening higher frequency responses leading to an incorrect system identification. Approaches which embed neural networks into solvers could be used in such cases [Shen et al., 2020, Poli et al., 2020]. Indeed we note that such Hypereuler approaches could be combined with the ERNODE regularization on machine learning prediction problems, which could be a fruitful avenue of research. Lastly, we note that while either the local error and stiffness regularization was effective on each chosen equation, neither was effective on all equations and at this time there does not seem to be a clear a priori indicator as to which regularization is necessary for a given problem. While it seems the error regularization was more effective on the image classification tasks while the stiffness regularization was more effective on the time series task, we believe more experiments will be required in order to ascertain whether this is a common phenomena, possibly worthy of theoretical investigation.

CHAPTER 6

Closing the Blackbox: Local Regularization of Neural DEs using Local Error Estimates

Implicit Models, such as Neural Ordinary Differential Equations Chen et al. [2018] and Deep Equilibrium Models Bai et al. [2019], Pal et al. [2022], have emerged as a promising technique to determine the depth of neural networks automatically. To maximize performance on a dataset, explicit models are tuned to the "hardest" training sample, which hurts the inference timings for "easier" – more abundant – samples. Using adaptive differential equation solvers allow these implicit models to choose the number of steps they need effectively. This idea of representing neural networks as ODEs has since been generalized to Stochastic Differential Equations [Liu et al., 2019, Rackauckas et al., 2020] and other architectures to improve robustness.

Despite the rapid progress in these methods, the core problem of the scalability of these models is still persistent. Several solutions to these have been proposed:

- Kelly et al. [2020], Finlay et al. [2020] use higher order derivatives for regularization.
- Poli et al. [2020] learn neural solvers to solve Neural ODEs faster.
- In Chapter 5, we described a "zero-cost" global regularization scheme.
- Behl et al. [2020] randomize the integration stop time to "smoothen" the dynamics.

All these methods have definite tradeoffs. Finlay et al. [2020], Kelly et al. [2020] have relied on using higher-order regularization terms to constrain the space of learnable dynamics. These models speed up predictions, but their benefits are often overshadowed by a massive training slowdown [Pal et al., 2021]. More recently, quite a few first-order schemes have been proposed. Behl et al. [2020] randomized the endpoint of Neural ODEs to incentivize simpler dynamics. However, Pal et al. [2021] didn't find significant benefits of using STEER in their experiments. Pal et al. [2021] used internal solver heuristics – local error and stiffness estimates – to control the learned dynamics in a way that decreased both prediction and training time.

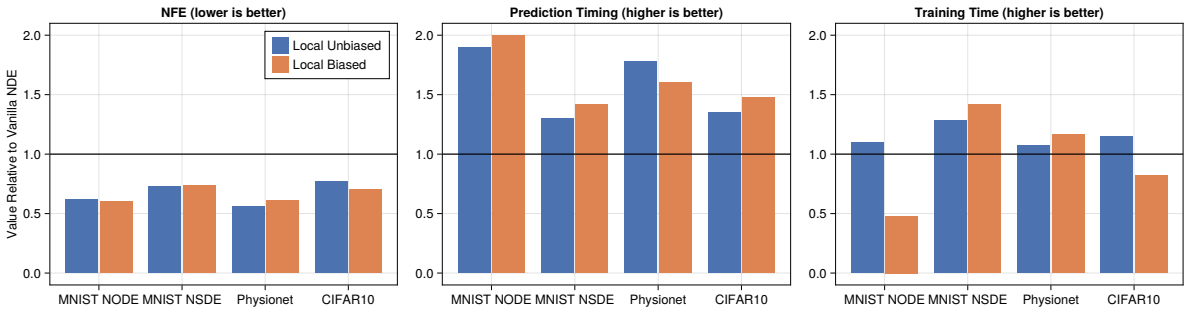

Figure 6.1: Locally Regularized NDE leads to faster predictions and faster training compared to vanilla NDE.

This paper presents a generally applicable method to force the neural differential equation training process to choose the least expensive option. We build upon the global regularization scheme proposed in Pal et al. [2021] and "close" the blackbox allowing our method to work across various sensitivity algorithms. Our main contributions include the following[1]:

1. We show that our local regularization method – building upon the primitives proposed in Pal et al. [2021] – performs at par with global regularization.
2. We present two sampling methods that trade-off small computational costs for consistently better performance.
3. Using local regularization allows our models to leverage optimize-then-discretize in the backward pass (in addition to discretize-then-optimize methods). Our method works around the several engineering limitations of automatic differentiation (AD) systems [Rackauckas, 2022] that are needed to make global regularization work.
4. We empirically show that regularizing solver heuristics with biased sampling stabilizes the training of larger neural ODEs.

The contents of this chapter has appeared previously in the pre-print: Pal, A., Edelman, A. and Rackauckas, C., 2023. Locally Regularized Neural Differential Equations: Some Black Boxes Were Meant to Remain Closed!. arXiv preprint arXiv:2303.02262. [Pal et al., 2023]

1 Randomized Local Regularization: Overcoming the shortcomings of Global Regularization

In Section 4.1, we discussed the downsides of using global regularization with local error estimates. To summarize:

1. Global Regularization relies on discrete sensitivity analysis, which is *more memory intensive.*
2. Global Regularization depends on AD tooling to support dynamic compute graphs in an efficient way,

[1]Our code is publicly available at `https://github.com/avik-pal/LocalRegNeuralDE.jl`

Algorithm 6.1 Unbiased Sampling: Training

function $\texttt{ERNODE}_{\texttt{UNBIASED}}(x, f_\theta, t_{\texttt{span}})$
  $\texttt{Define } \frac{du}{dt} = f_\theta(u, t)$
  $t_0, t_1 \leftarrow t_{span}$
  $\texttt{t}_{\texttt{reg}} \sim \mathbb{U}[t_0,\ t_1]$
  $\texttt{sol} \leftarrow \texttt{solve}(\frac{du}{dt},\ \texttt{DE Solver},\ t_{\texttt{span}})$
  $u_{\texttt{t}_{\texttt{reg}}} \leftarrow \texttt{sol}(\texttt{t}_{\texttt{reg}})$
  Run single step for the solver with time-span $(\texttt{t}_{\texttt{reg}}, t_1)$
  $\texttt{r} \leftarrow$ Local Error Estimate $@t = \texttt{t}_{\texttt{reg}}$
  return sol, r
end function

making it *hard to incorporate into existing code-bases.*

To get around these limitations, we developed a new technique using local sampling of error estimates at specific time points, rather than globally over the full interval. We deal with sampling the "appropriate" time point for regularization by two strategies:

- **Algorithm 6.1 Unbiased Sampling**: We random uniformly sample the time-point in the integration time span. Intuitively, since we will perform the training for "a large number of steps," the learned dynamical system would end up being faster to solve "everywhere" over the time span.
- **Algorithm 6.2 Biased Sampling**: Adaptive Time-Stepping Differential Equation Solvers naturally take more steps around the area, which is harder to integrate. We can bias the regularization to operate around parts of the dynamical system which are "harder" by sampling a time-point from the solution time points.

