Title
Towards Faster and More Accurate Neural ODEs

Author
Xia, Hedi

Publication Date
2023

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Towards Faster and More Accurate Neural ODEs

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Mathematics

by

Hedi Xia

2023

ABSTRACT OF THE DISSERTATION

Towards Faster and More Accurate Neural ODEs

by

Hedi Xia
Doctor of Philosophy in Mathematics
University of California, Los Angeles, 2023
Professor Stanley J. Osher, Co-Chair
Professor Andrea L. Bertozzi, Co-Chair

Neural Ordinary Differential Equations (NODEs) [Che+18] have improved accuracy and memory efficiency over general deep neural networks but suffer from the vanishing gradient problem and the large number of function evaluations (NFEs) problem. In chapter 3, we proposed Heavy Ball NODEs (HBNODEs), leveraging the continuous limit of classical momentum-accelerated gradient descent, to improve NODEs training and inference. We show that HBNODEs has two major advantages: (1) The adjoint of HBNODEs also satisfies Heavy Ball ODEs, accelerating both forward and backward solvers; (2) the spectrum of HBNODEs is well-structured so that HBNODEs is capable of learning long-term dependencies from long time series. In chapter 4, we proposed Graph Neural Diffusion with a Source Term (GRAND++), a class of NODEs on graphs, for deep learning on graphs with a limited number of labels. We study the limiting behavior of GRAND++ and show that it does not converge to a constant even when the depth goes to infinity. We provide experiments to show that GRAND++ can provide accurate classification even when the number of labels is limited. In chapter 5, we study how proximal implicit solvers can improve NODEs training in some

scenarios. We show that for stiff NODEs, proximal implicit solvers have smaller NFEs than commonly used explicit solvers, and thus speed up training.

The dissertation of Hedi Xia is approved.

Bao Wang

Lin Yang

Guido Montufar

Andrea L. Bertozzi, Committee Co-Chair

Stanley J. Osher, Committee Co-Chair

University of California, Los Angeles

2023

To my family, friends, collaborators,

and everyone who supports me

TABLE OF CONTENTS

List of Figures

List of Tables

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisors, Professor Stanley J. Osher and Professor Andrea L. Bertozzi, for their supervision, support, and encouragement during my research. Professor Osher introduced me to working on the intersection of deep learning and differential equations, and Professor Bertozzi introduced me to working on graph-related deep learning problems. I would like to pay special regards to Professor Bao Wang for his hands-on help with experiments and paper writing. I would also like to thank the other members of my committee, Professor Guido Montufar and Professor Yang Lin, for their advice and feedback.

I would like to thank Vai Suliafu, Tan Nguyen, Matthew Thorpe, Justin Baker, Bohan Chen, Hangjie Ji, and all of my collaborators for their ideas and hard work on our shared projects. Chapter 3 is based on "Heavy Ball Neural ODEs" [Xia+21] by Hedi Xia, Vai Suliafu, Hangjie Ji, Tan M. Nguyen, Andrea L. Bertozzi, Stanley J. Osher, and Bao Wang. Hedi Xia contributed to the model design, theory, coding and numerical experiments in section 3.3 and 3.5.3in the paper. Vai Suliafu contributed to the experiments in 3.5.1 and 3.5.2. Bao Wang provided the general idea and helped with model and experiment designs. All authors assisted with manuscript preparation. Chapter 4 is based on "GRAND++: Graph Neural Diffusion with A Source Term" [Tho+22] by Matthew Thorpe, Tan Minh Nguyen, Hedi Xia, Thomas Strohmer, Andrea Bertozzi, Stanley Osher and Bao Wang. Matthew Thorpe contributed to the theory and ideas. Tan Nguyen and Hedi Xia cooperated on the experiments on CORA, CiteSeer, PubMed, CoauthorCS, Computer, Photo. Tan Nguyen also contributed to the experiments on OGBN-arXiv. Hedi Xia also contributed to the coding and statistical testing of the results. Bao Wang provided the general idea and helped with model and experiment designs. All authors assisted with manuscript preparation. This research is sponsored by NSF grants DMS-1924935, DMS-1952339, DMS-2027248 and NSF CCF-1934568, DOE grant DE-SC0021142, ONR grant N00014-18-1-2527, and the MURI grant N00014-20-1-2787. Chapter 5 is based on "Proximal Implicit ODE Solvers for Accelerating Learning Neural

ODEs" [Bak+22b] by Justin Baker, Hedi Xia, Yiwei Wang, Elena Cherkaev, Akil Narayan, Long Chen, Jack Xin, Andrea L. Bertozzi, Stanley J. Osher, and Bao Wang. Justin Baker contributed to coding, model design, and experiments in section 5.4.1, 5.4.2. Hedi Xia contributed to experiments in section 5.4.3. Justin Baker, Hedi Xia, Yiwei Wang, and Bao Wang all contributed to the theory in the paper. Bao Wang provided the general idea and helped with model and experiment designs. All authors assisted with manuscript preparation. This research is sponsored by NSF grants DMS-1924935, DMS-1952339, DMS-2110145, DMS-2152762, and DMS-2208361, and DOE grant DE-SC0021142.

I would also like to thank Yizhou Chen, Yanli Liu, Mingtao Xia, and Xia Li for their academic discussions and suggestions. I would like to thank all of my friends for their friendship and emotional support, including Yushan Han, Haiyu Huang, Haoling Xiang, Weiyi Liu, Ben Spitz, and Jerry Luo.

Last but not least, I am very grateful for my family. It has been a rough time, and I was not able to be with them due to COVID-19 when they needed me the most. Nevertheless, they still provide me with unconditional support and love.

VITA

2015–2019 B.S. (Mathematics) and minor (Statistics), University of California, Santa Barbara

2019–2021 M.A. (Mathematics), University of California, Los Angeles

2019–2022 Teaching assistant in Mathematics Department, University of California, Los Angeles

2019–present Graduate Research assistant in Mathematics Department, University of California, Los Angeles

PUBLICATIONS

Justin Baker*, Hedi Xia*, Yiwei Wang, Elena Cherkaev, Akil Narayan, Long Chen, Jack Xin, Andrea Bertozzi, Stanley Osher, Bao Wang. Proximal Implicit ODE Solvers for Accelerating Learning Neural ODEs. Submitted. (* co-first author)

Matthew Thorpe*, Tan Nguyen*, Hedi Xia*, Thomas Strohmer, Andrea Bertozzi, Stanley Osher, Bao Wang. GRAND++: Graph Neural Diffusion with A Source Term. International Conference on Learning Representations, 2022. (* equal contribution)

Bao Wang, Hedi Xia, Tan Nguyen, Stanley Osher. How Does Momentum Benefit Deep Neural Networks Architecture Design? A Few Case Studies. Research in Mathematical Sciences, 9 (3) 1-37, 2022.

Hedi Xia*, Vai Suliafu*, Tan Nguyen, Hangjie Ji, Andrea Bertozzi, Stanley Osher and Bao Wang. Heavy Ball Neural Ordinary Differential Equations. Neural Information Processing Systems, 2021. (* co-first author)

CHAPTER 1

Introduction

In recent years, neural networks have been widely used to model a variety of problems, such as computer vision [He+16a; Wan+22; Sha+22], natural language processing [Vas+17; Wan+22; LHE22], and recommendation systems [PAC18; Pal+20; ASS20]. Challenging datasets, such as ImageNet [Rus+15], call for more complicated neural networks [ZF13]. Thus, by increasing the number of stacked layers (depth), deep neural networks were invented. Deeper neural networks can better incorporate multi-level features [ZF13], and as a result, they show promising performance and robustness, while also increasing the complexity of model training and inference [SZ15; Sze+15; He+15; IS15]. Thus, the depth of neural networks becomes an important hyper-parameter to study [SZ15; Sze+15]. The promising performances of deep neural networks give rise to a question: can we construct neural networks with infinite depth? As the general designs of deep neural networks do not converge as depth tends to infinity, various classes of deep neural networks with convergence guaranteed designs have been studied. Deep Equilibrium Models [BKK19] study the limits of deep neural networks that follow the contraction mapping theorem. Differentiable Convex Optimization Layers [Agr+19] study the limits of deep neural networks where each layer acts as an optimization step of some scalar-value function, and Neural Ordinary Differential Equations (NODEs) [Che+18] study the limit of deep neural networks as Euler discretization of ordinary differential equations. The focus of this thesis is to discuss a set of approaches to improve the performance and reduce computation cost in training and inference of NODEs.

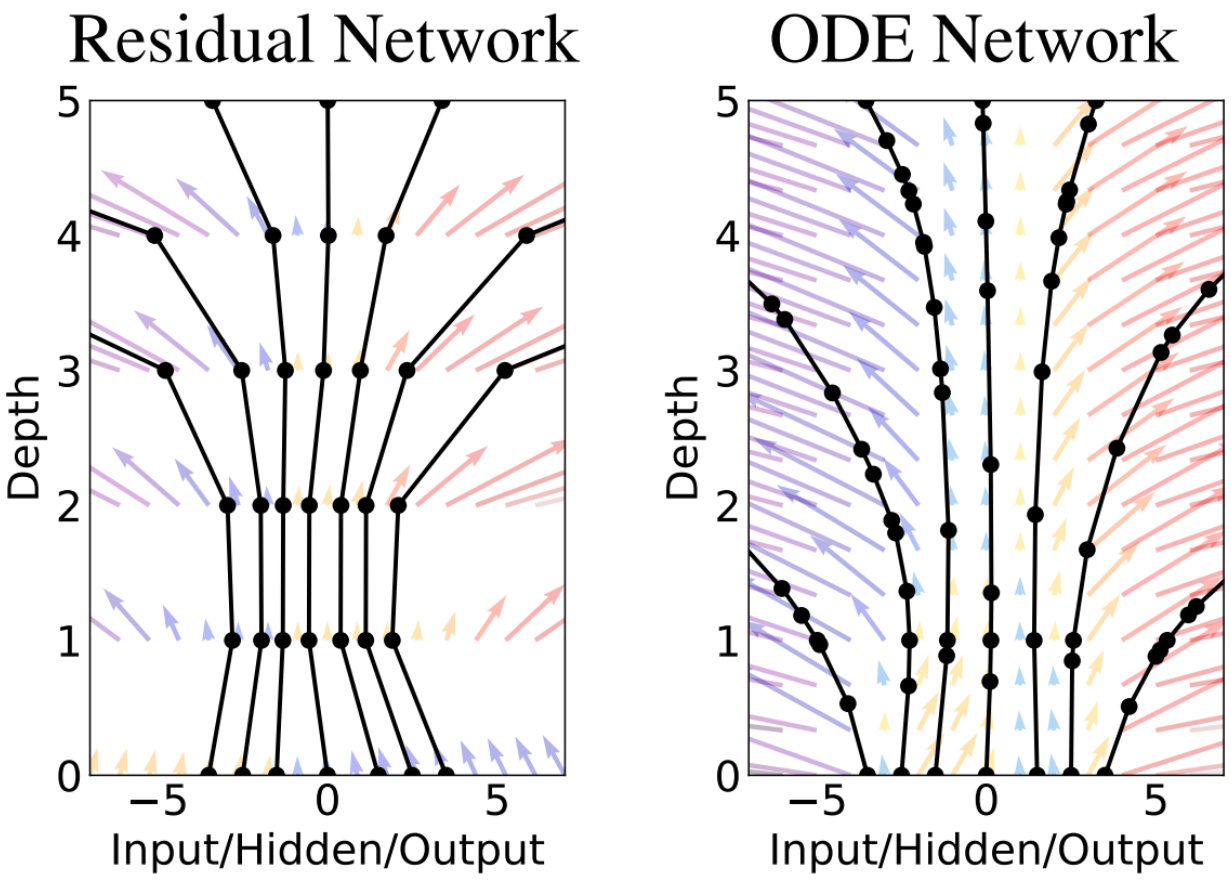


Figure 1.1: Comparison between ResNets (left) and NODEs (right). NODEs learn smooth trajectories whereas ResNets learn discrete dynamics. Image from Neural Ordinary Differential Equations [Che+18].

1.1 Neural Ordinary Differential Equations

NODEs [Che+18] takes its name from ordinary differential equations (1.1). For decades, differential equations have been the center of mathematical modeling for various application fields, such as physics [Ghi+81; DP78; PP10], biology [Ahm+20; DHS11; XGC20], and finance [TT13; RG15; GRK10]. Differential equations vary in their structure, including ordinary differential equations, partial differential equations, and stochastic differential equations.

There have been extensive studies on their well-posedness [Hil13], dynamical systems [For98], and numerical solvers [BCP95]. NODEs [Che+18] combine deep learning and ordinary differential equations by solving the θ-parametrized initial value problem

$$\frac{d\boldsymbol{h}}{dt}(t) = f(\boldsymbol{h}(t), t, \theta), \text{ for } t \in [t_0, T]. \tag{1.1}$$

Thus, NODEs can be considered as a neural network layer with $\boldsymbol{h}(t_0)$ as the input, $\boldsymbol{h}(T)$ as the output, and θ as the set of parameters. The Euler discretization of NODEs with step size $\frac{T-t_0}{N}$ takes the form

$$\boldsymbol{h}\left(t + \frac{1}{N}\right) = \boldsymbol{h}(t) + \frac{1}{N} f(\boldsymbol{h}(t), t, \theta), \text{ for } t = t_0 + \frac{k}{N},\ k = 0, 1, ..., N-1, \tag{1.2}$$

with input $\boldsymbol{h}(t_0)$ and output $\boldsymbol{h}(T)$, respectively. The Euler discretization is a deep neural network with N layers, and more precisely, it is a class of Residual Networks (ResNets) [He+16a] according to the residual patterns of each block. Therefore, when the limit of Euler discretization exists, the sequence of ResNets as the number of steps/depth goes to infinity will converge to NODEs.

Compared to other deep learning models, NODEs greatly benefit from their ODEs structure. Since trajectories of ODEs cannot intersect, NODEs are guaranteed to be a continuous bijection whose inverse can be calculated through solving the exact same equation backward in time. This property enables memory-efficient adjoint methods for gradient computation. The adjoint method solves the adjoint ODEs

$$\frac{d\boldsymbol{r}}{dt}(t) = -\boldsymbol{r}(t)\frac{\partial f}{\partial \boldsymbol{h}}(\boldsymbol{h}(t), t, \theta), \text{ for } t \in [t_0, T], \tag{1.3}$$

with a terminal condition of $\boldsymbol{r}(T) = \frac{d\mathcal{L}}{d\boldsymbol{h}(T)}$ backward in time. As the adjoint ODEs (1.3) require evaluation of $\boldsymbol{h}(t)$, the adjoint method reconstructs $\boldsymbol{h}(t)$ by solving equation (1.1) but with a terminal condition of $\boldsymbol{h}(T)$ and backward in time [Che+18]. Continuous bijection functions are also topology-preserving, and thus NODEs can be applied to generative modeling by creating continuous normalizing flows [Gra+18]. By evaluating $\boldsymbol{h}(t)$ at multiple times t, NODEs also naturally apply to time series, and in particular, irregularly sampled time

series, which can be challenging for traditional time series models [RCD19]. As shown in figure 1.1, NODEs also have smoother trajectories compared to their discrete counterpart, the Residual Networks (ResNets). For problems with smooth trajectories or even with physical laws, NODEs approximate the underlying differential equations better in both interpolation and extrapolation [ZDC20; LP21]. NODEs also generalize to a broader context beyond ODEs, such as control differential equations [Kid+20], stochastic differential equations [Liu+19; JB19; Son+20], and differential equations on graphs [Cha+21a; Cha+21b].