1.1 Unbiased Sampling of Local Error Estimates

When training a Neural ODE, the integration time-span is fixed. Training any deep learning model involves several thousand steps. We compute the total local error estimate over the entire time-span when performing global regularization. For unbiased sampling, we hypothesize that if we regularize at random uniformly sampled time points in the time-span, the learned dynamical system will demonstrate similar properties in terms of NFE compared to global regularization. Our new regularization term becomes $(\mathcal{R}_E)_{\texttt{unbiased}}$ as:

$$(\mathcal{R}_E)_{\texttt{unbiased}} = (\texttt{E}_{\texttt{Est}})_{\texttt{t}_{\texttt{reg}}} \cdot |\mathrm{d}t_{\texttt{t}_{\texttt{reg}}}| \tag{6.1}$$

$$\texttt{t}_{\texttt{reg}} \sim \mathbb{U}[\texttt{tspan}] \tag{6.2}$$

1.2 Biased Sampling of Local Error Estimates

Algorithm 6.2 Biased Sampling: Training

function $\texttt{ERNODE}_{\texttt{BIASED}}(x, f_\theta, t_{\texttt{span}})$
  $\texttt{Define } \frac{du}{dt} = f_\theta(u, t)$
  $t_0, t_1 \leftarrow t_{span}$
  $\texttt{sol} \leftarrow \texttt{solve}(\frac{du}{dt},\ \texttt{DE Solver},\ t_{\texttt{span}})$
  $\texttt{t}_{\texttt{reg}} \sim \mathbb{U}(\texttt{sol}.t)$
  $u_{\texttt{t}_{\texttt{reg}}} \leftarrow \texttt{sol}(\texttt{t}_{\texttt{reg}})$
  Run single step for the solver with time-span $(\texttt{t}_{\texttt{reg}}, t_1)$
  $\texttt{r} \leftarrow$ Local Error Estimate $@t = \texttt{t}_{\texttt{reg}}$
  return $\texttt{sol}, \texttt{r}$
end function

Consider a simple scenario where the learned dynamics of the DE is harder to solve in $[0.25, 0.35]$, and we are solving the DE from $t_0 = 0$ to $t_1 = 1$. Our primary aim is to modify the learned system s.t. it becomes simpler to solve in $[0.25, 0.35]$. If we use unbiased sampling, the probability that we regularize at $\texttt{t}_{\texttt{reg}} \in [0.25, 0.35]$ is 0.1 (which is low). The problem gets even more severe if the range is lowered. An extreme version of this problem is observed for stiff systems like Robertson's Equations (See Figure 6.2) where 75% of the time is spent in solving 5% of the problem. We note that these extreme scenarios rarely occur for traditional deep learning tasks since Pal et al. [2021] observed minor speedups using stiffness regularization. However, the problem that some parts of the dynamical system are harder to integrate persists, and designing a regularization scheme targeting those parts is highly desirable.

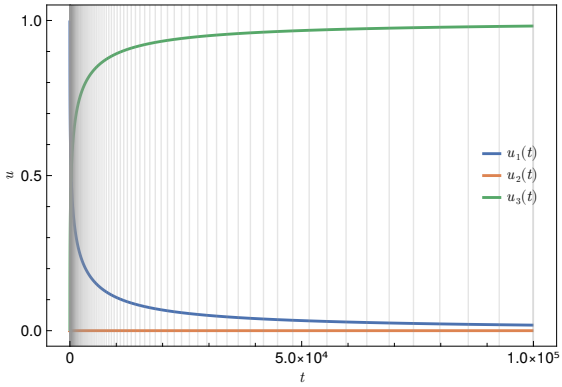

Figure 6.2: Robertson Stiff ODE System: Solving stiff systems like Robertson [Robertson, 1966] (using Rodas5 [Piché, 1995]) involves spending around 75% of the time in $t < 5000$ (i.e. 5% of the time-span). The vertical lines denote the time-points at which the ODE System was solved.

We considered a simple scenario where the learned dynamical system was fixed. However, while training NDEs, this system evolves with training, and apriori predicting the more difficult portions to integrate is not feasible. Adaptive solvers take more frequent steps in the parts of the DE where it is harder to integrate. Anantharaman et al. [2020] leveraged this property of adaptive solvers to learn surrogates for stiff systems. Since these solvers adapt to concentrate around the most numerically difficult time points, we automatically obtain the time points where we want to regularize the model. Hence, for our biased sampling regularize, we uniformly sample the regularization timepoint $\texttt{t}_{\texttt{reg}}$ from the time points at which the solver solved the differential equation.

2 Adjoint for Local Regularized Neural Differential Equations

Adjoint Sensitivity Analysis of Local Regularization works by piggy-backing on the existing adjoint sensitivity analysis algorithms. Our algorithm is effectively equivalent to using the default adjoint sensitivity algorithm with direct reverse mode differentiation through a single step of the solver, i.e. $t = 1$. Thus, our algorithm adds a constant overhead of $O(s \times \text{stages})$ memory to the underlying sensitivity algorithm.

Sensitivity Algorithm	Memory Requirement	Memory Requirement with Local Reularization
Backsolve Adjoint [Chen et al., 2018]	$O(s)$	$O(s \times (1 + \text{stages}))$
Backsolve Adjoint with Checkpointing [Chen et al., 2018]	$O(s \times c)$	$O(s \times (c + \text{stages}))$
Interpolating Adjoint [Hindmarsh et al., 2005]	$O(s \times t)$	$O(s \times (t + \text{stages}))$
Interpolating Adjoint with Checkpointing [Hindmarsh et al., 2005]	$O(s \times c)$	$O(s \times (c + \times\text{stages}))$
Quadrature Adjoint [Kim et al., 2021]	$O((s + p) \times t)$	$O((s + p) \times t + s \times \text{stages})$
Direct Reverse Mode Differentiation	$O(s \times t \times \text{stages})$	$O(s \times (t + 1) \times \text{stages})$

Table 6.1: Memory Requirements for various Sensitivity Algorithms for ODEs with Local Regularization

3 Experimental Results

In this section, we compare the effectiveness of unbiased and biased local regularization's effectiveness on the training and prediction timings of NDEs. We choose image classification and time series prediction problems in line with prior works on accelerating NDEs. We consider the following baselines:

1. Vanilla Neural ODE with Continuous Interpolating Adjoint.
2. Vanilla Neural SDE with discrete sensitivities.
3. Global Regularization of Neural Differential Equations using discrete sensitivity analysis [Pal et al., 2021].
4. TayNODE [Kelly et al., 2020] and STEER [Behl et al., 2020] for models reported in Pal et al. [2021].

We use the DifferentialEquations.jl [Rackauckas et al., 2019] and Lux.jl [Pal, 2023] software stack written in the Julia Programming Language [Bezanson et al., 2017] for all our experiments.

Some details about the data presented in the tables:

- All experimental results in the tables marked with ¶ were taken directly from Pal et al. [2021].
- We have tried to match the hardware details presented in the paper and the corresponding GitHub repository for Pal et al. [2021], but we note that differences in wall clock timings can be partially attributed to hardware.
- TayNODE [Kelly et al., 2020] uses a different ODE integrator. Hence the NFEs are not directly comparable.

3.1 MNIST Image Classification

We train a neural differential equation classifier to map flattened MNIST [LeCun et al., 1998] images to their corresponding labels.

3.1.1 Neural Ordinary Differential Equation

Training Details: We use the same model architecture as described in Kelly et al. [2020]. Our model comprises of single hidden layered explicit model f_θ modeling the ODE dynamics followed by a linear classifier g_ϕ. The hidden layer is 100-dimensional. We train with a batch size of 512 for a total of 7500 steps. We use Adam [Kingma

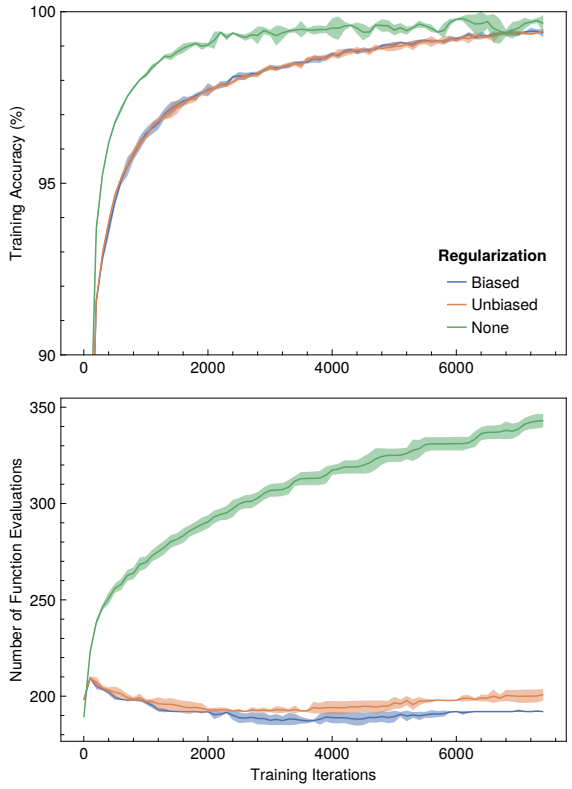

Figure 6.3: MNIST Classification using Neural ODE

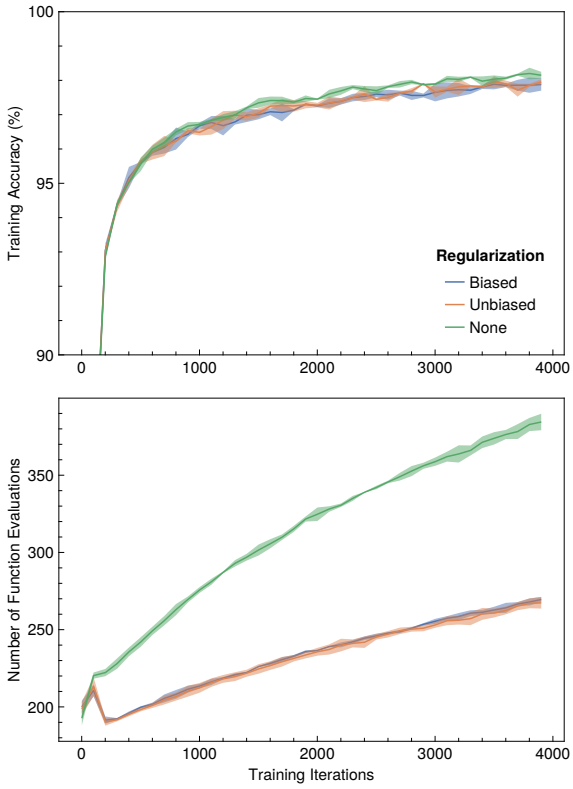

Figure 6.4: MNIST Classification using Neural SDE

and Ba, 2014b] with a constant learning rate of 0.001. For error estimate regularization, we exponentially decrease the regularization coefficient from 2.5 to 1.0. We use Tsit5 [Tsitouras, 2011] as the ODE integrator with an absolute and relative tolerance of 10^{-8}.[2]

Baselines: We consider a Vanilla Neural ODE trained with the exact aforementioned specifications. All other baselines are directly taken from Pal et al. [2021].

Results: We summarize the results in Table 6.2 and Figure 6.3. Using local regularization speeds up prediction in all cases while it leads to a minor slowdown during training for biased sampling.

3.1.2 Neural Stochastic Differential Equation

Training Details: We downsample the flattened images to a 32-dimensional vector before feeding it into the Neural SDE which uses a diffusion model (f_θ) having a 64-dimensional hidden layer and a linear drift model (g_ϕ). Finally, a linear classifier (h_γ) predicts the label. We train our models on CPU with a batch size of 512 for a total of 4000 steps. We optimize the weights using Adam [Kingma and Ba, 2014b] with a constant learning rate of 0.01. We use SOSRI2 SDE solver [Rackauckas and Nie, 2017b] with a tolerance of 0.14. We fix our regularization coefficient to be 10^3. For this experiment, we rely on discrete sensitivity analysis.

[2]We note that this is not a realistic tolerance at which image classification models are trained. We use this tolerance to allow a direct comparison to prior works.

Method	Train Accuracy (%)	Test Accuracy (%)	Training Time (hr)	Prediction Time (s / batch)	Testing NFE
Vanilla NODE	99.898 ± 0.066	97.612 ± 0.163	0.54 ± 0.001	0.088 ± 0.020	303.559 ± 3.194
STEER ¶	100.00 ± 0.000	97.94 ± 0.03	1.31 ± 0.07	0.092 ± 0.002	265.0 ± 3.46
TayNODE ¶	98.98 ± 0.06	97.89 ± 0.00	1.19 ± 0.07	0.079 ± 0.007	80.3 ± 0.43
ERNODE ¶	99.71 ± 0.28	97.32 ± 0.06	0.82 ± 0.02	0.060 ± 0.001	177.0 ± 0.00
SRNODE ¶	100.00 ± 0.000	98.08 ± 0.15	1.24 ± 0.06	0.094 ± 0.003	259.0 ± 3.46
Local Unbiased ERNODE	99.447 ± 0.039	97.526 ± 0.131	0.49 ± 0.002	0.046 ± 0.002	187.961 ± 1.812
Local Biased ERNODE	99.477 ± 0.051	97.488 ± 0.016	1.12 ± 0.065	0.044 ± 0.002	182.849 ± 1.578
Vanilla NSDE	98.27 ± 0.11	96.66 ± 0.16	2.70 ± 0.00	0.51 ± 0.07	313.86 ± 2.94
ERNSDE ¶	98.16 ± 0.11	96.27 ± 0.35	4.19 ± 0.04	7.23 ± 0.14	184.67 ± 2.31
SRNSDE ¶	98.79 ± 0.12	96.80 ± 0.07	8.54 ± 0.37	14.50 ± 0.40	382.00 ± 4.00
Local Unbiased ERNSDE	98.05 ± 0.09	96.57 ± 0.13	2.10 ± 0.01	0.39 ± 0.10	228.93 ± 1.77
Local Biased ERNSDE	98.02 ± 0.07	96.44 ± 0.16	1.90 ± 0.00	0.36 ± 0.03	230.10 ± 0.71

Table 6.2: MNIST Image Classification using Neural DE: Using local unbiased regularization on neural ODE speeds up training by 1.1× and predictions by 1.9× while reducing the total NFEs to 0.619×. Local Biased Regularization tends to slow down training for smaller models on GPU while it further reduces the NFEs by 0.602×. For Neural SDE, we observe a similar reduction of NFEs by $0.729\times - 0.733\times$ and a training time improvement of $1.28\times - 1.42\times$. The best global regularization method gets lower NFEs but overall takes more wall clock than the best performing local regularization method.

Baselines: ERNSDE and SRNSDE results were taken from Pal et al. [2021]. These were trained for 40 epochs, nearly equivalent to training for 4000 iterations.

Results: We summarize the results in Table 6.2 and Figure 6.4. Local regularization improves training and prediction performance while keeping the test accuracy nearly constant.

3.2 Physionet Time Series Interpolation

Training Details: We use the experimental setup for Physionet 2012 Challenge Dataset [Citi and Barbieri, 2012] from Kelly et al. [2020]. We use a Latent Neural ODE [Rubanova et al., 2019] to perform time series interpolation on the dataset. We use the preprocessed dataset from Kelly et al. [2020] to ensure a fair comparison and independent runs are performed using an 80:20 split of the dataset.