1.1.1 Challenges with NODEs

The benefits of NODEs also come with challenges. One of the major concerns with NODEs is the number of function evaluations (NFEs) problem [Che+18]. Compared to deep neural networks with finite depth, where computation and memory costs are proportional to their depth, NODEs are implicitly defined by an initial value problem with no bound on their computation and memory costs. As general nonlinear ODEs do not have general solutions, numerical solvers are needed to approximate the solutions of the NODEs. NFEs are defined as the number of function evaluations of f in equation (1.1) needed by the solver [Che+18]. As the computation cost of the neural network f generally dominates the computation cost of other operations in numerical solvers, NFEs are roughly proportional to the computation time of NODEs' forward pass, as well as memory usage when the adjoint method is not used. Under a given tolerance, the number of function evaluations depends on the maximum step size the solver can take, which is largely dependent on the dynamics of the ODEs and could potentially grow unbounded during training. Large NFEs could lead to long computation time, as well as unreliable results, as numerical error could potentially be amplified during successive computation.

There are many causes of the NFEs problems, and one of them is the topological-preserving property of ODEs. Dupont et al [DDT19] show in Figure 1.2 that if there are topological differences between the input space and output features, NODEs may force topology change by creating large distortions in space through gaps in the dataset. During training, this can

cause the learned NODEs to become stiff and hard to solve, leading to a large increase in NFEs and slow computation. The NFEs problems can also occur when computing gradients with the adjoint method. Although the adjoint equation (1.3) is linear in $\boldsymbol{r}$ and thus easier to compute, backward solving the state equation (1.1) can be difficult or even harder than the forward pass. For example, when the forward initial value problem approximates diffusion equations, solving backward-time diffusion equations is not possible as it is not a well-posed problem, making inverting the ODEs numerically unstable.

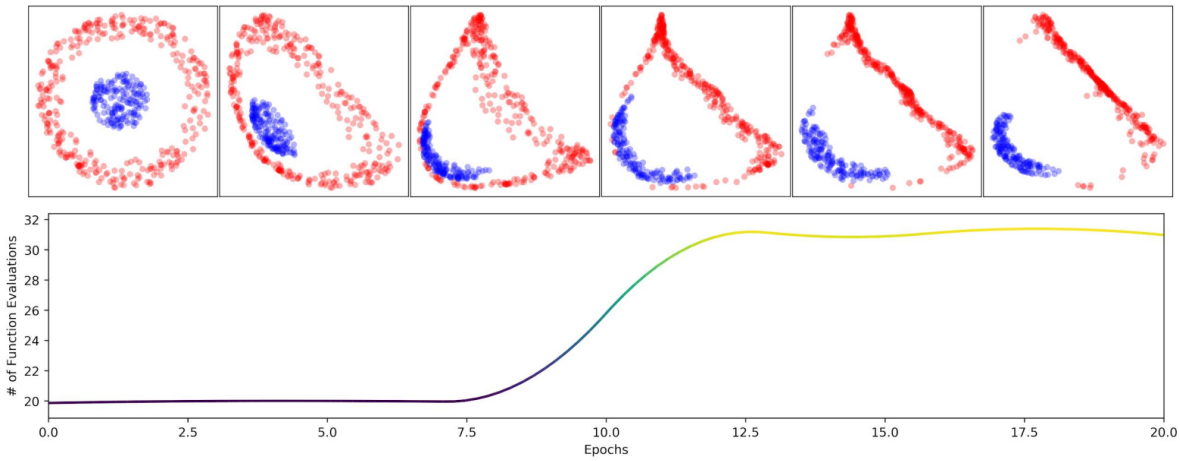


Figure 1.2: Change of dynamics of circles during training. To classify blue and red parts, NODEs needs to transform them into linearly separable parts. Because NODEs preserve topology, it needs to break through the circle, resulting in increase in NFEs. Image from Augmented Neural ODEs [DDT19].

Another concern with NODEs is the vanishing gradient problem. The vanishing gradient problem occurs during the training of deep neural networks with gradient-based methods [Hoc+01], which can lead to stopped convergence from the start [GB10; Hoc91]. When training deep neural networks, computing the gradient with the chain rule requires back-propagation step by step through all layers. This successive operation might lead to a diminishing size of the final gradients, which can cause the gradient-based optimizer to fail in

accurately estimating the gradient and taking proper steps to train the model.

Since NODEs are unbounded in their depth and complexity, they could suffer from the vanishing gradient problem during the training phase. There is also the issue of the exploding gradient problem, where the final gradients become too large for optimizers. As the directions of the exploding gradients are still reliable, gradient clipping [Sze+16] is proposed to solve the problem by normalizing the gradients. However, the vanishing gradient problem does not have a similar general solution because when the gradients are smaller than the floating point error, the gradient no longer provides any valid information. Therefore, we are interested in architectures that can alleviate the vanishing gradient problem in modeling with NODEs.

In this thesis, we will look into model designs and numerical solvers to alleviate the NFEs problem and the vanishing gradient problem for general NODEs in chapter 3 and 5.

1.2 Graph Neural Diffusion

Graphs are inherent structures in many datasets, such as co-purchase graphs in Amazon Photos datasets [Shc+18], citation graphs in Pubmed datasets [Nam+12], knowledge graphs in WikiGraphs dataset [Wan+21a], social networks in Reddit dataset [PAS17], and protein structure graphs in Enzymes dataset [DD03]. The nodes of a graph usually represent an item and are typically provided with an associated representation vector, while edges indicate the relationships between the items represented by the two connected nodes and could be directed, undirected, or even carry a vector representing the properties of the relation. Graph learning problems generally fall into three categories: node prediction, which predicts the properties of each node; edge prediction, which predicts the existence and properties of edges; and graph prediction, which predicts the properties of the entire graph.

One mathematical approach to solving graph learning problems is Spectral Graph Theory [Chu], which studies the properties of the adjacency matrix and Laplacian matrix of the graph. By studying the Laplace operator on graphs [Moh+91], partial differential equations with space derivatives in the form of Laplacian operators can be generalized to graphs, such

as Laplace equations, Poisson equations, and Diffusion equations. Based on these graph differential equations, graph learning algorithms like Laplace learning [ZGL03], Poisson learning [Cal+20], and the graph Merriman–Bence–Osher (MBO) scheme [MKB13] have been designed.

In deep learning, Message-Passing neural networks are used to solve graph learning problems, such as Graph Convolutional Neural Network (GCN) [Zha+19a], Graph Attention Network (GAT) [Vel+], and GraphSAGE [HYL17]. Each layer of a message passing neural network consists of an aggregation step, which collects information from adjacent nodes, and an update step, which updates the vector representation of each node with information from the aggregation step using some neural network. Compared to spectral models, these models better utilize feature vectors on nodes and edges and are generally shallower. In fact, in the experiments with GAT by Veličković et al [Vel+], only 2 to 3 layers of GAT are used, and in the experiments with GCN by Yao et al [YML19], only 2 layers of GCN are used. Some experiments show that adding more layers does not improve performance [KW17; LHW18]. While shallow graph neural networks cannot capture relations between nodes with a distance greater than their depth, deep graph neural networks, achieved by simply stacking layers, result in large memory costs and worse performance due to over-smoothing [Che+20].

Graph Neural Diffusion (GRAND) [Cha+21a] is a class of NODEs [Che+18] on graphs that combines a neural network with the graph diffusion equation by estimating the diffusivity term in the Perona-Malik diffusion equation [PM90] using a graph attention network [Vel+] with transformer-type attention [Vas+17]. This allows GRAND to nicely utilize feature vectors on nodes like other graph neural networks while also achieving a depth greater than 2 to 3 layers, resulting in improved performance on various tasks. The NODEs nature of GRAND also enables the adjoint method during training, which alleviates memory requirements for deep neural networks.

However, GRAND still suffers from the over-smoothing problem [Che+20]. In particular, on sparsely labeled graphs, some unlabeled nodes are distant from all of the labeled ones and

require a very deep graph neural network for the node to gather any label information from the graph. In this thesis, we will discuss improvements on GRAND that can enable even deeper modeling of sparsely labeled graphs in Chapter 4.

CHAPTER 2

Background

2.1 Residual Networks (ResNets)

Residual Networks [He+16a] are a class of deep neural networks [KSH17] with shortcut connections. A cell in a residual network is a layer $\mathbb{R}^m \times \mathbb{R}^r \to \mathbb{R}^m$ such that

$$\boldsymbol{y} = \boldsymbol{x} + f(\boldsymbol{x}, \theta), \tag{2.1}$$

where f is some neural network cell, $\boldsymbol{x}$ is the input, θ is the parameter, and $\boldsymbol{y}$ is the output. A residual network with N cells is a successive operation from input $\boldsymbol{x}_0$ to output $\boldsymbol{x}_N$, such that

$$\boldsymbol{x}_i = \boldsymbol{x}_{i-1} + f_i(\boldsymbol{x}_{i-1}, \theta), \quad i = 1, \ldots, N, \tag{2.2}$$

where f_i are neural network cells corresponding to each residual cells. The gradients of ResNets satisfy the successive relation

$$\frac{d\mathcal{L}}{d\boldsymbol{x}_{i-1}} = \frac{d\mathcal{L}}{d\boldsymbol{x}_i} + \frac{d\mathcal{L}}{d\boldsymbol{x}_i}\frac{\partial f_i}{\partial \boldsymbol{x}}(x_{i-1}, \theta). \tag{2.3}$$

This implies the overall relation

$$\frac{d\mathcal{L}}{d\boldsymbol{x}_1} = \frac{d\mathcal{L}}{d\boldsymbol{x}_N}\prod_{i=1}^{N-1}\left(\boldsymbol{I} + \frac{\partial f_i}{\partial \boldsymbol{x}}(x_i, \theta)\right), \tag{2.4}$$

where $\boldsymbol{I}$ denotes the identity matrix. Because of the design of the skip connection, an extra identity matrix term is included during gradient computation, making the multiplication less likely to vanish. Thus, ResNets are easier to train compared to plain deep neural networks due to fewer issues with vanishing gradients.

2.2 Neural Ordinary Differential Equations (NODEs)

Similar to ResNets [He+16a], NODEs [Che+18] are also defined based on a neural network cell f. Let $\mathbb{R}^m$ be the hidden space and $\mathbb{R}^r$ be the parameter space. We define $f : \mathbb{R}^m \times \mathbb{R} \times \mathbb{R}^r \to \mathbb{R}^m$ as a continuously differentiable neural network (see 3.2.2.3 for Lipchitz continuous generalization). Given an interval $[t_0, T]$, a Neural Ordinary Differential Equations (NODE) layer is defined as $F : \mathbb{R}^m \times \mathbb{R}^r \to \mathbb{R}^m$, where the output is the terminal value

$$F(\boldsymbol{x}, \theta) := \boldsymbol{h}(T). \tag{2.5}$$

of the initial value problem satisfying

$$\begin{aligned} &\boldsymbol{h}(t_0) = \boldsymbol{x}, \\ &\frac{d\boldsymbol{h}}{dt}(t) = f(\boldsymbol{h}(t), t, \theta), \quad \forall t \in [t_0, T]. \end{aligned} \tag{2.6}$$

The definition (2.5) is well-defined if equation (2.6) has a unique solution for all $\boldsymbol{x} \in \mathbb{R}^m$ and $\theta \in \mathbb{R}^r$, which can be guaranteed by Lipchitz assumptions on f (detailed proofs in 2.2.1).

NODEs can be seen as a continuous limit of a particular classes of ResNets. Suppose layers of ResNets satisfy

$$\frac{1}{N} f_i(\boldsymbol{x}_{i-1}, \theta) = f\Big(\boldsymbol{x}_{i-1}, \frac{i}{N}, \theta\Big) \tag{2.7}$$

Then equation (2.2) becomes

$$\frac{\boldsymbol{x}_i - \boldsymbol{x}_{i-1}}{N} = \frac{1}{N} f_i(\boldsymbol{x}_{i-1}, \theta) = f\Big(\boldsymbol{x}_{i-1}, \frac{i}{N}, \theta\Big). \tag{2.8}$$

If we take the limit as $N \to +\infty$, we arrive at

$$\frac{d\boldsymbol{x}}{dt}(t) = f(\boldsymbol{x}(t), t, \theta), \tag{2.9}$$

which is equivalent to NODEs equation (2.6).

There is no analytical solution to Equation (2.6) for a generic neural network f. Therefore, NODEs use numerical solvers for ODEs to solve the initial value problem (2.6) during training and inference. Generic ODE solvers, such as Runge-Kutta methods [RK95], or those tailored

to the structure of f, can be used for solving the equation. When the Forward Euler method is used, NODEs degenerate into ResNets [He+16a]. In the majority of experiments [Che+18], a 5th-order Dormand-Prince method [DP80b], which is a class of multi-step Runge-Kutta methods, is used for training and inference of NODEs as default, unless explicitly noted.

2.2.1 Proof of Existence and Uniqueness of Solutions of NODEs

In this section, we will review a direct corollary of Picard Lindelöf theorem that can be used to guarantee that NODEs are well defined by the initial value problem (2.6). Suppose $f : \mathbb{R}^m \times [t_0, T] \to \mathbb{R}^m$ (with fixed parameter θ) is uniformly Lipchitz continuous with Lipchitz constant L, then there exists an unique solution to the equation

$$\frac{d\boldsymbol{h}}{dt}(t) = f(\boldsymbol{h}(t), t), \ \ for\ t \in [t_0, T], \tag{2.10}$$

with initial condition $\boldsymbol{h}(t_0) = \boldsymbol{h}_0$. Compared to Picard Lindelöf theorem that guarantees local existence and uniqueness properties, to obtain global existence and uniqueness property, we make further assumption on uniformly Lipchitz condition. This assumptions is reasonable for neural networks, as a majority of neural networks, such as multiple layer perceptrons, convolutional neural networks, attention neural networks, are all uniformly Lipchitz continuous. The proof follows similar pattern as that of Picard Lindelöf. Define operator Γ as

$$\Gamma\boldsymbol{h}(t) = \boldsymbol{h}_0 + \int_{t_0}^{t} f(\boldsymbol{h}(s), s)ds. \tag{2.11}$$

We will prove by induction on the following inequality

$$\left\|\Gamma^k\phi_1(t) - \Gamma^k\phi_2(t)\right\| \leq \frac{L^k(t-t_0)^k}{k!}\left\|\phi_1 - \phi_2\right\|_\infty, \tag{2.12}$$

for all $\phi_1, \phi_2 : \mathbb{R}^m \times [t_0, T] \to \mathbb{R}^m$ such that $\phi_1(t_0) = \phi_2(t_0) = \boldsymbol{h}_0$. For $k = 0$, it follows by definition on the infinity norm. For general k, we have

$$\left\|\Gamma^k\phi_1(t) - \Gamma^k\phi_2(t)\right\| = \left\|\int_{t_0}^{t}\left(f(\Gamma^{k-1}\phi_1(s), s) - f(\Gamma^{k-1}\phi_2(s), s)\right)ds\right\|. \tag{2.13}$$

By Lipchitz continuity of f, we have

$$\left\|\int_{t_0}^{t}\left(f(\Gamma^{k-1}\phi_1(s), s) - f(\Gamma^{k-1}\phi_2(s), s)\right)ds\right\| \leq \left\|\int_{t_0}^{t} L\left(\Gamma^{k-1}\phi_1(s) - \Gamma^{k-1}\phi_2(s)\right)ds\right\|. \tag{2.14}$$

By induction hypothesis (2.12), we have

$$\left\|\Gamma^{k-1}\phi_1(s) - \Gamma^{k-1}\phi_2(s)\right\| \leq \frac{L^{k-1}(s-t_0)^{k-1}}{(k-1)!}\left\|\phi_1 - \phi_2\right\|_\infty . \tag{2.15}$$

Therefore, combining these inequalities, we have

$$\left\|\Gamma^{k}\phi_1(t) - \Gamma^{k}\phi_2(t)\right\| \leq L\int_{t_0}^{t} \frac{L^{k-1}(s-t_0)^{k-1}}{(k-1)!}\left\|\phi_1 - \phi_2\right\|_\infty ds = \frac{L^{k}(t-t_0)^{k}}{k!}\left\|\phi_1 - \phi_2\right\|_\infty , \tag{2.16}$$

which proves the inequality through induction. A direct corollary of the induction is that

$$\left\|\Gamma^{k}\phi_1 - \Gamma^{k}\phi_2\right\|_\infty \leq \frac{L^{k}(T-t_0)^{k}}{k!}\left\|\phi_1 - \phi_2\right\|_\infty . \tag{2.17}$$

For sufficiently large M, for every $k > M$, $\frac{L^k(T-t_0)^k}{k!} < 1$, and thus Γ^k is a contraction mapping. By contraction mapping theorem, there exists an unique fix point. Let $\boldsymbol{h}_k, \boldsymbol{h}_{k+1}, \boldsymbol{h}_{k(k+1)}$ be the unique fix point for contraction mappings $\Gamma^k, \Gamma^{k+1}, \Gamma^{k(k+1)}$. Notice that both $\boldsymbol{h}_k$ and $\boldsymbol{h}_{k+1}$ has to be fix point of contraction mapping $\Gamma^{k(k+1)}$, and because $\boldsymbol{h}_{k(k+1)}$ is unique, these three fix point has to be the same. Thus

$$\boldsymbol{h}_k = \boldsymbol{h}_{k+1} = \Gamma^{k+1}\boldsymbol{h}_{k+1} = \Gamma\Gamma^{k}\boldsymbol{h}_k = \Gamma\boldsymbol{h}_k, \tag{2.18}$$

which means that $\boldsymbol{h}_k$ is a fix point of Γ. Also, it is the unique fix point of Γ because every fix point of Γ has to be fix point of Γ^k and thus has to be unique. Therefore, there exists an unique solution to the equation

$$\boldsymbol{h}(t) = \boldsymbol{h}_0 + \int_{t_0}^{t} f(\boldsymbol{h}(s), s)ds, \tag{2.19}$$

which implies that (2.6) is well-defined.

2.2.2 Bijection Property of NODEs

The bijective property of NODEs allows for the use of adjoint method for gradient computation (details in 2.2.3) and the application of continuous normalizing flows [Che+18]. To prove that NODEs with uniformly Lipchitz condition is a bijection between the input

$\boldsymbol{h}(t_0)$ and output $\boldsymbol{h}(T)$, we simply need to find its inverse. If we define the forward function that solves

$$\frac{d\boldsymbol{h}}{dt}(t) = f(\boldsymbol{h}(t), t, \theta), \ \ for\ t \in [t_0, T], \tag{2.20}$$

with initial condition $\boldsymbol{h}(t_0) = \boldsymbol{h}_0$ be the input and $F(\boldsymbol{h}_0) = \boldsymbol{h}(T)$ be the output, the inverse function can be defined as that solves the same equation with terminal condition $\boldsymbol{h}(T) = \boldsymbol{h}_T$ be the input and $F^{-1}(\boldsymbol{h}_T) = \boldsymbol{h}(t_0)$ be the output. In section 2.2.1, we have already proved that the forward initial value problem has an unique solution under Lipchitz continuous f. To prove that the terminal value problem also has an unique solution, consider the mirror initial value problem as

$$\frac{d\boldsymbol{g}}{d\tau}(\tau) = -f(\boldsymbol{g}(\tau), \tau, \theta), \ \ for\ \tau \in [-T, -t_0], \tag{2.21}$$

with initial condition $\boldsymbol{g}(-T) = \boldsymbol{h}_T$. Since $-f$ has the same Lipchitz constant as f, the initial value problem has an unique solution. Therefore, by the equivalence $\boldsymbol{h}(t) = \boldsymbol{g}(-\tau)$, we know that the terminal value problem has an unique solution as well. Notice that if $F(\boldsymbol{h}_0) = \boldsymbol{h}_T$, then the solution to the forward initial value problem is also a solution to the terminal value problem. Since the solution to the terminal value problem is unique, we have $F^{-1}(\boldsymbol{h}_T) = \boldsymbol{h}(t_0) = \boldsymbol{h}_0$, which means that the terminal value problem defines the inverse of the forward initial value problem. Thus, NODEs is a bijection with an inverse implicitly defined by another set of ODEs.

2.2.3 Adjoint Solutions

Computing gradients for Neural ODEs can be done by Automatic Differentiation [Pas+17] through its forward numerical solver. However, similar to other deep neural networks, such methods require storing all the intermediate steps in order to compute gradients backwards. This leads to prohibitively large memory costs when both the hidden dimension m and the number of steps used are large. Chen et al [Che+18] proposed an adjoint method to compute the gradient as follows: Suppose a set of NODEs satisfies the equation (2.6), and $\mathcal{L}$ is the

loss depending on the output of NODEs, then by chain rule,

$$\frac{d\mathcal{L}}{d\boldsymbol{h}(t)} = \frac{d\mathcal{L}}{d\boldsymbol{h}(t+\Delta t)}\frac{d\boldsymbol{h}(t+\Delta t)}{d\boldsymbol{h}(t)}. \tag{2.22}$$

To compute $\frac{d\boldsymbol{h}(t+\Delta t)}{d\boldsymbol{h}(t)}$, we can rewrite equation (2.6) in integral form

$$\boldsymbol{h}(t+\Delta t) = \boldsymbol{h}(t) + \int_t^{t+\Delta t} f(\boldsymbol{h}(s), s, \theta)ds. \tag{2.23}$$

Assuming f is analytic, we have the approximation

$$f(\boldsymbol{h}(s), s, \theta) = f(\boldsymbol{h}(t), t, \theta) + O(\Delta t). \tag{2.24}$$

Substituting into equation (2.23), we have

$$\boldsymbol{h}(t+\Delta t) = \boldsymbol{h}(t) + \Delta t(f(\boldsymbol{h}(t), t, \theta) + O(\Delta t)) = \boldsymbol{h}(t) + \Delta t f(\boldsymbol{h}(t), t, \theta) + O(\Delta t^2). \tag{2.25}$$

Taking derivative with respect to $\boldsymbol{h}(t)$ with analytic assumption on f yields

$$\frac{d\boldsymbol{h}(t+\Delta t)}{d\boldsymbol{h}(t)} = \mathbf{I} + \Delta t\frac{\partial f}{\partial \boldsymbol{h}}(\boldsymbol{h}(t), t, \theta) + O(\Delta t^2). \tag{2.26}$$

Define gradient $\boldsymbol{r}$ as

$$\boldsymbol{r}(t) = \frac{d\mathcal{L}}{d\boldsymbol{h}(t)}, \tag{2.27}$$

we can rewrite equation (2.22) as

$$\boldsymbol{r}(t) = \boldsymbol{r}(t+\Delta t)\frac{d\boldsymbol{h}(t+\Delta t)}{d\boldsymbol{h}(t)}. \tag{2.28}$$

If we consider the limiting behavior

$$\frac{d\boldsymbol{r}}{dt}(t) = \lim_{\Delta t\to 0}\frac{\boldsymbol{r}(t+\Delta t) - \boldsymbol{r}(t)}{\Delta t} = \lim_{\Delta t\to 0}\frac{\boldsymbol{r}(t+\Delta t) - \boldsymbol{r}(t+\Delta t)\frac{d\boldsymbol{h}(t+\Delta t)}{d\boldsymbol{h}(t)}}{\Delta t}, \tag{2.29}$$

we can combine with equation (2.26) and get the adjoint equations

$$\frac{d\boldsymbol{r}}{dt}(t) = \lim_{\Delta t\to 0}\frac{\boldsymbol{r}(t+\Delta t)(-\Delta t\frac{df}{d\boldsymbol{h}}(\boldsymbol{h}(t), t, \theta) - O(\Delta t^2))}{\Delta t} = -\boldsymbol{r}(t)\frac{\partial f}{\partial \boldsymbol{h}}(\boldsymbol{h}(t), t, \theta). \tag{2.30}$$

This leads to the following reverse-time initial value problem

$$\begin{aligned}
\boldsymbol{h}(T) &= F(\boldsymbol{x}, \theta), \\
\boldsymbol{r}(T) &= \frac{d\mathcal{L}}{d\boldsymbol{h}(T)}, \\
\frac{d\boldsymbol{h}}{dt}(t) &= f(\boldsymbol{h}(t), t, \theta), \ \forall t \in [t_0, T], \\
\frac{d\boldsymbol{r}}{dt}(t) &= -\boldsymbol{r}(t)\frac{\partial f}{\partial \boldsymbol{h}}(\boldsymbol{h}(t), t, \theta), \ \forall t \in [t_0, T],
\end{aligned} \tag{2.31}$$

where the terminal conditions $F(\boldsymbol{x}, \theta)$ is computed and stored while solving the forward equation, $\frac{d\mathcal{L}}{d\boldsymbol{h}(T)}$ is computed through automatic differentiation of loss function, and the vector-Jacobi product $\boldsymbol{r}(t)\frac{df}{d\boldsymbol{h}}$ is computed through automatic differentiation through neural network f. By solving the adjoint equations (2.31), we can directly obtain the gradient at initial value $\frac{d\mathcal{L}}{d\boldsymbol{h}(t_0)}$. To obtain the gradient with respect to parameter θ, we define $\tilde{\boldsymbol{h}}$ as the concatenation of $\boldsymbol{h}$ and θ

$$\tilde{\boldsymbol{h}} := \begin{bmatrix} \boldsymbol{h} \\ \theta \end{bmatrix}, \tag{2.32}$$

and the ODEs become

$$\frac{d\tilde{\boldsymbol{h}}}{dt} = \begin{bmatrix} \frac{d\boldsymbol{h}}{dt} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} f(\boldsymbol{h}(t), t, \theta) \\ \mathbf{0} \end{bmatrix}. \tag{2.33}$$

Then the corresponding $\tilde{\boldsymbol{r}}$ will satisfy equation (2.31)

$$\frac{d\tilde{\boldsymbol{r}}}{dt}(t) = -\tilde{\boldsymbol{r}}(t) \begin{bmatrix} \frac{\partial f}{\partial \boldsymbol{h}}(\boldsymbol{h}(t), t, \theta) & \frac{\partial f}{\partial \theta}(\boldsymbol{h}(t), t, \theta) \\ \mathbf{0} & \mathbf{0} \end{bmatrix}. \tag{2.34}$$

By definition of $\tilde{\boldsymbol{r}}$,

$$\tilde{\boldsymbol{r}}(t) = \begin{bmatrix} \frac{d\mathcal{L}}{d\boldsymbol{h}}(t) & \frac{d\mathcal{L}}{d\theta}(t) \end{bmatrix} = \begin{bmatrix} \boldsymbol{r}(t) & \frac{d\mathcal{L}}{d\theta}(t) \end{bmatrix}, \tag{2.35}$$

we can combine equations (2.34) and (2.35), we arrive at

$$\frac{d^2\mathcal{L}}{dtd\theta} = -\boldsymbol{r}(t)\frac{\partial f}{\partial \theta}(\boldsymbol{h}(t), t, \theta). \tag{2.36}$$

Which integrates against t provides us with

$$\frac{d\mathcal{L}}{d\theta}(t_0) - \frac{d\mathcal{L}}{d\theta}(T) = -\int_T^{t_0} \boldsymbol{r}(t)\frac{\partial f}{\partial \theta}(\boldsymbol{h}(t), t, \theta)dt. \tag{2.37}$$

By definition, $\frac{d\mathcal{L}}{d\theta} = \frac{d\mathcal{L}}{d\theta}(t_0)$, and $\frac{d\mathcal{L}}{d\theta}(T) = 0$, providing the equation

$$\frac{d\mathcal{L}}{d\theta} = -\int_T^{t_0} \boldsymbol{r}(t)\frac{\partial f}{\partial \theta}(\boldsymbol{h}(t), t, \theta)dt. \tag{2.38}$$

2.2.4 ODE-RNN

ODE-RNN is a combination of NODEs and Recurrent Neural Network (RNN) to model time series. To model a time series $\{(t_i, \boldsymbol{x}_i), i = 0, ..., N\}$ with $t_0 < t_1 < \cdots < t_N \leq T$, ODE-RNN uses a NODEs cell for progression in time that is defined except the nodes points

$$\frac{d\boldsymbol{h}}{dt}(t) = f_{NODEs}(\boldsymbol{h}(t), t, \theta), \ for\ t \in [t_0, T],\ t \neq t_i\ for\ i = 0, ..., N, \tag{2.39}$$

and an RNN cell that defines the jump on those nodes due to observation

$$\boldsymbol{h}^+(t_i) = f_{RNN}(\boldsymbol{h}^-(t_i), \boldsymbol{x}_i, \theta), \ for\ i = 0, ..., N. \tag{2.40}$$

For each interval $[t_i, t_{i+1}]$, ODE-RNN solves its NODEs cell with initial condition $\boldsymbol{h}^+(t_i)$ and obtains $\boldsymbol{h}^-(t_{i+1})$. It then incorporates observation $\boldsymbol{x}_{i+1}$ within the RNN cell and obtains $\boldsymbol{h}^+(t_{i+1})$, which is used as the input for the next interval. ODE-RNN uses $\boldsymbol{h}^+(t_i)$ as its hidden layer output and can be stacked with further layers to produce final outputs. For both interpolation ($t \in [t_0, T]$) and extrapolation tasks ($t \in [T, +\infty)$), ODE-RNN can make predictions and forecasts using the hidden information $\boldsymbol{h}^+(t)$ if $\boldsymbol{x}(t)$ is provided or $\boldsymbol{h}(t)$ otherwise.

CHAPTER 3

Heavy Ball Neural Ordinary Differential Equations (HBNODEs)

3.1 Introduction

Neural ordinary differential equations (NODEs) are a family of continuous-depth machine learning (ML) models whose forward and backward propagations rely on solving an ODE and its adjoint equation [Che+18]. NODEs model the dynamics of hidden features $\boldsymbol{h}(t) \in \mathbb{R}^N$ using an ODE, which is parametrized by a neural network $f(\boldsymbol{h}(t), t, \theta) \in \mathbb{R}^N$ with learnable parameters θ, i.e.,

$$\frac{d\boldsymbol{h}(t)}{dt} = f(\boldsymbol{h}(t), t, \theta). \tag{3.1}$$

Starting from the input $\boldsymbol{h}(t_0)$, NODEs obtain the output $\boldsymbol{h}(T)$ by solving (3.1) for $t_0 \leq t \leq T$ with the initial value $\boldsymbol{h}(t_0)$, using a black-box numerical ODE solver. The number of function evaluations (NFEs) that the black-box ODE solver requires in a single forward pass is an analogue for the continuous-depth models [Che+18] to the depth of networks in ResNets [He+16a]. The loss between NODE prediction $\boldsymbol{h}(T)$ and the ground truth is denoted by $\mathcal{L}(\boldsymbol{h}(T))$; we update parameters θ using the following gradient

$$\frac{d\mathcal{L}(\boldsymbol{h}(T))}{d\theta} = \int_{t_0}^{T} \boldsymbol{a}(t) \frac{\partial f(\boldsymbol{h}(t), t, \theta)}{\partial \theta} dt, \tag{3.2}$$

where $\boldsymbol{a}(t) := \partial \mathcal{L} / \partial \boldsymbol{h}(t)$ is the adjoint state, which satisfies the following adjoint equation

$$\frac{d\boldsymbol{a}(t)}{dt} = -\boldsymbol{a}(t) \frac{\partial f(\boldsymbol{h}(t), t, \theta)}{\partial \boldsymbol{h}}. \tag{3.3}$$