Method	Test Loss ($\times 10^{-3}$)	Training Time (hr)	Prediction Time (s / batch)	Testing NFE
Vanilla NODE	3.41 ± 0.10	2.48 ± 0.22	0.16 ± 0.01	758.0 ± 25.87
STEER ¶	3.48 ± 0.01	1.62 ± 0.26	0.54 ± 0.06	699.0 ± 141.1
TayNODE ¶	4.21 ± 0.01	12.3 ± 0.32	0.22 ± 0.02	167.3 ± 11.93
ERNODE ¶	3.57 ± 0.00	0.94 ± 0.13	0.21 ± 0.02	287.0 ± 17.32
SRNODE ¶	3.58 ± 0.05	0.87 ± 0.09	0.20 ± 0.01	273.0 ± 0.000
Local Unbiased ERNODE	3.64 ± 0.07	2.31 ± 0.02	0.09 ± 0.00	422.0 ± 4.580
Local Biased ERNODE	3.63 ± 0.08	2.12 ± 0.24	0.10 ± 0.01	463.0 ± 63.02

Table 6.3: Physionet Time Series Interpolation: Local Regularization reduces NFEs by $0.556\times - 0.610\times$ reducing the prediction timings by $1.6\times - 1.78\times$. Our methods additionally improve training timings by $1.073\times - 1.167\times$. We note that the difference in training time compared to (E/S)RNODE methods is due to change in the sensitivity algorithm.

For specific model architecture details, we refer the readers to Pal et al. [2021]. We train the model for a total of 3000 iterations using Adamax [Kingma and Ba, 2014b] with a learning rate of 0.01 with 10^{-5} inverse decay per step. We use a batch size of 512. We diverge from Pal et al. [2021], in using the regularization term as $(\mathsf{E}_{\mathsf{Est}})_{\mathsf{t}_{\mathsf{reg}}} \cdot |dt|_{\mathsf{t}_{\mathsf{reg}}}$ instead of the squared regularization term $\sum_j (\mathsf{E}_{\mathsf{Est}})_j^2$. Additionally, we decay the regularization coefficient exponentially from 100 to 10 over the 3000 training iterations.

Baselines: Vanilla NODE was trained with the exact aforementioned configuration. All the other baselines were trained using discrete sensitivity analysis, and the exact details are present in Pal et al. [2021].

Results: We summarize the results in Figure 6.5 and Table 6.3.

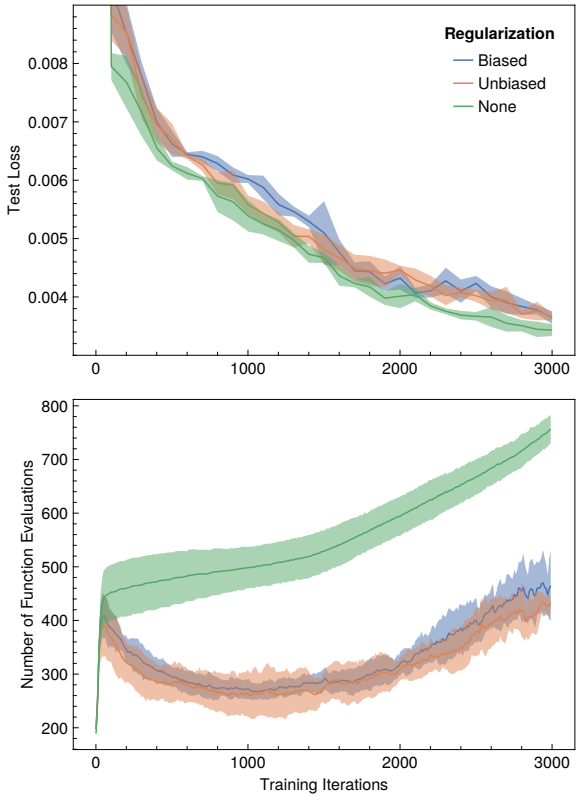

Figure 6.5: Physionet Time Series Interpolation using Latent ODE

3.3 CIFAR10 Image Classification

3.3.1 Neural Ordinary Differential Equation

Training Details: We use the CNN architecture for CIFAR10 as described in Poli et al. [2020]. We train the models for 31250 steps with Adam [Kingma and Ba, 2014b] using a cosine-annealing learning rate scheduler from 0.003 to 0.0001. We train the models with a batch size of 32 and keep the regularization coefficient fixed at 2.5. We use Tsit5 [Tsitouras, 2011] with a tolerance of 10^{-4}.

Configuration	Method	Train Accuracy (%)	Test Accuracy (%)	Training Time (s / batch)	Prediction Time (s / batch)	Testing NFE
Standard	Vanilla	83.683 ± 1.450	67.394 ± 0.849	0.457 ± 0.018	0.130 ± 0.013	115.315 ± 12.136
	Local Unbiased ER	83.665 ± 0.805	67.678 ± 0.874	0.399 ± 0.014	0.096 ± 0.007	89.048 ± 7.335
	Local Biased ER	83.958 ± 1.032	67.745 ± 0.824	0.555 ± 0.008	0.088 ± 0.003	81.301 ± 1.255
Multi-Scale	Vanilla	92.807 ± 12.458	80.048 ± 6.740	0.572 ± 0.012	0.170 ± 0.005	27.616 ± 0.905
	Local Unbiased ER	94.159 ± 9.694	80.432 ± 5.548	0.641 ± 0.025	0.175 ± 0.019	27.760 ± 0.177
	Local Biased ER	99.987 ± 0.023	83.460 ± 0.727	0.774 ± 0.293	0.163 ± 0.015	26.334 ± 0.992

Table 6.4: CIFAR10 Image Classification using Neural DE: For the standard Neural ODE, local regularization reduces the NFE by $0.705\times - 0.772\times$, thereby improving prediction timings by $1.35\times - 1.477\times$. However, unregularized model training takes $0.823\times$ the time for the biased model. For multi-scale models, the NFE and prediction time improvements are marginal and come at the cost of higher training time.

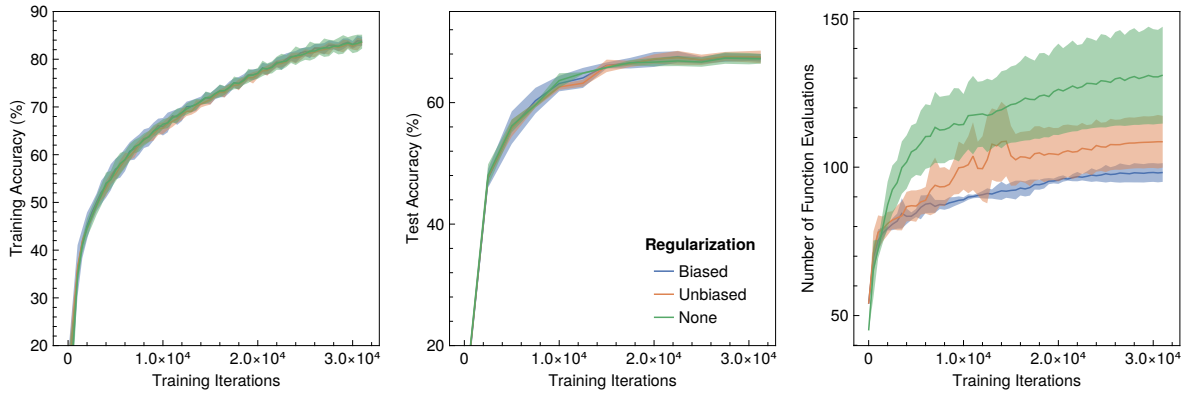

Figure 6.6: CIFAR10 Image Classification using Standard Neural ODE

Results: We summarize the results in Figure 6.6 and Table 6.4.

3.3.2 Multiscale Neural ODE

Training Details: We modify the Tiny Multiscale DEQ architecture for CIFAR10 from Bai et al. [2020] as Multiscale Neural ODE with Input Injection. To stabilize the training for larger models, we exponentially increase the regularization coefficient from 0.1 to 5.0. We train with a batch size of 128 using VCAB3 [Wanner and Hairer, 1996] with a tolerance of 0.05.

Results: We summarize the results in Figure 6.7 and Table 6.4. The benefits from regularization for NFEs and prediction timings seem marginal. However, regularization using biased sampling makes the training dynamics stable as observed in Figure 6.7.

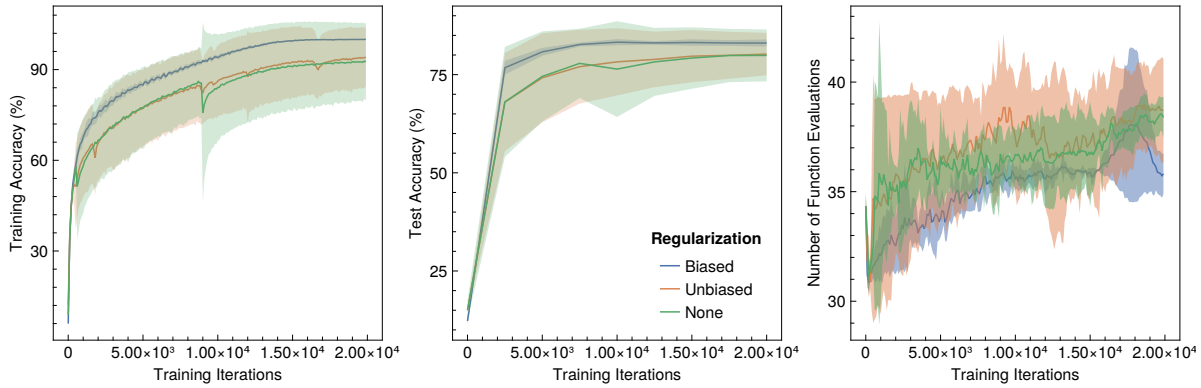

Figure 6.7: CIFAR10 Image Classification using Multi-Scale Neural ODE

4 Discussion

In this chapter, we have shown that we can obtain similar properties to global regularization by regularizing dynamical systems at randomly sampled time points. Additionally, this comes with the benefit of not being forced into a specific sensitivity analysis method. We have taken every experiment in Pal et al. [2021] and empirically showed that our local regularization works at par with global regularization. However, our experiments using stiffness estimate for local regularization did not yield positive results and were not presented in this chapter. Thus, we have demonstrated that we can "close the blackbox" and still leverage all the benefits of internal solver heuristics to improve training and predictions of neural differential equations.

4.1 Limitations

We note the following limitations of our work:

- Similar to Pal et al. [2021], if the objective is to learn the actual dynamical system, our method will not yield proper results. Our method is applicable only when the final state is relevant, i.e., in most classical deep learning tasks.
- Regularization introduces a new regularization coefficient hyperparameter which, if not tuned correctly, can lead to unstable dynamics or might negate the scheme's usefulness.

Part III

OPEN SOURCE SOFTWARES

CHAPTER 7

Lux.jl: Bridging Scientific Computing and Deep Learning

1 Introduction

2 Composability via Generic Parameterization

One of the distinguishing features of Lux.jl is its generic parameters interface. Lux can accept any special parameter type as long as the parameters are accessible via *getproperty*. This allows Lux to seamlessly interface with packages that are completely agnostic to the specifics of Lux. This is in contrast to most prior works that require glue code for interfacing – like Flux.jl [Innes, 2018b] – or requires reimplementing algorithms in specific Domain Specific Languages (DSLs) – like Pytorch [Paszke et al., 2019, 2017], JAX [Bradbury et al., 2018], Tensorflow [Abadi et al., 2015].

Scientific Computing softwares differ from most modern ML softwares in that they are designed to operate on arrays, while ML softwares operate on deeply nested structures. This is a fundamental difference that makes it difficult to interface between the two. However, Lux.jl is designed to be agnostic to the underlying data structure. In this section we will demonstrate several examples interfacing Lux with packages completely agnostic to Lux.

2.1 Neural Differential Equations

In this thesis, we have described Neural ODEs in great depth. However, for physics based modelling we often need to rely on higher order differential equations. In this section, we will describe how to model a second order differential equation using Lux.jl and OrdinaryDiffEq.jl. We will attempt to model the acceleration of a system

using a Neural Network[1]:

$$\frac{d^2u}{dt^2} = \mathbf{NN}(u) \tag{7.1}$$

```julia
using ComponentArrays, Lux, Optimization, OptimizationOptimisers, OrdinaryDiffEq, RecursiveArrayTools,
  SciMLSensitivity, StableRNGs

u0, du0, tspan = [0.0f0; 2.0f0], [0.0f0; 0.0f0], (0.0f0, 1.0f0)
ts = range(tspan[1], tspan[2], length=21)

model = Chain(Dense(2 ⇒ 50, tanh), Dense(50 ⇒ 2))
ps, st = Lux.setup(StableRNG(1234), model)
ps = ComponentArray(ps)

ff(du, u, p, t) = first(model(u, p, st))
prob = SecondOrderODEProblem{false}(ff, u0, du0, tspan, ps)

predict(θ) = Array(solve(prob, Tsit5(); p=θ, saveat=ts))

y_true = vcat(collect(0:0.05f0:1)', collect(2:-0.05f0:1)')

loss_function(θ) = sum(abs2, predict(θ)[1:2, :] .- y_true)

optprob = OptimizationProblem(OptimizationFunction{false}((x, p) -> loss_function(x),
  Optimization.AutoZygote()), ps)

res = Optimization.solve(optprob, Adam(0.01f0); maxiters=1000)
```

Figure 7.1 shows that our model is able to accurately learn the dynamics of the system from a few discrete data points. This is a very simple example, but it demonstrates the composability of Lux.jl and DifferentialEquations ecosystem.

2.2 Gradient Free Optimization Algorithms

Lux allows the parameters of a complicated neural network to be represented as a flattened vector. This allows it to interface directly with optimization packages without any glue code. In this example, we will train a neural network with gradient-free optimization algorithms to learn the function:

$$r(\theta) = e^{sin(\theta)} - 2cos(4\theta) + sin\left(\frac{2\theta - \pi}{12}\right)^5 \tag{7.2}$$

$$\texttt{where}\ \ \theta \in [0, 2\pi] \tag{7.3}$$

We will use algorithms implemented in packages that are agnostic to the specifics of Lux[2]:

- Covariance Matrix Adaptation Evolutionary Strategy (CMAES) [Hansen, 2016] from CMAEvolutionStrategy.jl.
- LN_NEWUOA [Powell, 2006] from NLopt.jl [Johnson and Schueller, 2021]. Since Lux allows flattened parameters we can easily interoperate with NLopt which is a C library.

[1] We have modified this example from https://docs.sciml.ai/SciMLSensitivity/stable/examples/ode/second_order_neural/

[2] Note that these are not the most efficient algorithms to solve the problem, but these simply demonstrate the composability of Lux.