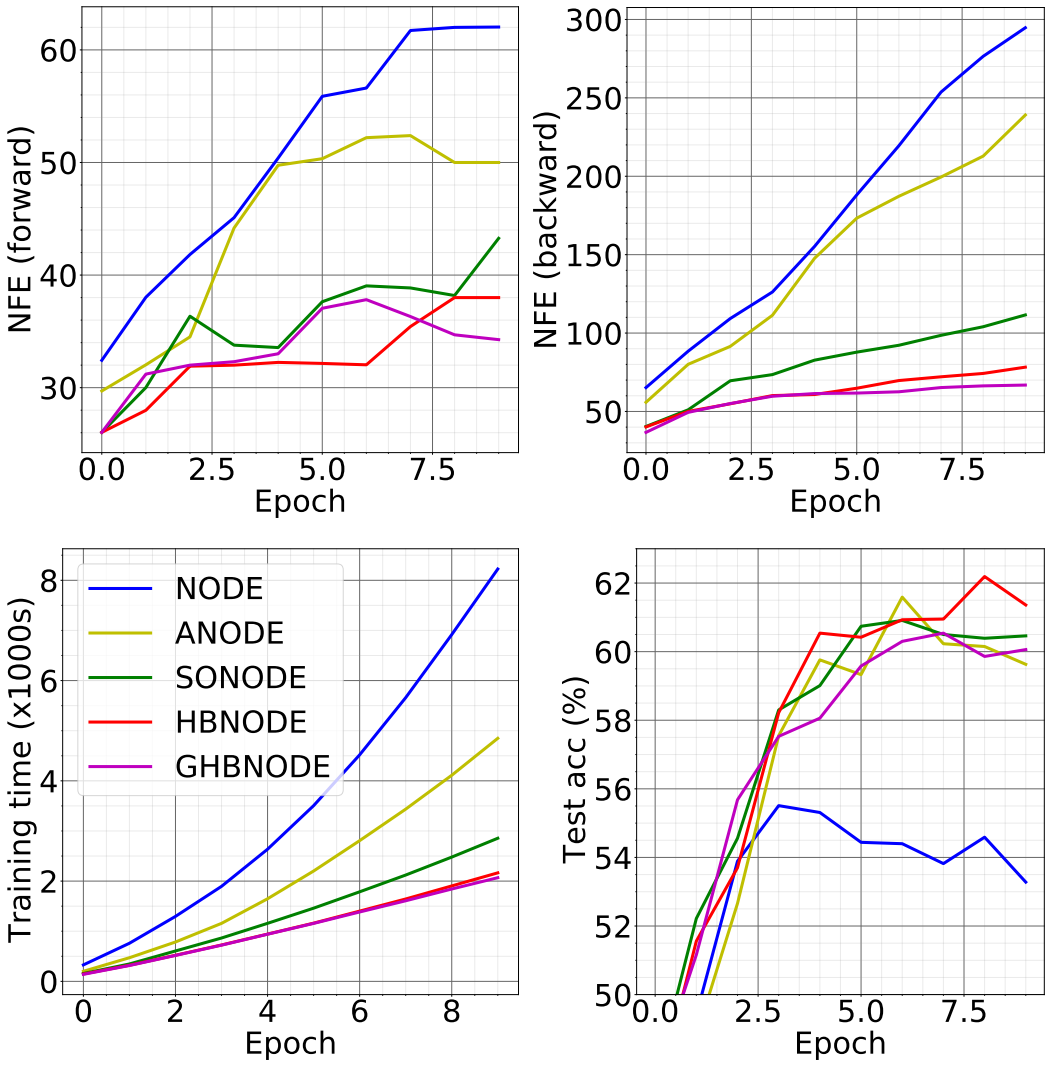


Figure 3.1: Contrasting NODE, ANODE, SONODE, HBNODE, and GHBNODE for CIFAR10 classification in NFEs, training time, and test accuracy. (Tolerance: 10^{-5}, see Sec. 3.5.2 for experimental details.)

NODEs are flexible in learning from irregularly-sampled sequential data and particularly suitable for learning complex dynamical systems [Che+18; RCD19; Zha+19b; Nor+20; DFD20; Kid+20], which can be trained by efficient algorithms [Qua+20; Dau+20; Zhu+21]. NODE-based continuous generative models have computational advantages over the classical normalizing flows [Che+18; Gra+19; YHL19; Fin+20]. NODEs have also been generalized to neural stochastic differential equations, stochastic processes, and graph NODEs [JB19; Li+20b; Pol+19; TR19; HSW20; Nor+21].

The drawback of NODEs is also prominent. In many ML tasks, NODEs require very high NFEs in both training and inference, especially in high accuracy settings where a lower tolerance is needed. The NFEs increase rapidly with training; high NFEs reduce computational speed and accuracy of NODEs and can lead to blow-ups in the worst-case scenario [Gra+19; DDT19; Mas+20; Nor+20]. As an illustration, we train NODEs for CIFAR10 classification using the same model and experimental settings as in [DDT19], except using a tolerance of 10^{-5}; Fig. 3.1 shows both forward and backward NFEs and the training time of different ODE-based models; we see that NFEs and computationaltimes increase very rapidly for NODE, ANODE [DDT19], and SONODE [Nor+20]. More results on the large NFE and degrading utility issues for different benchmark experiments are available in Sec. 3.5. Another issue is that NODEs often fail to effectively learn long-term dependencies in sequential data [LH20], as discussed in Sec. 3.4.

3.1.1 Contribution

We propose heavy ball neural ODEs (HBNODEs), leveraging the continuous limit of the classical momentum accelerated gradient descent, to improve NODE training and inference. At the core of HBNODE is replacing the first-order ODE (3.1) with a heavy ball ODE (HBODE), i.e., a second-order ODE with an appropriate damping term. HBNODEs have two theoretical properties that imply practical advantages over NODEs:

- The adjoint equation used for training a HBNODE is also a HBNODE (see Prop. 1 and Prop. 2), accelerating both forward and backward propagation, thus significantly reducing

both forward and backward NFEs. The reduction in NFE using HBNODE over existing benchmark ODE-based models becomes more aggressive as the error tolerance of the ODE solvers decreases.

- The spectrum of the HBODE is well-structured (see Prop. 4), alleviating the vanishing gradient issue in back-propagation and enabling the model to effectively learn long-term dependencies from sequential data.

To mitigate the potential blow-up problem in training HBNODEs, we further propose generalized HBNODEs (GHBNODEs) by integrating skip connections [He+16b] and gating mechanisms [HS97] into the HBNODE. See Sec. 3.3 for details.

3.1.2 Organization

We organize the chapter as follows: In Secs. 3.2 and 3.3, we present our motivation, algorithm, and analysis of HBNODEs and GHBNODEs, respectively. We analyze the spectrum structure of the adjoint equation of HBNODEs/GHBNODEs in Sec. 3.4, which indicates that HBNODEs/GHBNODEs can learn long-term dependency effectively. We test the performance of HBNODEs and GHBNODEs on benchmark point cloud separation, image classification, learning dynamics, and sequential modeling in Sec. 3.5. We discuss more related work in Sec. 3.6, followed by concluding remarks.

3.2 Heavy Ball Neural Ordinary Differential Equations

3.2.1 Heavy ball ordinary differential equation

Classical momentum method, a.k.a., the heavy ball method, has achieved remarkable success in accelerating gradient descent [Pol64] and has significantly improved the training of deep neural networks [Sut+13]. As the continuous limit of the classical momentum method, heavy ball ODE (HBODE) has been studied in various settings and has been used to analyze the acceleration phenomenon of the momentum methods. For the ease of reading and completeness, we derive the HBODE from the classical momentum method. Starting from

initial points $\boldsymbol{x}^0$ and $\boldsymbol{x}^1$, gradient descent with classical momentum searches a minimum of the function $F(\boldsymbol{x})$ through the following iteration

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - s\nabla F(\boldsymbol{x}^k) + \beta(\boldsymbol{x}^k - \boldsymbol{x}^{k-1}), \tag{3.4}$$

where $s > 0$ is the step size and $0 \leq \beta < 1$ is the momentum hyperparameter. For any fixed step size s, let $\boldsymbol{m}^k := (\boldsymbol{x}^{k+1} - \boldsymbol{x}^k)/\sqrt{s}$, and let $\beta := 1 - \gamma\sqrt{s}$, where $\gamma \geq 0$ is another hyperparameter. Then we can rewrite (3.4) as

$$\boldsymbol{m}^{k+1} = (1 - \gamma\sqrt{s})\boldsymbol{m}^k - \sqrt{s}\nabla F(\boldsymbol{x}^k);\ \boldsymbol{x}^{k+1} = \boldsymbol{x}^k + \sqrt{s}\boldsymbol{m}^{k+1}. \tag{3.5}$$

Let $s \to 0$ in (3.5); we obtain the following system of first-order ODEs,

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{m}(t);\ \frac{d\boldsymbol{m}(t)}{dt} = -\gamma\boldsymbol{m}(t) - \nabla F(\boldsymbol{x}(t)). \tag{3.6}$$

This can be further rewritten as a second-order heavy ball ODE (HBODE), which also models a damped oscillator,

$$\frac{d^2\boldsymbol{x}(t)}{dt^2} + \gamma\frac{d\boldsymbol{x}(t)}{dt} = -\nabla F(\boldsymbol{x}(t)). \tag{3.7}$$

We compare the dynamics of HBODE (3.7) and the following ODE limit of the gradient descent (GD)

$$\frac{d\boldsymbol{x}}{dt} = -\nabla F(\boldsymbol{x}). \tag{3.8}$$

In particular, we solve the ODEs (3.7) and (3.8) with $F(\boldsymbol{x})$ defined as a `Rosenbrock` [Ros60] or `Beale` [Gon71] function. The comparisons show that HBODE can accelerate the dynamics of the ODE for a gradient system, which motivates us to propose HBNODE to accelerate forward propagation of NODE.

Rosenbrock function. The Rosenbrock function is given by

$$F(\boldsymbol{x}) := F(x, y) = 100(y - x^2)^2 + (1 - x)^2,$$

which has the minimum $(x, y) = (1, 1)$. Starting from $(0, 0)$, we apply Dormand–Prince-45 solver using a step size $\Delta t = 0.001$ to solve both ODEs (3.7) and (3.8) for t from 0 to 1. For the HBODE, we set $\gamma = 0.9$ and set the initial value of $d\boldsymbol{x}/dt = (0, 0)$.

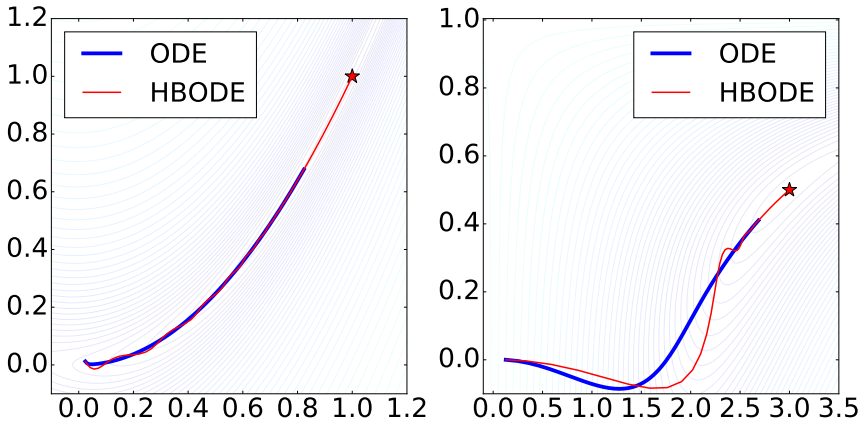


Figure 3.2: Comparing the trajectory of ODE and HBODE when $F(\boldsymbol{x})$ is the Rosenbrock (left) and Beale (right) functions.

Beale function. The Beale function is given by

$$F(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$$

which has the minimum $(x, y) = (3, 0.5)$. Starting from $(0, 0)$, we apply Dormand–Prince-45 solver using a step size 0.01 to solve both ODEs in (3.7) and (3.8) for t from 0 to 2. For the HBODE, we set $\gamma = 0.7$ and set the initial value of $d\boldsymbol{x}/dt = (0, 0)$.

To numerically show that the HBNODE (3.7) converges faster to the stationary point than the ODE limit of gradient descent (3.8), we apply the Dormand–Prince-45 ODE solver, which is the default solver for NODEs, to solve both ODEs. We set $F(\boldsymbol{x})$ to be the Rosenbrock and the Beale functions. Fig. 3.2 shows that with the same numerical ODE solver, HBODE converges to the stationary point (marked by stars) faster than (3.8).

3.2.2 Adjoint Equation for the First- and Second-order ODEs

The adjoint sensitivity method is the key to assuring constant memory usage in training neural ODEs [Che+18]. In this section, we present two different proofs for the first-order adjoint sensitivity equations. The differentiation proof in 3.2.2.2 is adapted from the proof by Norcliffe et al [Nor+20]. We provide a new integral proof in 3.2.2.3 to extend theoretical

support for the Lipschitz continuous functions. We also revisit the proof of the second-order adjoint sensitivity equations by Norcliffe et al [Nor+20].

3.2.2.1 First-order Adjoint Sensitivity Equation

A Neural ODE for hidden features $\boldsymbol{h}(t) \in \mathbb{R}^N$ takes the form

$$\frac{\partial \boldsymbol{h}}{\partial t} = f(\boldsymbol{h}(t), t, \theta), \quad \boldsymbol{h}(t_0) = \boldsymbol{h}_{t_0}, \quad \boldsymbol{h}(T) = \boldsymbol{h}_T, \tag{3.9}$$

where $f(\boldsymbol{h}(t), t, \theta) \in \mathbb{R}^N$ is a neural network with learnable parameters θ. The corresponding adjoint equation, with $\mathcal{L}$ being a scalar loss function, is defined by the following ODE,

$$\frac{\partial \boldsymbol{A}(t)}{\partial t} = -\boldsymbol{A}(t)\frac{\partial f}{\partial \boldsymbol{h}}, \quad \boldsymbol{A}(T) = -\boldsymbol{I}, \quad \boldsymbol{a}(t) = -\frac{d\mathcal{L}}{d\boldsymbol{h}_T}\boldsymbol{A}(t). \tag{3.10}$$

For gradient-based optimization, we need to compute the following derivatives

$$\frac{d\mathcal{L}}{d\theta} = \frac{d\mathcal{L}}{d\boldsymbol{h}_T}\frac{d\boldsymbol{h}_T}{d\theta}, \quad \frac{d\mathcal{L}}{d\boldsymbol{h}_{t_0}} = \frac{d\mathcal{L}}{d\boldsymbol{h}_T}\frac{d\boldsymbol{h}_T}{d\boldsymbol{h}_{t_0}}. \tag{3.11}$$

In the following sections, we show that

$$\frac{d\boldsymbol{h}_T}{d\theta} = \int_T^{t_0} \boldsymbol{A}\frac{\partial f}{\partial \theta}dt, \quad \frac{d\boldsymbol{h}_T}{d\boldsymbol{h}_{t_0}} = -\boldsymbol{A}(t_0). \tag{3.12}$$

By linearity, we immediately arrive at the following adjoint sensitivity equations

$$\frac{d\mathcal{L}}{d\theta} = \int_{t_0}^{T} \boldsymbol{a}\frac{\partial f}{\partial \theta}dt, \quad \frac{d\mathcal{L}}{d\boldsymbol{h}_{t_0}} = \boldsymbol{a}(t_0). \tag{3.13}$$

3.2.2.2 Proof of the First-Order Adjoint Sensitivity Equation: Differentiation Approach

We adapt the proof of the adjoint sensitivity equations from Norcliffe et al [Nor+20]. Assume that $f \in C^1$, ϕ is either θ or $\boldsymbol{h}_{t_0}$, then the following equations hold

$$\frac{\partial \boldsymbol{A}(t)}{\partial t} = -\boldsymbol{A}(t)\frac{\partial f}{\partial \boldsymbol{h}}, \quad \frac{\partial^2 \boldsymbol{h}}{\partial \phi \partial t} = \frac{\partial f}{\partial \theta}\frac{d\theta}{d\phi} + \frac{\partial f}{\partial \boldsymbol{h}}\frac{d\boldsymbol{h}}{d\phi}, \quad \frac{\partial\big(\boldsymbol{A}\frac{\partial \boldsymbol{h}}{\partial \phi}\big)}{\partial t} = \frac{\partial \boldsymbol{A}}{\partial t}\frac{\partial \boldsymbol{h}}{\partial \phi} + \boldsymbol{A}\frac{\partial^2 \boldsymbol{h}}{\partial \phi \partial t}. \tag{3.14}$$

Combining the three equations in (3.14) yields the differential equation

$$\frac{\partial\big(\boldsymbol{A}\frac{\partial \boldsymbol{h}}{\partial \phi}\big)}{\partial t} = \frac{\partial \boldsymbol{A}}{\partial t}\frac{\partial \boldsymbol{h}}{\partial \phi} + \boldsymbol{A}\frac{\partial^2 \boldsymbol{h}}{\partial \phi \partial t} = -\boldsymbol{A}(t)\frac{\partial f}{\partial \boldsymbol{h}}\frac{\partial \boldsymbol{h}}{\partial \phi} + \boldsymbol{A}\Big(\frac{\partial f}{\partial \theta}\frac{d\theta}{d\phi} + \frac{\partial f}{\partial \boldsymbol{h}}\frac{d\boldsymbol{h}}{d\phi}\Big) = \boldsymbol{A}\frac{\partial f}{\partial \theta}\frac{d\theta}{d\phi}. \tag{3.15}$$

Integrating both sides of (3.15) in t from T to t_0, we arrive at the integral equation

$$\Big(\boldsymbol{A}\frac{\partial \boldsymbol{h}}{\partial \phi}\Big)\Big|_T^{t_0} = \int_T^{t_0} \boldsymbol{A}\frac{\partial f}{\partial \theta}\frac{d\theta}{d\phi}dt. \tag{3.16}$$

Using the conditions $\boldsymbol{A}(T) = -\boldsymbol{I}$, $\boldsymbol{h}(t_0) = \boldsymbol{h}_{t_0}$, $\boldsymbol{h}(T) = \boldsymbol{h}_T$, we rewrite the equation (3.16) as

$$\frac{d\boldsymbol{h}_T}{d\phi} = -\boldsymbol{A}(t_0)\frac{d\boldsymbol{h}_{t_0}}{d\phi} + \int_T^{t_0} \boldsymbol{A}\frac{\partial f}{\partial \theta}\frac{d\theta}{d\phi}dt. \tag{3.17}$$

Substituting $\phi = \boldsymbol{h}_{t_0}$ and $\phi = \theta$ respectively in (3.17) leads to

$$\frac{d\boldsymbol{h}_T}{d\boldsymbol{h}_{t_0}} = -\boldsymbol{A}(t_0), \quad \frac{d\boldsymbol{h}_T}{d\theta} = \int_T^{t_0} \boldsymbol{A}\frac{\partial f}{\partial \theta}dt. \tag{3.18}$$

This proof is adapted from the proof provided by Norcliffe et al [Nor+20] for general second-order neural ODEs by differentiation and this proof only holds for $f \in C^1$.

3.2.2.3 Proof of the First-Order Adjoint Sensitivity Equations: Integration Approach

The proof in 3.2.2.2 requires that $f \in C^1$. However, with activation functions like ReLU, f may not be smooth enough to satisfy this requirement. Meanwhile, the adjoint equation (3.10) that $\boldsymbol{A}$ satisfies may not have a continuous right hand side, which can fail the Picard-Lindelöf theorem that guarantees the existence and uniqueness of solutions to the adjoint equation.

To circumvent these deficiencies, we propose a new proof based on integration. Assume that $f(\boldsymbol{h}, t, \theta)$ is continuous in t and Lipschitz continuous in $\boldsymbol{h}, \theta$, and there exists some open ball around $\boldsymbol{h}_{t_0} = \boldsymbol{s}_0$, $\theta = \theta_0$ such that for every pair of initial condition and parameters in the open ball, there exists a unique solution for $t \in [t_0, T]$. We denote the solution starting from $\boldsymbol{h}_{t_0} = \boldsymbol{s}_0$, $\theta = \theta_0$ as $\boldsymbol{h}_0$. In order to avoid difficulties in proving the existence and uniqueness of the solution, we explicitly define the adjoint equation through the following matrix exponential

$$\boldsymbol{A}(t) = -\exp\left\{-\int_T^t \frac{\partial f}{\partial \boldsymbol{h}}(\boldsymbol{h}_0(\tau), \tau, \theta_0)d\tau\right\}. \tag{3.19}$$

By definition, $\boldsymbol{A}$ is Lipschitz continuous and satisfies the differential equation almost everywhere

$$\frac{d\boldsymbol{A}(t)}{dt} = -\boldsymbol{A}(t)\frac{\partial f}{\partial \boldsymbol{h}}(\boldsymbol{h}_0(t), t, \theta_0). \tag{3.20}$$

Since $\boldsymbol{h} \in C^1(t)$ and $\frac{d\boldsymbol{A}}{dt} \in L^1(t)$, we obtain the following using integration by parts,

$$\boldsymbol{A}\boldsymbol{h}|_{t_0}^{T} = \int_{t_0}^{T} \boldsymbol{A}\frac{\partial \boldsymbol{h}}{\partial t}dt + \int_{t_0}^{T} \frac{d\boldsymbol{A}}{dt}\boldsymbol{h}dt. \tag{3.21}$$

Taking partial derivatives with respect to ϕ on both sides of (3.21), as $\boldsymbol{A}(t)$ is only a function of t, we have

$$\boldsymbol{A}\frac{\partial \boldsymbol{h}}{\partial \phi}\Big|_{t_0}^{T} = \frac{\partial}{\partial \phi}\int_{t_0}^{T} \boldsymbol{A}\frac{\partial \boldsymbol{h}}{\partial t}dt + \frac{\partial}{\partial \phi}\int_{t_0}^{T} \frac{d\boldsymbol{A}}{dt}\boldsymbol{h}dt. \tag{3.22}$$

In order to exchange integral and derivatives, we use the dominated convergence theorem. Because f is Lipschitz continuous on $\boldsymbol{h}$, $\boldsymbol{h}$ is Lipschitz continuous on ϕ, and thus $\frac{\partial \boldsymbol{h}}{\partial \phi}$ is Lebesgue integrable. Therefore, by chain rule, the following equation holds almost everywhere,

$$\frac{\partial^2 \boldsymbol{h}}{\partial t \partial \phi} = \frac{\partial^2 \boldsymbol{h}}{\partial \phi \partial t} = \frac{df}{d\phi} = \frac{\partial f}{\partial \theta}\frac{d\theta}{d\phi} + \frac{\partial f}{\partial \boldsymbol{h}}\frac{d\boldsymbol{h}}{d\phi}. \tag{3.23}$$

Because t is bounded, the right hand side of equation (3.23) is Lebesgue integrable, and so is the left hand side. Because both $\frac{\partial \boldsymbol{h}}{\partial \phi}$ and $\frac{\partial^2 \boldsymbol{h}}{\partial t \partial \phi}$ are Lebesgue integrable, by dominated convergence theorem, we have the following exchange of integrals and derivatives

$$\frac{\partial}{\partial \phi}\int_{t_0}^{T} \boldsymbol{A}\frac{\partial \boldsymbol{h}}{\partial t}dt = \int_{t_0}^{T} \boldsymbol{A}\frac{\partial^2 \boldsymbol{h}}{\partial t \partial \phi}dt, \quad \frac{\partial}{\partial \phi}\int_{t_0}^{T} \frac{d\boldsymbol{A}}{dt}\boldsymbol{h}dt = \int_{t_0}^{T} \frac{d\boldsymbol{A}}{dt}\frac{\partial \boldsymbol{h}}{\partial \phi}dt. \tag{3.24}$$

Combining equation (3.22) with (3.24) gives us

$$\boldsymbol{A}\frac{\partial \boldsymbol{h}}{\partial \phi}\Big|_{t_0}^{T} = \int_{t_0}^{T} \boldsymbol{A}\frac{\partial^2 \boldsymbol{h}}{\partial t \partial \phi}dt + \int_{t_0}^{T} \frac{d\boldsymbol{A}}{dt}\frac{\partial \boldsymbol{h}}{\partial \phi}dt. \tag{3.25}$$

By taking Lebesgue integral of equation (3.23), we have the equation

$$\int_{t_0}^{T} \boldsymbol{A}\frac{\partial^2 \boldsymbol{h}}{\partial t \partial \phi}dt = \int_{t_0}^{T} \boldsymbol{A}\Big(\frac{\partial f}{\partial \theta}\frac{d\theta}{d\phi} + \frac{\partial f}{\partial \boldsymbol{h}}\frac{d\boldsymbol{h}}{d\phi}\Big)dt. \tag{3.26}$$

Meanwhile, at $\boldsymbol{h}_0$, we can integrate equation (3.20) to a similar form as

$$\int_{t_0}^{T} \frac{d\boldsymbol{A}}{dt}\frac{\partial \boldsymbol{h}}{\partial \phi}dt = -\int_{t_0}^{T} \boldsymbol{A}\frac{\partial f}{\partial \boldsymbol{h}}\frac{d\boldsymbol{h}}{d\phi}dt. \tag{3.27}$$

Consequently, at $\boldsymbol{h}_0$, we can sum up equations (3.25), (3.26), and (3.27) and arrive at

$$\boldsymbol{A}\frac{\partial \boldsymbol{h}}{\partial \phi}\Big|_{t_0}^{T} = \int_{t_0}^{T} \boldsymbol{A}\frac{\partial f}{\partial \theta}\frac{d\theta}{d\phi}dt, \tag{3.28}$$

which is the same integral equation as equation (3.16) in the differentiation proof in 3.2.2.2. Thus, plugging in the initial conditions provides us with the same result.

3.2.2.4 Corollary of the First-order Gradient Propagation

An immediate corollary of the above proof is that combining equations (3.12) and (3.19) results in

$$\frac{d\boldsymbol{h}_T}{d\boldsymbol{h}_{t_0}} = -\boldsymbol{A}(t_0) = \exp\left\{ -\int_T^{t_0} \frac{\partial f}{\partial \boldsymbol{h}}(\boldsymbol{h}_0(\tau), \tau, \theta_0) d\tau \right\}. \tag{3.29}$$

As (3.29) is true for every choice of t_0, we can also generalize it to

$$\frac{d\boldsymbol{h}_T}{d\boldsymbol{h}_t} = \exp\left\{ -\int_T^{t} \frac{\partial f}{\partial \boldsymbol{h}}(\boldsymbol{h}_0(\tau), \tau, \theta_0) d\tau \right\}, \tag{3.30}$$

which shows the relative gradient between different times in integral.

3.2.2.5 Second-order Adjoint Sensitivity Equation

A SONODE satisfies the following equations

$$\frac{\partial \boldsymbol{h}}{\partial t} = \boldsymbol{v}, \quad \frac{\partial \boldsymbol{v}}{\partial t} = f(\boldsymbol{h}(t), \boldsymbol{v}(t), t, \theta), \quad \boldsymbol{h}(t_0) = \boldsymbol{h}_{t_0}, \quad \boldsymbol{v}(t_0) = \boldsymbol{v}_{t_0}, \tag{3.31}$$

which can be viewed as a coupled first-order ODE system of the form

$$\frac{\partial}{\partial t}\begin{bmatrix} \boldsymbol{h} \\ \boldsymbol{v} \end{bmatrix} = \begin{bmatrix} \boldsymbol{v} \\ f(\boldsymbol{h}(t), \boldsymbol{v}(t), t, \theta) \end{bmatrix}, \quad \begin{bmatrix} \boldsymbol{h} \\ \boldsymbol{v} \end{bmatrix}(t_0) = \begin{bmatrix} \boldsymbol{h}_{t_0} \\ \boldsymbol{v}_{t_0} \end{bmatrix}. \tag{3.32}$$

Denote $\boldsymbol{z} = \begin{bmatrix} \boldsymbol{h} \\ \boldsymbol{v} \end{bmatrix}$ and final state as

$$\begin{bmatrix} \boldsymbol{h}(T) \\ \boldsymbol{v}(T) \end{bmatrix} = \begin{bmatrix} \boldsymbol{h}_T \\ \boldsymbol{v}_T \end{bmatrix} = \boldsymbol{z}_T. \tag{3.33}$$

Using the conclusions from 3.2.2.1, then the adjoint equation is given by

$$\frac{\partial \boldsymbol{A}(t)}{\partial t} = -\boldsymbol{A}(t) \begin{bmatrix} \boldsymbol{0} & \boldsymbol{I} \\ \frac{\partial f}{\partial \boldsymbol{h}} & \frac{\partial f}{\partial \boldsymbol{v}} \end{bmatrix}, \quad \boldsymbol{A}(T) = -\boldsymbol{I}, \quad \boldsymbol{a}(t) = -\frac{d\mathcal{L}}{d\boldsymbol{z}_T}\boldsymbol{A}(t). \tag{3.34}$$

By rewriting $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_{\boldsymbol{h}} & \boldsymbol{A}_{\boldsymbol{v}} \end{bmatrix}$, we have the following differential equations

$$\frac{\partial \boldsymbol{A}_{\boldsymbol{h}}(t)}{\partial t} = -\boldsymbol{A}_{\boldsymbol{v}}(t)\frac{\partial f}{\partial \boldsymbol{h}}, \quad \frac{\partial \boldsymbol{A}_{\boldsymbol{v}}(t)}{\partial t} = -\boldsymbol{A}_{\boldsymbol{h}}(t) - \boldsymbol{A}_{\boldsymbol{v}}(t)\frac{\partial f}{\partial \boldsymbol{v}}, \tag{3.35}$$

with initial conditions

$$\boldsymbol{A}_{\boldsymbol{h}}(T) = -\begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{0} \end{bmatrix}, \quad \boldsymbol{A}_{\boldsymbol{v}}(T) = -\begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{I} \end{bmatrix}, \tag{3.36}$$

and adjoint states

$$\boldsymbol{a}_{\boldsymbol{h}}(t) = \frac{d\mathcal{L}}{d\boldsymbol{z}_T}\boldsymbol{A}_{\boldsymbol{h}}(t), \quad \boldsymbol{a}_{\boldsymbol{v}}(t) = \frac{d\mathcal{L}}{d\boldsymbol{z}_T}\boldsymbol{A}_{\boldsymbol{v}}(t). \tag{3.37}$$

The gradient equations becomes

$$\frac{d\mathcal{L}}{d\theta} = \int_{t_0}^{T} \boldsymbol{a} \begin{bmatrix} \boldsymbol{0} \\ \frac{\partial f}{\partial \theta} \end{bmatrix} dt = \int_{t_0}^{T} \boldsymbol{a}_{\boldsymbol{v}} \frac{\partial f}{\partial \theta} dt, \quad \frac{d\mathcal{L}}{d\boldsymbol{h}_{t_0}} = \boldsymbol{a}_{\boldsymbol{h}}(t_0), \quad \frac{d\mathcal{L}}{d\boldsymbol{v}_{t_0}} = \boldsymbol{a}_{\boldsymbol{v}}(t_0). \tag{3.38}$$

In SONODE, $\boldsymbol{h}_{t_0}$ is fixed, and thus $\boldsymbol{a}_{\boldsymbol{h}}$ disappears in gradient computation. Therefore, we are only interested in $\boldsymbol{a}_{\boldsymbol{v}}$. Thus the adjoint $\boldsymbol{A}_{\boldsymbol{v}}$ satisfies the following second-order ODE

$$\frac{\partial^2 \boldsymbol{A}_{\boldsymbol{v}}(t)}{\partial t^2} = \boldsymbol{A}_{\boldsymbol{v}}(t)\frac{\partial f}{\partial \boldsymbol{h}} - \frac{\partial (\boldsymbol{A}_{\boldsymbol{v}}(t)\frac{\partial f}{\partial \boldsymbol{v}})}{\partial t}, \tag{3.39}$$

and thus

$$\frac{\partial^2 \boldsymbol{a}_{\boldsymbol{v}}(t)}{\partial t^2} = \boldsymbol{a}_{\boldsymbol{v}}(t)\frac{\partial f}{\partial \boldsymbol{h}} - \frac{\partial (\boldsymbol{a}_{\boldsymbol{v}}(t)\frac{\partial f}{\partial \boldsymbol{v}})}{\partial t}, \tag{3.40}$$

with initial conditions

$$\boldsymbol{a}_{\boldsymbol{v}}(T) = -\frac{d\mathcal{L}}{d\boldsymbol{z}}\boldsymbol{A}_{\boldsymbol{v}}(T) = \frac{d\mathcal{L}}{d\boldsymbol{v}_T}, \quad \frac{\partial \boldsymbol{a}_{\boldsymbol{v}}(T)}{\partial t} = -\frac{d\mathcal{L}}{d\boldsymbol{h}_T} - \boldsymbol{a}_{\boldsymbol{v}}(T)\frac{\partial f}{\partial \boldsymbol{v}}(T). \tag{3.41}$$

This proves the second order adjoint equations for $\boldsymbol{a}_{\boldsymbol{v}}$.