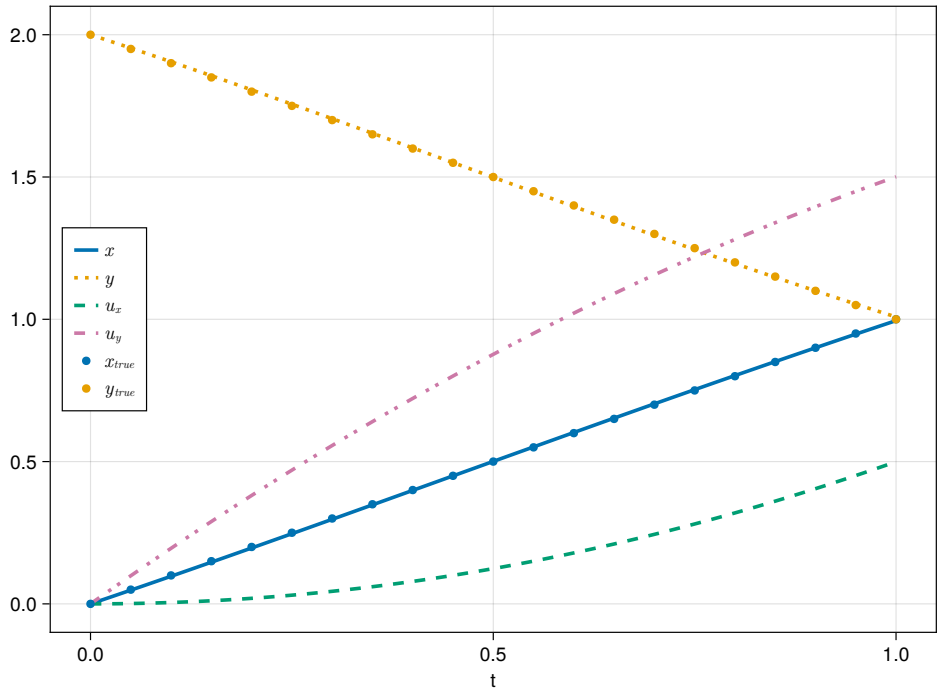

Figure 7.1: Learning 2^{nd} order differential equation with Lux.jl.

```julia
using ComponentArrays, FillArrays, Lux, Optimization, OptimizationCMAEvolutionStrategy,
    OptimizationNLopt, StableRNGs, Statistics, Zygote

r(θ) = exp(sin(θ)) - 2cos(4θ) + sin((2θ - oftype(θ, π)) / 12)^5

θs = collect(0:1.0e-2:2π)
rs = r.(θs)

model = Chain(Dense(1 => 16, tanh), Dense(16 => 16, tanh), Dense(16 => 1))
ps, st = Lux.setup(StableRNG(1234), model)
ps = ComponentArray(ps)
ps_flat, ps_ax = Float64.(getdata(ps)), getaxes(ps)

function loss_function(ps, _)
    ps_ = ComponentArray(ps, ps_ax)
    r_pred, st_ = model(reshape(θs, 1, :), ps_, st)
    return mean(abs2, vec(r_pred) .- rs)
end

opt_func = OptimizationFunction{false}(loss_function, Optimization.AutoZygote())
opt_prob = OptimizationProblem(opt_func, ps_flat)

sol_LN_NEWUOA = solve(opt_prob, OptimizationNLopt.NLopt.LN_NEWUOA(); maxiters=100_000)
sol_CMAES = solve(opt_prob, CMAEvolutionStrategyOpt(); maxiters=100_000)
```

2.3 Physics Informed Neural Networks

(**todo**) I want to showcase higher-order AD: not sure what example to use

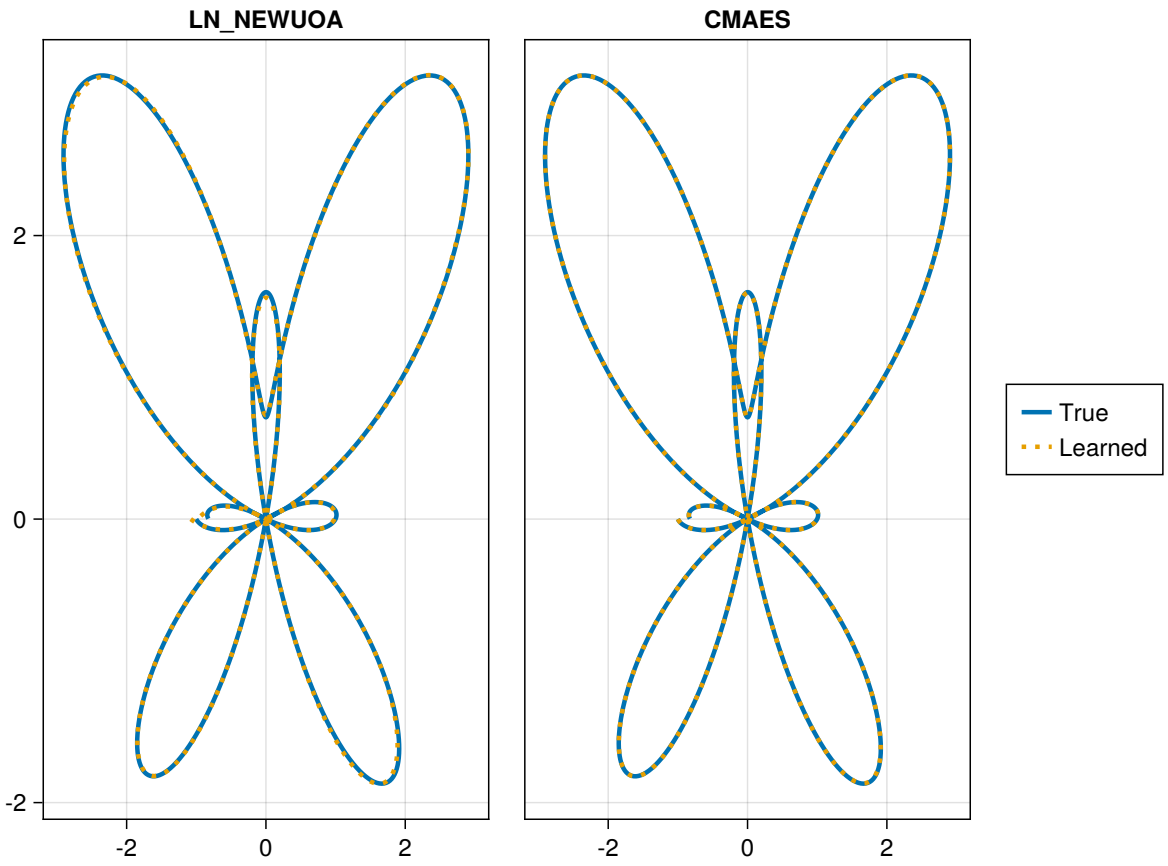

Figure 7.2: Gradient Free Optimization to train a neural network to approximate the function $r(\theta) = e^{sin(\theta)} - 2cos(4\theta) + sin\left(\frac{2\theta - \pi}{12}\right)^5$.

2.4 Taylor Mode Automatic Differentiation

3 Leveraging Cross-Language Capabilities

(**todo**) pycallchainrules for lux + pytorch and lux + jax

4 Performance

(**todo**) compare against flux and maybe knet

5 Discussion

5.1 Current Limitations

Part IV

CONCLUSIONS AND FUTURE WORK

CHAPTER 8

Conclusion

References

Martín Abadi and Gordon D Plotkin. A simple differentiable programming language. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, 2019. (page 35)

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org. (page 48)

Brandon Amos and J Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*, pages 136–145. PMLR, 2017. (page 36)

Ranjan Anantharaman, Yingbo Ma, Shashi Gowda, Chris Laughman, Viral Shah, Alan Edelman, and Chris Rackauckas. Accelerating simulation of stiff nonlinear systems using continuous-time echo state networks. *arXiv preprint arXiv:2010.04004*, 2020. (page 40)

Donald G Anderson. Iterative procedures for nonlinear integral equations. *Journal of the ACM (JACM)*, 12(4): 547–560, 1965. (page 15)

Uri M Ascher and Linda R Petzold. *Computer methods for ordinary differential equations and differential-algebraic equations*, volume 61. Siam, 1998. (page 5)

Shaojie Bai, J Zico Kolter, and Vladlen Koltun. Deep equilibrium models. *arXiv preprint arXiv:1909.01377*, 2019. (page 13, 18, 21, 35, 37)

Shaojie Bai, Vladlen Koltun, and J. Zico Kolter. Multiscale Deep Equilibrium Models. *arXiv:2006.08656 [cs, stat]*, November 2020. URL `http://arxiv.org/abs/2006.08656`. arXiv: 2006.08656. (page 15, 18, 21, 22, 23, 24, 25, 26, 45)

Shaojie Bai, Vladlen Koltun, and J Zico Kolter. Neural deep equilibrium solvers. In *International Conference on Learning Representations*, 2021a. (page 16, 21, 26)

Shaojie Bai, Vladlen Koltun, and J Zico Kolter. Stabilizing equilibrium models by jacobian regularization. *arXiv preprint arXiv:2106.14342*, 2021b. (page 16, 18)

HS Behl, A Ghosh, E Dupont, PHS Torr, and V Namboodiri. Steer: simple temporal regularization for neural odes. pages 1–13. Neural Information Processing Systems Foundation, Inc., 2020. (page 10, 27, 29, 37, 41)

Jesse Bettencourt, Matthew J Johnson, and David Duvenaud. Taylor-mode automatic differentiation for higher-order derivatives in jax. 2019. (page 10, 31)

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. doi: 10.1137/141000671. (page 22, 28, 41)

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`. (page 48)

Charles G Broyden. A class of methods for solving nonlinear simultaneous equations. *Mathematics of computation*, 19(92):577–593, 1965. (page 14)

A.B. Bulsari. *Neural Networks for Chemical Engineers*. Computer-aided chemical engineering. Elsevier, 1995. ISBN 9780444820976. URL `https://books.google.com/books?id=atBTAAAAMAAJ`. (page 20)

Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in neural information processing systems*, pages 6571–6583, 2018. (page 6, 8, 9, 18, 27, 29, 32, 36, 37, 41)

Luca Citi and Riccardo Barbieri. Physionet 2012 challenge: Predicting mortality of icu patients using a cascaded svm-glm paradigm. In *2012 Computing in Cardiology*, pages 257–260. IEEE, 2012. (page 43)

Benjamin Dauvergne and Laurent Hascoët. The data-flow equations of checkpointing in reverse automatic differentiation. In *International Conference on Computational Science*, pages 566–573. Springer, 2006. (page 29)

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. (page 22)

Franck Djeumou, Cyrus Neary, Eric Goubault, Sylvie Putot, and Ufuk Topcu. Taylor-lagrange neural ordinary differential equations: Toward fast training and evaluation of neural odes, 2022. (page 10)

John R Dormand and Peter J Prince. A family of embedded runge-kutta formulae. *Journal of computational and applied mathematics*, 6(1):19–26, 1980. (page 5, 11)

Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented neural odes. *Advances in Neural Information Processing Systems*, 32, 2019. (page 18)

Dale R Durran. The third-order adams-bashforth method: An attractive alternative to leapfrog time differencing. *Monthly weather review*, 119(3):702–720, 1991. (page 5)

Leonhard Euler. *Institutionum calculi integralis*, volume 1. impensis Academiae imperialis scientiarum, 1824. (page 3, 4)

Erwin Fehlberg. *Classical fifth-, sixth-, seventh-, and eighth-order Runge-Kutta formulas with stepsize control*. National Aeronautics and Space Administration, 1968. (page 5)

Chris Finlay, Jörn-Henrik Jacobsen, Levon Nurbekyan, and Adam M Oberman. How to train your neural ode. *arXiv preprint arXiv:2002.02798*, 2020. (page 10, 18, 28, 37)

Samy Wu Fung, Howard Heaton, Qiuwei Li, Daniel McKenzie, Stanley Osher, and Wotao Yin. Jfb: Jacobian-free backpropagation for implicit networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2022. (page 16)

Laurent El Ghaoui, Fangda Gu, Bertrand Travacca, Armin Askari, and Alicia Y. Tsai. Implicit Deep Learning. *arXiv:1908.06315 [cs, math, stat]*, August 2020. URL http://arxiv.org/abs/1908.06315. arXiv: 1908.06315. (page 18)

Amir Gholami, Kurt Keutzer, and George Biros. Anode: Unconditionally accurate memory-efficient gradients for neural odes. *arXiv preprint arXiv:1902.10298*, 2019. (page 7, 29)

Will Grathwohl, Ricky TQ Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. Ffjord: Free-form continuous dynamics for scalable reversible generative models. *arXiv preprint arXiv:1810.01367*, 2018. (page 9, 18)

Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008. (page 10)

Ernst Hairer, Syvert Norsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*, volume 8. 01 1993. ISBN 978-3-540-56670-0. doi: 10.1007/978-3-540-78862-1. (page 5, 29)

Nikolaus Hansen. The cma evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*, 2016. (page 49)

Desmond J Higham and Lloyd N Trefethen. Stiffness of odes. *BIT Numerical Mathematics*, 33(2):285–303, 1993. (page 6)

Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, 31(3):363–396, sep 2005. ISSN 0098-3500. doi: 10.1145/1089014.1089020. URL https://doi.org/10.1145/1089014.1089020. (page 6, 41)

Michael F Hutchinson. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics-Simulation and Computation*, 18(3):1059–1076, 1989. (page 9)

Michael Innes. Don't unroll adjoint: Differentiating ssa-form programs. *arXiv preprint arXiv:1810.07951*, 2018a. (page 29)

Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. Fashionable modelling with flux. *CoRR*, abs/1811.01457, 2018. URL http://arxiv.org/abs/1811.01457. (page 30)

Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 2018b. doi: 10.21105/joss.00602. (page 48)

Joseph Ian Jeisman. *Estimation of the parameters of stochastic differential equations*. PhD thesis, Queensland University of Technology, 2006. (page 34)

Steven G Johnson. Notes on adjoint methods for 18.335. *Introduction to Numerical Methods*, 2006. (page 13)

Steven G Johnson and Julien Schueller. Nlopt: Nonlinear optimization library. *Astrophysics Source Code Library*, pages ascl–2111, 2021. (page 49)

Carl T Kelley. *Iterative methods for optimization*. SIAM, 1999. (page 36)

Jacob Kelly, Jesse Bettencourt, Matthew James Johnson, and David Duvenaud. Learning differential equations that are easy to solve. *arXiv preprint arXiv:2007.04504*, 2020. (page 10, 18, 22, 27, 28, 29, 30, 31, 32, 37, 41, 43)

Patrick Kidger, Ricky T. Q. Chen, and Terry Lyons. "hey, that's not an ode": Faster ode adjoints via seminorms, 2021. (page 10, 29)

Suyong Kim, Weiqi Ji, Sili Deng, Yingbo Ma, and Christopher Rackauckas. Stiff neural ordinary differential equations. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 31(9):093122, 2021. (page 4, 6, 41)

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014a. (page 22, 24)

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014b. URL http://arxiv.org/abs/1412.6980. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. (page 32, 35, 41, 42, 44)

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009. (page 22)

Wilhelm Kutta. Beitrag zur naherungsweisen integration totaler differentialgleichungen. *Z. Math. Phys.*, 46: 435–453, 1901. (page 4)

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. (page 22, 41)

Xuanqing Liu, Tesi Xiao, Si Si, Qin Cao, Sanjiv Kumar, and Cho-Jui Hsieh. Neural sde: Stabilizing neural ode networks with stochastic noise, 2019. (page 27, 37)

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017. (page 23)

Alexander Lück and Verena Wolf. Generalized method of moments for estimating parameters of stochastic reaction networks. *BMC systems biology*, 10(1):1–12, 2016. (page 34)

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. 2011. (page 22)

Derek Onken and Lars Ruthotto. Discretize-optimize vs. optimize-discretize for time-series regression and continuous normalizing flows. *arXiv preprint arXiv:2005.13420*, 2020. (page 7, 29)

Avik Pal. Lux: Explicit Parameterization of Deep Neural Networks in Julia. April 2023. doi: 10.5281/zenodo.7808904. URL https://doi.org/10.5281/zenodo.7808904. (page 22, 41)

Avik Pal, Yingbo Ma, Viral Shah, and Christopher V Rackauckas. Opening the Blackbox: Accelerating Neural Differential Equations by Regularizing Internal Solver Heuristics. In *International Conference on Machine Learning*, pages 8325–8335. PMLR, 2021. (page 10, 28, 37, 38, 40, 41, 42, 43, 44, 46)