3.2.3 Heavy ball neural ordinary differential equations

Similar to NODE, we parameterize $-\nabla F$ in (3.7) using a neural network $f(\boldsymbol{h}(t), t, \theta)$, resulting in the following HBNODE with initial position $\boldsymbol{h}(t_0)$ and momentum $\boldsymbol{m}(t_0) := d\boldsymbol{h}/dt(t_0)$,

$$\frac{d^2\boldsymbol{h}(t)}{dt^2} + \gamma \frac{d\boldsymbol{h}(t)}{dt} = f(\boldsymbol{h}(t), t, \theta), \tag{3.42}$$

where $\gamma \geq 0$ is the damping parameter, which can be set as a tunable or a learnable hyperparmater with positivity constraint. In the trainable case, we use $\gamma = \epsilon \cdot \textbf{Sigmoid}(\omega)$ for a trainable $\omega \in \mathbb{R}$ and a fixed tunable upper bound ϵ (we set $\epsilon = 1$ below). According to (3.6), HBNODE (3.42) is equivalent to

$$\frac{d\boldsymbol{h}(t)}{dt} = \boldsymbol{m}(t); \quad \frac{d\boldsymbol{m}(t)}{dt} = -\gamma \boldsymbol{m}(t) + f(\boldsymbol{h}(t), t, \theta). \tag{3.43}$$

Equation (3.42) (or equivalently, the system (3.43)) defines the forward ODE for the HBNODE, and we can use either the first-order (Prop. 2) or the second-order (Prop. 1) adjoint sensitivity method to update the parameter θ [Nor+20].

Proposition 1 (Adjoint equation for HBNODE)**.** *The adjoint state $\boldsymbol{a}(t) := \partial \mathcal{L}/\partial \boldsymbol{h}(t)$ for the HBNODE* (3.42) *satisfies the following HBODE with the same damping parameter γ as that in* (3.42)*,*

$$\frac{d^2\boldsymbol{a}(t)}{dt^2} - \gamma \frac{d\boldsymbol{a}(t)}{dt} = \boldsymbol{a}(t) \frac{\partial f}{\partial \boldsymbol{h}}(\boldsymbol{h}(t), t, \theta). \tag{3.44}$$

Remark 1. *Note that we solve the adjoint equation* (3.44) *from time $t = T$ to $t = t_0$ in the backward propagation. By letting $\tau = T - t$ and $\boldsymbol{b}(\tau) = \boldsymbol{a}(T - \tau)$, we can rewrite* (3.44) *as follows,*

$$\frac{d^2\boldsymbol{b}(\tau)}{d\tau^2} + \gamma \frac{d\boldsymbol{b}(\tau)}{d\tau} = \boldsymbol{b}(\tau) \frac{\partial f}{\partial \boldsymbol{h}}(\boldsymbol{h}(T - \tau), T - \tau, \theta). \tag{3.45}$$

Therefore, the adjoint of the HBNODE is also a HBNODE and they have the same damping parameter.

Proof. As HBNODE takes the form

$$\frac{d^2\boldsymbol{h}(t)}{dt^2} + \gamma \frac{d\boldsymbol{h}(t)}{dt} = f(\boldsymbol{h}(t), t, \theta), \tag{3.46}$$

which can also be viewed as a SONODE. By applying the adjoint equation of SONODE (3.40), we arrive at

$$\frac{\partial^2 \boldsymbol{a}(t)}{\partial t^2} = \boldsymbol{a}(t)\frac{\partial f}{\partial \boldsymbol{h}} + \gamma \frac{\partial \boldsymbol{a}(t)}{\partial t}. \tag{3.47}$$

As HBNODE only carries its state $\boldsymbol{h}$ to the loss $\mathcal{L}$, we have $\frac{d\mathcal{L}}{d\boldsymbol{v}_T} = 0$, and thus the initial conditions in equation (3.41) becomes

$$\boldsymbol{a}(T) = \boldsymbol{0}, \quad \frac{\partial \boldsymbol{a}(T)}{\partial t} = -\frac{d\mathcal{L}}{d\boldsymbol{h}_T}. \tag{3.48}$$

We can also employ (3.43) and its adjoint for the forward and backward propagations, respectively. □

Proposition 2 (Adjoint equations for the first-order HBNODE system)**.** *The adjoint states* $\boldsymbol{a}_{\boldsymbol{h}}(t) := \partial\mathcal{L}/\partial\boldsymbol{h}(t)$ *and* $\boldsymbol{a}_{\boldsymbol{m}}(t) := \partial\mathcal{L}/\partial\boldsymbol{m}(t)$ *for the first-order HBNODE system* (3.43) *satisfy*

$$\frac{d\boldsymbol{a}_{\boldsymbol{h}}(t)}{dt} = -\boldsymbol{a}_{\boldsymbol{m}}(t)\frac{\partial f}{\partial \boldsymbol{h}}(\boldsymbol{h}(t), t, \theta); \quad \frac{d\boldsymbol{a}_{\boldsymbol{m}}(t)}{dt} = -\boldsymbol{a}_{\boldsymbol{h}}(t) + \gamma \boldsymbol{a}_{\boldsymbol{m}}(t). \tag{3.49}$$

Remark 2. *Let* $\tilde{\boldsymbol{a}}_{\boldsymbol{m}}(t) = d\boldsymbol{a}_{\boldsymbol{m}}(t)/dt$*, then* $\boldsymbol{a}_{\boldsymbol{m}}(t)$ *and* $\tilde{\boldsymbol{a}}_{\boldsymbol{m}}(t)$ *satisfies the following first-order heavy ball ODE system*

$$\frac{d\boldsymbol{a}_{\boldsymbol{m}}(t)}{dt} = \tilde{\boldsymbol{a}}_{\boldsymbol{m}}(t); \quad \frac{d\tilde{\boldsymbol{a}}_{\boldsymbol{m}}(t)}{dt} = \boldsymbol{a}_{\boldsymbol{m}}(t)\frac{\partial f}{\partial \boldsymbol{h}}(\boldsymbol{h}(t), t, \theta) + \gamma \tilde{\boldsymbol{a}}_{\boldsymbol{m}}(t). \tag{3.50}$$

Note that we solve this system backward in time in back-propagation. Moreover, we have $\boldsymbol{a}_{\boldsymbol{h}}(t) = \gamma \boldsymbol{a}_{\boldsymbol{m}}(t) - \tilde{\boldsymbol{a}}_{\boldsymbol{m}}(t)$.

Proof. The coupled form of HBNODE is a coupled first-order ODE system of the form

$$\frac{\partial}{\partial t}\begin{bmatrix}\boldsymbol{h}\\ \boldsymbol{m}\end{bmatrix} = \begin{bmatrix}\boldsymbol{m}\\ -\gamma\boldsymbol{m} + f(\boldsymbol{h}(t), t, \theta)\end{bmatrix}, \quad \begin{bmatrix}\boldsymbol{h}\\ \boldsymbol{m}\end{bmatrix}(t_0) = \begin{bmatrix}\boldsymbol{h}_{t_0}\\ \boldsymbol{m}_{t_0}\end{bmatrix}. \tag{3.51}$$

Denote the final state as

$$\begin{bmatrix}\boldsymbol{h}(T)\\ \boldsymbol{m}(T)\end{bmatrix} = \begin{bmatrix}\boldsymbol{h}_T\\ \boldsymbol{m}_T\end{bmatrix} = z. \tag{3.52}$$

Using the conclusions from 3.2.2.1, we have the adjoint equation

$$\frac{\partial \boldsymbol{A}(t)}{\partial t} = -\boldsymbol{A}(t) \begin{bmatrix} \boldsymbol{0} & \boldsymbol{I} \\ \frac{\partial f}{\partial \boldsymbol{h}} & -\gamma \boldsymbol{I} \end{bmatrix}, \quad \boldsymbol{A}(T) = -\boldsymbol{I}, \quad \boldsymbol{a}(t) = -\frac{d\mathcal{L}}{d\boldsymbol{z}} \boldsymbol{A}(t). \tag{3.53}$$

Let $\begin{bmatrix} \boldsymbol{a_h} & \boldsymbol{a_m} \end{bmatrix} = \boldsymbol{a}$, by linearity we have

$$\frac{\partial \begin{bmatrix} \boldsymbol{a_h} & \boldsymbol{a_m} \end{bmatrix}}{\partial t} = -\begin{bmatrix} \boldsymbol{a_h} & \boldsymbol{a_m} \end{bmatrix} \begin{bmatrix} \boldsymbol{0} & \boldsymbol{I} \\ \frac{\partial f}{\partial \boldsymbol{h}} & -\gamma \boldsymbol{I} \end{bmatrix}, \quad \begin{bmatrix} \boldsymbol{a_h}(T) & \boldsymbol{a_m}(T) \end{bmatrix} = \begin{bmatrix} \frac{d\mathcal{L}}{d\boldsymbol{h}_T} & \frac{d\mathcal{L}}{d\boldsymbol{m}_T} \end{bmatrix}, \tag{3.54}$$

which gives us the initial conditions at $t = T$, and the simplified first-order ODE system

$$\frac{\partial \boldsymbol{a_h}}{\partial t} = -\boldsymbol{a_m} \frac{\partial f}{\partial \boldsymbol{h}}, \quad \frac{\partial \boldsymbol{a_m}}{\partial t} = -\boldsymbol{a_h} + \gamma \boldsymbol{a_m}. \tag{3.55}$$

Similar to [Nor+20], we use the coupled first-order HBNODE system (3.43) and its adjoint first-order HBNODE system (3.49) for practical implementation, since the entangled representation permits faster computation [Nor+20] of the gradients of the coupled ODE systems. □

3.3 Generalized Heavy Ball Neural Ordinary Differential Equations

In this section, we propose a generalized version of HBNODE (GHBNODE), see (3.56), to mitigate the potential blow-up issue in training ODE-based models. In our experiments, we observe that $\boldsymbol{h}(t)$ of ANODEs [DDT19], SONODEs [Nor+20], and HBNODEs (3.43) usually grows much faster than that of NODEs. The fast growth of $\boldsymbol{h}(t)$ can lead to finite-time blow up. As an illustration, we compare the performance of NODE, ANODE, SONODE, HBNODE, and GHBNODE on the Silverbox task as in [Nor+20]. The goal of the task is to learn the voltage of an electronic circuit that resembles a Duffing oscillator, where the input voltage $V_1(t)$ is used to predict the output $V_2(t)$. Similar to the setting in [Nor+20], we first augment ANODE by 1 dimension with 0-augmentation and augment SONODE, HBNODE, and GHBNODE with a dense network. We use a simple dense layer to parameterize f for all five models, with an extra input term for $V_1(t)$[1]. For both HBNODE and GHBNODE,

[1]Here, we exclude an $\boldsymbol{h}^3$ term that appeared in the original Duffing oscillator model because including it would result in finite-time explosion.

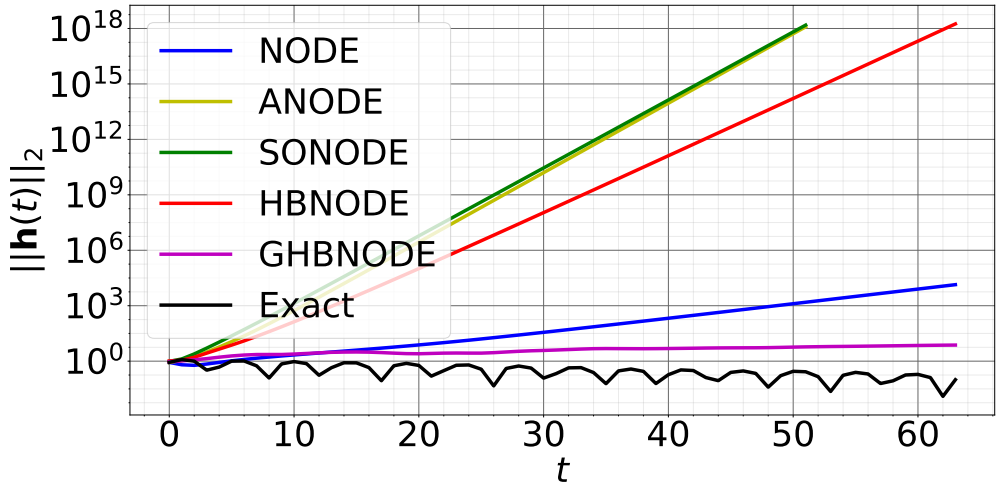


Figure 3.3: Contrasting $\boldsymbol{h}(t)$ for different models. $\boldsymbol{h}(t)$ in ANODE, SONODE, and HBNODE grows much faster than that in NODE. GHBNODE controls the growth of $\boldsymbol{h}(t)$ effectively when t is large.

we set the damping parameter γ to be $\textbf{Sigmoid}(-3)$. For GHBNODE (3.56) below, we set $\sigma(\cdot)$ to be the `hardtanh` function with bound $[-5, 5]$ and $\xi = \ln(2)$. As shown in Fig. 3.3, compared to the vanilla NODE, the L_2 norm of $\boldsymbol{h}(t)$ grows much faster when a higher order NODE is used, which leads to blow-up during training. Similar issues arise in the time series experiments (see Sec. 3.5.3.3), where SONODE blows up during long term integration in time, and HBNODE suffers from the same issue with same initialization.

To alleviate the problem above, we propose the following generalized HBNODE

$$\frac{d\boldsymbol{h}(t)}{dt} = \sigma(\boldsymbol{m}(t)); \quad \frac{d\boldsymbol{m}(t)}{dt} = -\gamma \boldsymbol{m}(t) + f(\boldsymbol{h}(t), t, \theta) - \xi \boldsymbol{h}(t), \tag{3.56}$$

where $\sigma(\cdot)$ is a nonlinear activation, which is set as tanh in our experiments. The positive hyperparameters $\gamma, \xi > 0$ are tunable or learnable. In the trainable case, we let $\gamma = \epsilon \cdot \textbf{Sigmoid}(\omega)$ as in HBNODE, and $\xi = \text{softplus}(\chi)$ to ensure that $\gamma, \xi \geq 0$. Here, we integrate two main ideas into the design of GHBNODE: (i) We incorporate the gating mechanism used in LSTM [HS97] and GRU [Cho+14], which can suppress the aggregation

of $\boldsymbol{m}(t)$; (ii) Following the idea of skip connection [He+16b], we add the term $\xi\boldsymbol{h}(t)$ into the governing equation of $\boldsymbol{m}(t)$, which benefits training and generalization of GHBNODEs. Fig. 3.3 shows that GHBNODE can indeed control the growth of $\boldsymbol{h}(t)$ effectively.

Proposition 3 (Adjoint equations for GHBNODEs)**.** *The adjoint states* $\boldsymbol{a}_{\boldsymbol{h}}(t) := \partial\mathcal{L}/\partial\boldsymbol{h}(t)$, $\boldsymbol{a}_{\boldsymbol{m}}(t) := \partial\mathcal{L}/\partial\boldsymbol{m}(t)$ *for the GHBNODE* (3.56) *satisfy the following first-order ODE system*

$$\frac{\partial\boldsymbol{a}_{\boldsymbol{h}}(t)}{\partial t} = -\boldsymbol{a}_{\boldsymbol{m}}(t)\Big(\frac{\partial f}{\partial\boldsymbol{h}}(\boldsymbol{h}(t),t,\theta) - \xi\boldsymbol{I}\Big), \quad \frac{\partial\boldsymbol{a}_{\boldsymbol{m}}(t)}{\partial t} = -\boldsymbol{a}_{\boldsymbol{h}}(t)\sigma'(\boldsymbol{m}(t)) + \gamma\boldsymbol{a}_{\boldsymbol{m}}(t). \tag{3.57}$$

Proof. The coupled form of GHBNODE is a first-order ODE system of the form

$$\frac{\partial}{\partial t}\begin{bmatrix}\boldsymbol{h}\\ \boldsymbol{m}\end{bmatrix} = \begin{bmatrix}\sigma(\boldsymbol{m})\\ -\gamma\boldsymbol{m} + f(\boldsymbol{h}(t),t,\theta) - \xi\boldsymbol{h}(t)\end{bmatrix}, \quad \begin{bmatrix}\boldsymbol{h}\\ \boldsymbol{m}\end{bmatrix}(t_0) = \begin{bmatrix}\boldsymbol{h}_{t_0}\\ \boldsymbol{m}_{t_0}\end{bmatrix}. \tag{3.58}$$

Denote the final state as

$$\begin{bmatrix}\boldsymbol{h}(T)\\ \boldsymbol{m}(T)\end{bmatrix} = \begin{bmatrix}\boldsymbol{h}_T\\ \boldsymbol{m}_T\end{bmatrix} = \boldsymbol{z}_T. \tag{3.59}$$

Using the conclusions from 3.2.2.1, we have the adjoint equation

$$\frac{\partial\boldsymbol{A}(t)}{\partial t} = -\boldsymbol{A}(t)\begin{bmatrix}\boldsymbol{0} & \sigma'(\boldsymbol{m})\\ \frac{\partial f}{\partial\boldsymbol{h}} - \xi\boldsymbol{I} & -\gamma\boldsymbol{I}\end{bmatrix}, \quad \boldsymbol{A}(T) = -\boldsymbol{I}, \quad \boldsymbol{a}(t) = -\frac{d\mathcal{L}}{d\boldsymbol{z}_T}\boldsymbol{A}(t). \tag{3.60}$$

Let $\begin{bmatrix}\boldsymbol{a}_{\boldsymbol{h}} & \boldsymbol{a}_{\boldsymbol{m}}\end{bmatrix} = \boldsymbol{a}$, by linearity we have

$$\frac{\partial\begin{bmatrix}\boldsymbol{a}_{\boldsymbol{h}} & \boldsymbol{a}_{\boldsymbol{m}}\end{bmatrix}}{\partial t} = -\begin{bmatrix}\boldsymbol{a}_{\boldsymbol{h}} & \boldsymbol{a}_{\boldsymbol{m}}\end{bmatrix}\begin{bmatrix}\boldsymbol{0} & \sigma'(\boldsymbol{m})\\ \frac{\partial f}{\partial\boldsymbol{h}} - \xi\boldsymbol{I} & -\gamma\boldsymbol{I}\end{bmatrix}, \quad \begin{bmatrix}\boldsymbol{a}_{\boldsymbol{h}}(T) & \boldsymbol{a}_{\boldsymbol{m}}(T)\end{bmatrix} = \begin{bmatrix}\frac{d\mathcal{L}}{d\boldsymbol{h}_T} & \frac{d\mathcal{L}}{d\boldsymbol{m}_T}\end{bmatrix}, \tag{3.61}$$

which gives us the initial conditions at $t = T$, and the simplified first-order ODE system

$$\frac{\partial\boldsymbol{a}_{\boldsymbol{h}}}{\partial t} = -\boldsymbol{a}_{\boldsymbol{m}}\Big(\frac{\partial f}{\partial\boldsymbol{h}} - \xi\boldsymbol{I}\Big), \quad \frac{\partial\boldsymbol{a}_{\boldsymbol{m}}}{\partial t} = -\boldsymbol{a}_{\boldsymbol{h}}\sigma'(\boldsymbol{m}) + \gamma\boldsymbol{a}_{\boldsymbol{m}}. \tag{3.62}$$

□

Though the adjoint state of the GHBNODE (3.57) does not satisfy the exact heavy ball ODE, based on our empirical study, it also significantly reduces the backward NFEs.

3.4 Learning long-term dependencies – Vanishing gradient

It is known that the vanishing and exploding gradients are two bottlenecks for training recurrent neural networks (RNNs) with long-term dependencies [BSF94; PMB13]. Recurrent cells are the building blocks of RNNs, and can be mathematically written as

$$\boldsymbol{h}_t = \sigma(\boldsymbol{U}\boldsymbol{h}_{t-1} + \boldsymbol{W}\boldsymbol{x}_t + \boldsymbol{b}),\ \boldsymbol{x}_t \in \mathbb{R}^d,\ \text{for } t = 1, 2, \cdots, T, \tag{3.63}$$

where $\boldsymbol{h}_t \in \mathbb{R}^h$ is the hidden state, $\boldsymbol{U} \in \mathbb{R}^{h\times h}, \boldsymbol{W} \in \mathbb{R}^{h\times d}$, and $\boldsymbol{b} \in \mathbb{R}^h$ are trainable parameters; $\sigma(\cdot)$ is a nonlinear activation function, e.g., sigmoid. Backpropagation through time is a popular algorithm for training RNNs, which usually results in exploding or vanishing gradients [BSF94]. Thus RNNs may fail to learn long term dependencies. As an illustration, let $\boldsymbol{h}_T$ and $\boldsymbol{h}_t$ be the state vectors at the timestamps T and t ($T \gg t$), respectively. Assume $\mathcal{L}$ is the loss to minimize, then

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_t} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} \cdot \frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_t} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} \cdot \prod_{k=t}^{T-1} \frac{\partial \boldsymbol{h}_{k+1}}{\partial \boldsymbol{h}_k} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} \cdot \prod_{k=t}^{T-1} (\boldsymbol{D}_k \boldsymbol{U}^\top), \tag{3.64}$$

where $\boldsymbol{D}_k = \mathrm{d}iag(\sigma'(\boldsymbol{U}\boldsymbol{h}_k + \boldsymbol{W}\boldsymbol{x}_{k+1}))$ is a diagonal matrix with $\sigma'(\boldsymbol{U}\boldsymbol{h}_k + \boldsymbol{W}\boldsymbol{x}_{k+1})$ being its diagonal entries. $\| \prod_{k=t}^{T-1} (\boldsymbol{D}_k \boldsymbol{U}^\top) \|_2$ tends to either vanish or explode [BSF94]. When applying RNNs to sequence applications with $\boldsymbol{x} = (\boldsymbol{x}_1, \cdots, \boldsymbol{x}_T)$ be an input sequence of length T and $\boldsymbol{y} = (y_1, \cdots, y_T)$ be the sequence of labels, we let $\mathcal{L}_t$ be the loss at the timestamp t and the total loss on the whole sequence be

$$\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t, \tag{3.65}$$

the vanishing or exploding issue can be shown following (3.64).

The exploding gradients issue can be effectively resolved via gradient clipping, training loss regularization, etc [PMB13; Eri+21]. Thus in practice the vanishing gradient is the major issue for learning long-term dependencies [PMB13]. As the continuous analogue of RNN, NODEs as well as their hybrid ODE-RNN models, may also suffer from vanishing in the adjoint state $\boldsymbol{a}(t) := \partial \mathcal{L} / \partial \boldsymbol{h}(t)$ [LH20]. When the vanishing gradient issue happens, $\boldsymbol{a}(t)$ goes to $\mathbf{0}$ quickly as $T - t$ increases, then $d\mathcal{L}/d\theta$ in (3.2) will be independent of these $\boldsymbol{a}(t)$.

3.4.1 Linear Analysis on NODEs and HBNODEs

Nonlinear ODEs do not have a general analytic solution, making it difficult to estimate their qualitative behavior. Thus, analyzing their linear counterparts may provide intuition on how these systems behave. Suppose the linearized system of ODEs satisfies

$$\frac{d\boldsymbol{h}}{dt} = \boldsymbol{L}(t,\theta)\boldsymbol{h}, \tag{3.66}$$

then its solution will satisfy

$$\boldsymbol{h}(t) = \exp\left\{\int_{t_0}^{t} \boldsymbol{L}(\tau,\theta)d\tau\right\}\boldsymbol{h}(t_0). \tag{3.67}$$

The reverse-time state equation will have the solution

$$\boldsymbol{h}(t) = \exp\left\{-\int_{t}^{T} \boldsymbol{L}(\tau,\theta)d\tau\right\}\boldsymbol{h}(T). \tag{3.68}$$

The adjoint equation will be linear and independent of $\boldsymbol{h}$

$$\frac{d\boldsymbol{a}}{dt}(t) = -\boldsymbol{a}(t)\boldsymbol{L}(t,\theta), \tag{3.69}$$

thus with the solution

$$\boldsymbol{a}(t) = \boldsymbol{a}(T)\exp\left\{\int_{t}^{T} \boldsymbol{L}(\tau,\theta)d\tau\right\}. \tag{3.70}$$

Therefore, the behavior of adjoint of NODEs when linear is associated with the spectral properties of $\int_t^T \boldsymbol{L}(\tau,\theta)d\tau$. In particular, if $\boldsymbol{h}(t)$ is decaying swiftly, its adjoint $\boldsymbol{a}(t)$ will also be vanishing.

3.4.2 Generic Analysis on Vanishing Gradients of NODEs and (G)HBNODEs

GHBNODE can be viewed as a system of higher dimensional NODE as in equation (3.58). With $\boldsymbol{z} = \begin{bmatrix}\boldsymbol{h}\\ \boldsymbol{m}\end{bmatrix}$, $\boldsymbol{z}$ satisfies NODE equation, and therefore it also satisfies the equation for relative gradient information as in (3.30),

$$\frac{d\boldsymbol{z}_T}{d\boldsymbol{z}_t} = \exp\left\{-\int_{T}^{t} \frac{\partial f}{\partial \boldsymbol{z}}(\boldsymbol{z}_0(\tau),\tau,\theta_0)d\tau\right\}. \tag{3.71}$$

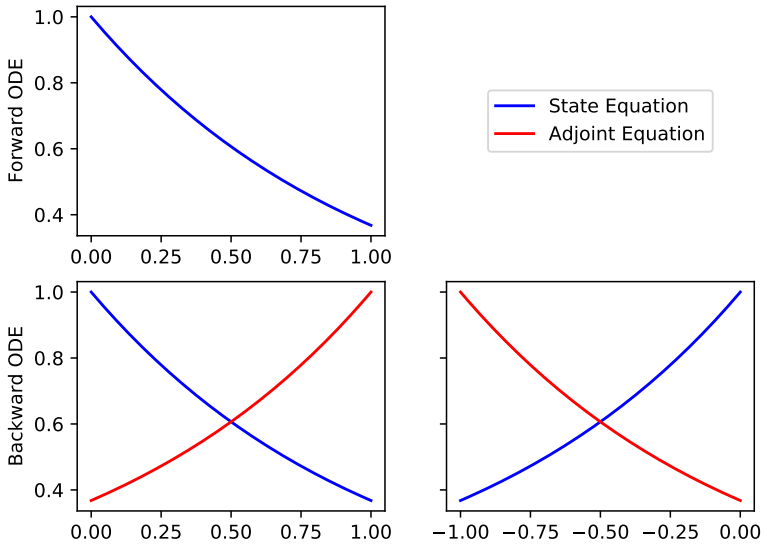


Figure 3.4: 1-D linear example of NODEs satisfying $\frac{d\boldsymbol{h}}{dt}(t) = -\boldsymbol{h}(t)$, with initial condition $\boldsymbol{h}(0) = 1$ and loss function $\mathcal{L}(\boldsymbol{h}(1)) = \frac{1}{2}\boldsymbol{h}(1)^2$. Upper left is the solution of $\boldsymbol{h}$ during forward iteration, solved from left to right. Lower left is the solution of $\boldsymbol{h}$ during backward iteration, solved from right to left. Lower right is the solution of $\boldsymbol{h}$ during backward iteration with $t \to -t$ flip, solved from left to right. When integrating from left to right, state equation in forward iteration is qualitatively similar to adjoint equation in backward iteration, whereas state equation in backward iteration is qualitatively different.

By definition of multivariate derivatives, we have

$$\frac{d\boldsymbol{z}_T}{d\boldsymbol{z}_t} = \begin{bmatrix} \frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_t} & \frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{m}_t} \\ \frac{\partial \boldsymbol{m}_T}{\partial \boldsymbol{h}_t} & \frac{\partial \boldsymbol{m}_T}{\partial \boldsymbol{m}_t} \end{bmatrix}, \tag{3.72}$$

and

$$\frac{\partial f}{\partial \boldsymbol{z}} = \begin{bmatrix} \boldsymbol{0} & \frac{\partial \sigma}{\partial \boldsymbol{m}} \\ \frac{\partial f}{\partial \boldsymbol{h}} - \xi \boldsymbol{I} & -\gamma \boldsymbol{I} \end{bmatrix}. \tag{3.73}$$

With equations (3.72) and (3.73), we can rewrite equation (3.71) in terms of $\boldsymbol{h}$ and $\boldsymbol{m}$ as

$$\begin{bmatrix} \frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_t} & \frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{m}_t} \\ \frac{\partial \boldsymbol{m}_T}{\partial \boldsymbol{h}_t} & \frac{\partial \boldsymbol{m}_T}{\partial \boldsymbol{m}_t} \end{bmatrix} = \exp\left\{ -\int_T^t \begin{bmatrix} \mathbf{0} & \frac{\partial \sigma}{\partial \boldsymbol{m}} \\ \frac{\partial f}{\partial \boldsymbol{h}} - \xi \boldsymbol{I} & -\gamma \boldsymbol{I} \end{bmatrix} \right\} ds. \tag{3.74}$$

In particular, since HBNODEs are GHBNODEs with $\xi = 0$ and σ being the identity map, the gradient equation of HBNODEs takes the form

$$\begin{bmatrix} \frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_t} & \frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{m}_t} \\ \frac{\partial \boldsymbol{m}_T}{\partial \boldsymbol{h}_t} & \frac{\partial \boldsymbol{m}_T}{\partial \boldsymbol{m}_t} \end{bmatrix} = \exp\left\{ -\int_T^t \begin{bmatrix} \mathbf{0} & \boldsymbol{I} \\ \frac{\partial f}{\partial \boldsymbol{h}} & -\gamma \boldsymbol{I} \end{bmatrix} \right\} ds. \tag{3.75}$$

Thus, we have the following expressions for the adjoint states of the NODE and HBNODE:

- For NODE, we have

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_t} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} \frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_t} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} \exp\Big\{ -\int_T^t \frac{\partial f}{\partial \boldsymbol{h}}(\boldsymbol{h}(s), s, \theta) ds \Big\}. \tag{3.76}$$

- For GHBNODE[2], from (3.49) we can derive

$$\begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_t} & \frac{\partial \mathcal{L}}{\partial \boldsymbol{m}_t} \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} & \frac{\partial \mathcal{L}}{\partial \boldsymbol{m}_T} \end{bmatrix} \begin{bmatrix} \frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_t} & \frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{m}_t} \\ \frac{\partial \boldsymbol{m}_T}{\partial \boldsymbol{h}_t} & \frac{\partial \boldsymbol{m}_T}{\partial \boldsymbol{m}_t} \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} & \frac{\partial \mathcal{L}}{\partial \boldsymbol{m}_T} \end{bmatrix} \exp\Big\{ -\underbrace{\int_T^t \begin{bmatrix} \mathbf{0} & \frac{\partial \sigma}{\partial \boldsymbol{m}} \\ (\frac{\partial f}{\partial \boldsymbol{h}} - \xi \boldsymbol{I}) & -\gamma \boldsymbol{I} \end{bmatrix} ds}_{:=\boldsymbol{M}} \Big\}. \tag{3.77}$$

Note that the matrix exponential is directly related to its eigenvalues. By Schur decomposition, there exists an orthogonal matrix $\boldsymbol{Q}$ and an upper triangular matrix $\boldsymbol{U}$, where the diagonal entries of $\boldsymbol{U}$ are eigenvalues of $\boldsymbol{Q}$ ordered by their real parts, such that

$$-\boldsymbol{M} = \boldsymbol{Q}\boldsymbol{U}\boldsymbol{Q}^\top \Longrightarrow \exp\{-\boldsymbol{M}\} = \boldsymbol{Q} \exp\{\boldsymbol{U}\} \boldsymbol{Q}^\top. \tag{3.78}$$

Let $\boldsymbol{v}^\top := \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} & \frac{\partial \mathcal{L}}{\partial \boldsymbol{m}_T} \end{bmatrix} \boldsymbol{Q}$, then (3.77) can be rewritten as

$$\begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_t} & \frac{\partial \mathcal{L}}{\partial \boldsymbol{m}_t} \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} & \frac{\partial \mathcal{L}}{\partial \boldsymbol{m}_T} \end{bmatrix} \exp\{-\boldsymbol{M}\} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} & \frac{\partial \mathcal{L}}{\partial \boldsymbol{m}_T} \end{bmatrix} \boldsymbol{Q} \exp\{\boldsymbol{U}\} \boldsymbol{Q}^\top = \boldsymbol{v}^\top \exp\{\boldsymbol{U}\} \boldsymbol{Q}^\top. \tag{3.79}$$

[2]HBNODE can be seen as a special GHBNODE with $\xi = 0$ and σ be the identity map.