Avik Pal, Alan Edelman, and Christopher Rackauckas. Continuous Deep Equilibrium Models: Training Neural Odes Faster by Integrating Them to Infinity. *arXiv preprint arXiv:2201.12240*, 2022. (page 19, 37)

Avik Pal, Alan Edelman, and Chris Rackauckas. Locally Regularized Neural Differential Equations: Some Black Boxes Were Meant to Remain Closed! *arXiv preprint arXiv:2303.02262*, 2023. (page 10, 38)

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017. (page 48)

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019. (page 48)

Robert Piché. An l-stable rosenbrock method for step-by-step time integration in structural dynamics. *Computer Methods in Applied Mechanics and Engineering*, 126(3-4):343–354, 1995. (page ix, 40)

Michael Poli, Stefano Massaroli, Atsushi Yamashita, Hajime Asama, and Jinkyoo Park. Hypersolvers: Toward fast continuous-depth models. *arXiv preprint arXiv:2007.09601*, 2020. (page 10, 11, 36, 37, 44)

Michael JD Powell. The newuoa software for unconstrained optimization without derivatives. *Large-scale nonlinear optimization*, pages 255–297, 2006. (page 49)

Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999. (page 31)

Chris Rackauckas. Parallel computing and scientific machine learning (sciml): Methods and applications (mit 18.337j/6.338j). https://github.com/SciML/SciMLBook, 2019. (page 4)

Chris Rackauckas and Qing Nie. Stability-optimized high order methods and stiffness detection for pathwise stiff stochastic differential equations. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2020. (page 5, 33)

Christopher Rackauckas. Engineering trade-offs in automatic differentiation: From tensorflow and pytorch to jax and julia, Jan 2022. URL http://www.stochasticlifestyle.com/engineering-trade-offs-in-automatic-differentiation-from-tensorflow-and-pytorch-to-jax-and-julia/. (page 38)

Christopher Rackauckas and Qing Nie. Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia. *The Journal of Open Research Software*, 5(1), 2017a. doi: 10.5334/jors.151. URL https://app.dimensions.ai/details/publication/pub.1085583166andhttp://openresearchsoftware.metajnl.com/articles/10.5334/jors.151/galley/245/download/. Exported from https://app.dimensions.ai on 2019/05/05. (page 22)

Christopher Rackauckas and Qing Nie. Adaptive methods for stochastic differential equations via natural embeddings and rejection sampling with memory. *Discrete and continuous dynamical systems. Series B*, 22(7):2731, 2017b. (page 5, 42)

Christopher Rackauckas and Qing Nie. Confederated modular differential equation apis for accelerated algorithm development and benchmarking. *Advances in Engineering Software*, 132:1–6, 2019. (page 6)

Christopher Rackauckas, Alan Edelman, Keno Fischer, Mike Innes, Elliot Saba, Viral B Shah, and Will Tebbutt. Generalized physics-informed learning through language-wide differentiable programming. (page 35)

Christopher Rackauckas, Yingbo Ma, Vaibhav Dixit, Xingjian Guo, Mike Innes, Jarrett Revels, Joakim Nyberg, and Vijay Ivaturi. A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions. *arXiv preprint arXiv:1812.01892*, 2018. (page 22)

Christopher Rackauckas, Mike Innes, Yingbo Ma, Jesse Bettencourt, Lyndon White, and Vaibhav Dixit. Diffeqflux.jl - A julia library for neural differential equations. *CoRR*, abs/1902.02376, 2019. URL http://arxiv.org/abs/1902.02376. (page 28, 30, 41)

Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, Ali Ramadhan, and Alan Edelman. Universal differential equations for scientific machine learning, 2020. (page 22, 27, 37)

R Rico-Martinez, K Krischer, IG Kevrekidis, MC Kube, and JL Hudson. Discrete-vs. continuous-time nonlinear signal processing of cu electrodissolution data. *Chemical Engineering Communications*, 118(1):25–48, 1992. (page 20)

HH Robertson. The solution of a set of reaction rate equations. *Numerical analysis: an introduction*, 178182, 1966. (page ix, 40)

Yulia Rubanova, Ricky TQ Chen, and David Duvenaud. Latent odes for irregularly-sampled time series. *arXiv preprint arXiv:1907.03907*, 2019. (page 9, 32, 43)

Carl Runge. Über die numerische auflösung von differentialgleichungen. *Mathematische Annalen*, 46(2):167–178, 1895. (page 4)

L. F. Shampine. Stiffness and nonstiff differential equation solvers, ii: Detecting stiffness with runge-kutta methods. *ACM Trans. Math. Softw.*, 3(1):44–53, March 1977. ISSN 0098-3500. doi: 10.1145/355719.355722. URL https://doi.org/10.1145/355719.355722. (page 6)

Lawrence F Shampine and Charles William Gear. A user's view of solving stiff ordinary differential equations. *SIAM review*, 21(1):1–17, 1979. (page 5)

Lawrence F Shampine and Skip Thompson. Stiff systems. *Scholarpedia*, 2(3):2855, 2007. (page 6)

Xing Shen, Xiaoliang Cheng, and Kewei Liang. Deep euler method: solving odes by approximating the local truncation error of the euler method. *arXiv preprint arXiv:2003.09573*, 2020. (page 36)

Jack Sherman and Winifred J Morrison. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics*, 21(1):124–127, 1950. (page 15)

Ikaro Silva, George Moody, Daniel J Scott, Leo A Celi, and Roger G Mark. Predicting in-hospital mortality of icu patients: The physionet/computing in cardiology challenge 2012. In *2012 Computing in Cardiology*, pages 245–248. IEEE, 2012. (page 32)

Ch Tsitouras. Runge–kutta pairs of order 5 (4) satisfying only the first column simplifying assumption. *Computers & Mathematics with Applications*, 62(2):770–775, 2011. (page 4, 5, 22, 30, 42, 44)

Fei Wang, James Decker, Xilun Wu, Gregory Essertel, and Tiark Rompf. Backpropagation with callbacks: Foundations for efficient and expressive differentiable programming. *Advances in Neural Information Processing Systems*, 31:10180–10191, 2018. (page 35)

Gerhard Wanner and Ernst Hairer. *Solving ordinary differential equations II*, volume 375. Springer Berlin Heidelberg New York, 1996. (page 4, 5, 6, 23, 24, 25, 36, 45)

Hedi Xia, Vai Suliafu, Hangjie Ji, Tan Nguyen, Andrea Bertozzi, Stanley Osher, and Bao Wang. Heavy ball neural ordinary differential equations. *Advances in Neural Information Processing Systems*, 34, 2021. (page 10)

Haibin Zhang, Yi Xue, Chunhua Zhang, and Lili Dong. Computing the high order derivatives with automatic differentiation and its application in chebyshev's method. In *2008 Fourth International Conference on Natural Computation*, volume 1, pages 304–308. IEEE, 2008. (page 29)

Hong Zhang and Adrian Sandu. Fatode: a library for forward, adjoint, and tangent linear integration of odes. *SIAM Journal on Scientific Computing*, 36(5):C504–C523, 2014. (page 29)

Juntang Zhuang, Tommy Tang, Yifan Ding, Sekhar Tatikonda, Nicha Dvornek, Xenophon Papademetris, and James Duncan. Adabelief optimizer: Adapting stepsizes by the belief in observed gradients. *Conference on Neural Information Processing Systems*, 2020. (page 34)

Juntang Zhuang, Nicha C Dvornek, sekhar tatikonda, and James s Duncan. Mali: A memory efficient and reverse accurate integrator for neural odes. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=blfSjHeFM_e. (page 10, 29)