By taking the L_2 norm in (3.79) and dividing both sides by $\left\| \left[\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} \ \frac{\partial \mathcal{L}}{\partial \boldsymbol{m}_T} \right] \right\|_2$, we arrive at

$$\frac{\left\| \left[\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_t} \ \frac{\partial \mathcal{L}}{\partial \boldsymbol{m}_t} \right] \right\|_2}{\left\| \left[\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} \ \frac{\partial \mathcal{L}}{\partial \boldsymbol{m}_T} \right] \right\|_2} = \frac{\left\| \boldsymbol{v}^\top \exp\{\boldsymbol{U}\} \boldsymbol{Q}^\top \right\|_2}{\left\| \boldsymbol{v}^\top \boldsymbol{Q}^\top \right\|_2} = \frac{\left\| \boldsymbol{v}^\top \exp\{\boldsymbol{U}\} \right\|_2}{\left\| \boldsymbol{v} \right\|_2} = \left\| \boldsymbol{e}^\top \exp\{\boldsymbol{U}\} \right\|_2, \tag{3.80}$$

i.e., $\left\| \left[\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_t} \ \frac{\partial \mathcal{L}}{\partial \boldsymbol{m}_t} \right] \right\|_2 = \left\| \boldsymbol{e}^\top \exp\{\boldsymbol{U}\} \right\|_2 \left\| \left[\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} \ \frac{\partial \mathcal{L}}{\partial \boldsymbol{m}_T} \right] \right\|_2$ where $\boldsymbol{e} = \boldsymbol{v} / \|\boldsymbol{v}\|_2$.

Proposition 4. *The eigenvalues of $-\boldsymbol{M}$ can be paired so that the sum of each pair equals $(t-T)\gamma$.*

Proof. Let $\boldsymbol{F} = \frac{1}{t-T} \int_T^t \frac{\partial f}{\partial \boldsymbol{h}}(\boldsymbol{h}(s), s, \theta) ds - \xi \boldsymbol{I}$, $\boldsymbol{J} = \frac{1}{t-T} \int_T^t \frac{\partial \sigma}{\partial \boldsymbol{m}}(\boldsymbol{m}(s)) ds$, and $\boldsymbol{H} = \frac{1}{t-T} \boldsymbol{M}$, then we have the following equation

$$\boldsymbol{H} = \frac{1}{t-T} \boldsymbol{M} = \begin{bmatrix} 0 & \boldsymbol{J} \\ \boldsymbol{F} & -\gamma \boldsymbol{I} \end{bmatrix}. \tag{3.81}$$

As $(\lambda + \gamma)\boldsymbol{I}$ commutes with any matrix $\boldsymbol{F}$, the characteristics polynomials of $\boldsymbol{H}$ and $\boldsymbol{JF}$ satisfy the relation

$$ch_{\boldsymbol{H}}(\lambda) = \det(\lambda \boldsymbol{I} - \boldsymbol{H}) = \det \begin{bmatrix} \lambda \boldsymbol{I} & -\boldsymbol{J} \\ -\boldsymbol{F} & (\lambda + \gamma)\boldsymbol{I} \end{bmatrix} = det(\lambda(\lambda + \gamma)\boldsymbol{I} - \boldsymbol{JF}) = -ch_{\boldsymbol{JF}}(\lambda(\lambda + \gamma)). \tag{3.82}$$

Since the characteristics polynomial of $\boldsymbol{JF}$ splits in the field $\mathbb{C}$ of complex numbers, i.e. $ch_{\boldsymbol{JF}}(x) = \prod_{i=1}^n (x - \lambda_{\boldsymbol{JF},i})$, we have

$$ch_{\boldsymbol{H}}(\lambda) = -ch_{\boldsymbol{JF}}(\lambda(\lambda + \gamma)) = -\prod_{i=1}^n (\lambda(\lambda + \gamma) - \lambda_{\boldsymbol{JF},i}). \tag{3.83}$$

Therefore, the eigenvalues of $\boldsymbol{H}$ appear in n pairs with each pair satisfying the quadratic equation

$$\lambda(\lambda + \gamma) - \lambda_{\boldsymbol{JF},i} = 0. \tag{3.84}$$

By Vieta's formulas, the sum of these pairs are all $-\gamma$. Therefore, the eigenvalues of $\boldsymbol{M}$ comes in n pairs and the sum of each pair is $-(t-T)\gamma$. □

For a given constant $a > 0$, we can group the upper triangular matrix $\exp\{\boldsymbol{U}\}$ as follows

$$\exp\{\boldsymbol{U}\} := \begin{bmatrix} \exp\{\boldsymbol{U}_L\} & \boldsymbol{P} \\ \boldsymbol{0} & \exp\{\boldsymbol{U}_V\} \end{bmatrix}, \tag{3.85}$$

where the diagonal of $\boldsymbol{U}_L$ ($\boldsymbol{U}_V$) contains eigenvalues of $-\boldsymbol{M}$ that are no less (greater) than $(t-T)a$. Then, we have $\|\boldsymbol{e}^\top \exp\{\boldsymbol{U}\}\|_2 \geq \|\boldsymbol{e}_L^\top \exp\{\boldsymbol{U}_L\}\|_2$ where the vector $\boldsymbol{e}_L$ denotes the first m columns of $\boldsymbol{e}$ with m be the number of columns of $\boldsymbol{U}_L$. By choosing $0 \leq \gamma \leq 2a$, for every pair of eigenvalues of $-\boldsymbol{M}$ there is at least one eigenvalue whose real part is no less than $(t-T)a$. Therefore, $\exp\{\boldsymbol{U}_L\}$ decays at a rate at most $(t-T)a$, and the dimension of $\boldsymbol{U}_L$ is at least $N \times N$. We avoid exploding gradients by clipping the L_2 norm of the adjoint states similar to that used for training RNNs.

In contrast, all eigenvalues of the matrix $\int_T^t \partial f/\partial \boldsymbol{h} ds$ in (3.76) for NODE can be very positive or negative, resulting in exploding or vanishing gradients. As an illustration, we consider the benchmark Walker2D kinematic simulation task that requires learning long-term dependencies effectively [LH20; Bro+16]. We train ODE-RNN [RCD19] and (G)HBNODE-RNN on this benchmark dataset, and the detailed experimental settings are provided in Sec. 3.5.3.3. Figure 3.4.2 plots $\|\partial \mathcal{L}/\partial \boldsymbol{h}_t\|_2$ for ODE-RNN and $\|[\partial \mathcal{L}/\partial \boldsymbol{h}_t \ \partial \mathcal{L}/\partial \boldsymbol{m}_t]\|_2$ for (G)HBNODE-RNN, showing that the adjoint state of ODE-RNN vanishes quickly, while that of (G)HBNODE-RNN does not vanish even when the gap between T and t is very large.

3.5 Experimental Results

In this section, we compare the performance of the proposed HBNODE and GHBNODE with existing ODE-based models, including NODE [Che+18], ANODE [DDT19], and SONODE [Nor+20] on the benchmark point cloud separation, image classification, learning dynamical systems, and kinematic simulation. For all the experiments, we use Adam [KB15] as the benchmark optimization solver (the learning rate and batch size for each experiment are listed in Table 3.1) and Dormand–Prince-45 as the numerical ODE solver. For HBNODE and GHBNODE, we set $\gamma = \textbf{Sigmoid}(\theta)$, where θ is a trainable weight initialized as $\theta = -3$.

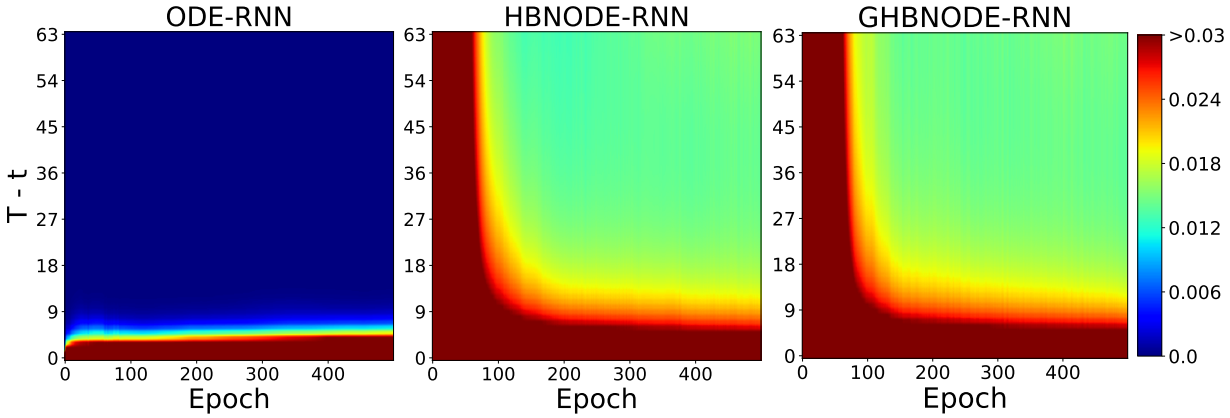


Figure 3.5: Plot of the the L_2-norm of the adjoint states for ODE-RNN and (G)HBNODE-RNN back-propagated from the last time stamp. The adjoint state of ODE-RNN vanishes quickly when the gap between the final time T and intermediate time t becomes larger, while the adjoint states of (G)HBNODE-RNN decays much more slowly. This implies that (G)HBNODE-RNN is more effective in learning long-term dependency than ODE-RNN.

The network architecture used to parameterize $f(\boldsymbol{h}(t), t, \theta)$ for each experiment below are described in the corresponding sections. All experiments are conducted on a server with 2 NVIDIA Titan Xp GPUs.

Table 3.1: The batch size and learning rate for different datasets.

Dataset	Point Cloud	MNIST	CIFAR10	Plane Vibration	Walker2D
Batch Size	50	64	64	64	256
Learning Rate	0.01	0.001	0.001	0.0001	0.003

We first list some common settings below:

- NODE and ANODE do not have initial layers.
- For SONODE $n^* = 2n$, and for other ones $n^* = n$.
- tpad: Padding with time t within ODE. i.e., transform the shape $c \times x \times y$ to $(c+1) \times x \times y$ by concatenating with a tensor of shape $1 \times x \times y$ filled with all t.

- For all tasks, we use learnable γ with $\epsilon = 1$ for both HBNODE and GHBNODE, and learnable ξ.

- fc_n: a fully connected layer with output dimension to be n.

3.5.1 Point cloud separation

In this subsection, we consider the two-dimensional point cloud separation benchmark. A total of 120 points are sampled, in which 40 points are drawn uniformly from the circle $\|\boldsymbol{r}\| < 0.5$, and 80 points are drawn uniformly from the annulus $0.85 < \|\boldsymbol{r}\| < 1.0$. This experiment aims to learn effective features to classify these two point clouds. Following [DDT19], we use a three-layer neural network to parameterize the right-hand side of each ODE-based model, integrate the ODE-based model from $t_0 = 0$ to $T = 1$, and pass the integration results to a dense layer to generate the classification results. The details of the model are specified as follows

- Initial Velocity : $\text{input}_2 \to \text{fc}_h \to \text{HTanh} \to \text{fc}_h \to \text{HTanh} \to \text{fc}_n$,

- ODE : $\text{input}_{n^*} \to \text{fc}_h \to \text{ELU} \to \text{fc}_h \to \text{ELU} \to \text{fc}_n$,

- Output : $\text{input}_n \to \text{fc}_1 \to \text{Tanh}$,

where fc_n denotes fully connected layers with n dimensional output, HTanh denotes the hard tanh activation function defined entry-wise as $[\text{HTanh}(v)]_i = \min\{5, \max\{-5, v_i\}\}$, and ELU denotes the elu function defined entry-wise as

$$[\text{ELU}(v)]_i = \begin{cases} v_i & \text{if } v_i \geq 0, \\ e^{v_i} - 1 & \text{if } v_i < 0. \end{cases} \tag{3.86}$$

The hidden dimensions are specified in table 3.2 such that models contain similar number of parameters for fair comparison. To avoid the effects of numerical error of the black-box ODE solver we set tolerance of ODE solver to be 10^{-7}.

Table 3.2: The hyper-parameters and the number of parameters for point cloud separation.

Model	NODE	ANODE	SONODE	HBNODE	GHBNODE
n (Point Cloud)	2	3	2	2	2
h (Point Cloud)	20	20	13	14	14
#Params	525	567	528	568	568

Figure 3.6 plots a randomly selected evolution of the point cloud separation for each model; we also compare the forward and backward NFEs and the training loss of these models (100 independent runs). HBNODE and GHBNODE improve training as the training loss consistently goes to zero over different runs, while ANODE and SONODE often get stuck at local minima, and NODE cannot separate the point cloud since it preserves the topology [DDT19].

3.5.2 Image classification

We compare the performance of HBNODE and GHBNODE with the existing ODE-based models on MNIST and CIFAR10 classification tasks using the same setting as in [DDT19]. The details of the models are as follows:

MNIST

- Initial Velocity : $\text{input}_{1\times 28\times 28} \to \text{conv}_{h,1} \to \text{LReLU} \to \text{conv}_{h,3} \to \text{LReLU} \to \text{conv}_{2n-1,1}$,
- ODE : $\text{input}_{n^*\times 28\times 28} \to \text{tpad} \to \text{conv}_{h,1} \to \text{ReLU} \to \text{tpad} \to \text{conv}_{h,3} \to \text{ReLU} \to \text{tpad} \to \text{conv}_{n,1}$,
- Output : $\text{input}_{n\times 28\times 28} \to \text{fc}_{10}$.

CIFAR

- Initial Velocity : $\text{input}_{3\times 28\times 28} \to \text{conv}_{h,1} \to \text{LReLU} \to \text{conv}_{h,3} \to \text{LReLU} \to \text{conv}_{2n-3,1}$,
- ODE : $\text{input}_{n^*\times 32\times 32} \to \text{tpad} \to \text{conv}_{h,1} \to \text{ReLU} \to \text{tpad} \to \text{conv}_{h,3} \to \text{ReLU} \to \text{tpad} \to \text{conv}_{n,1}$,

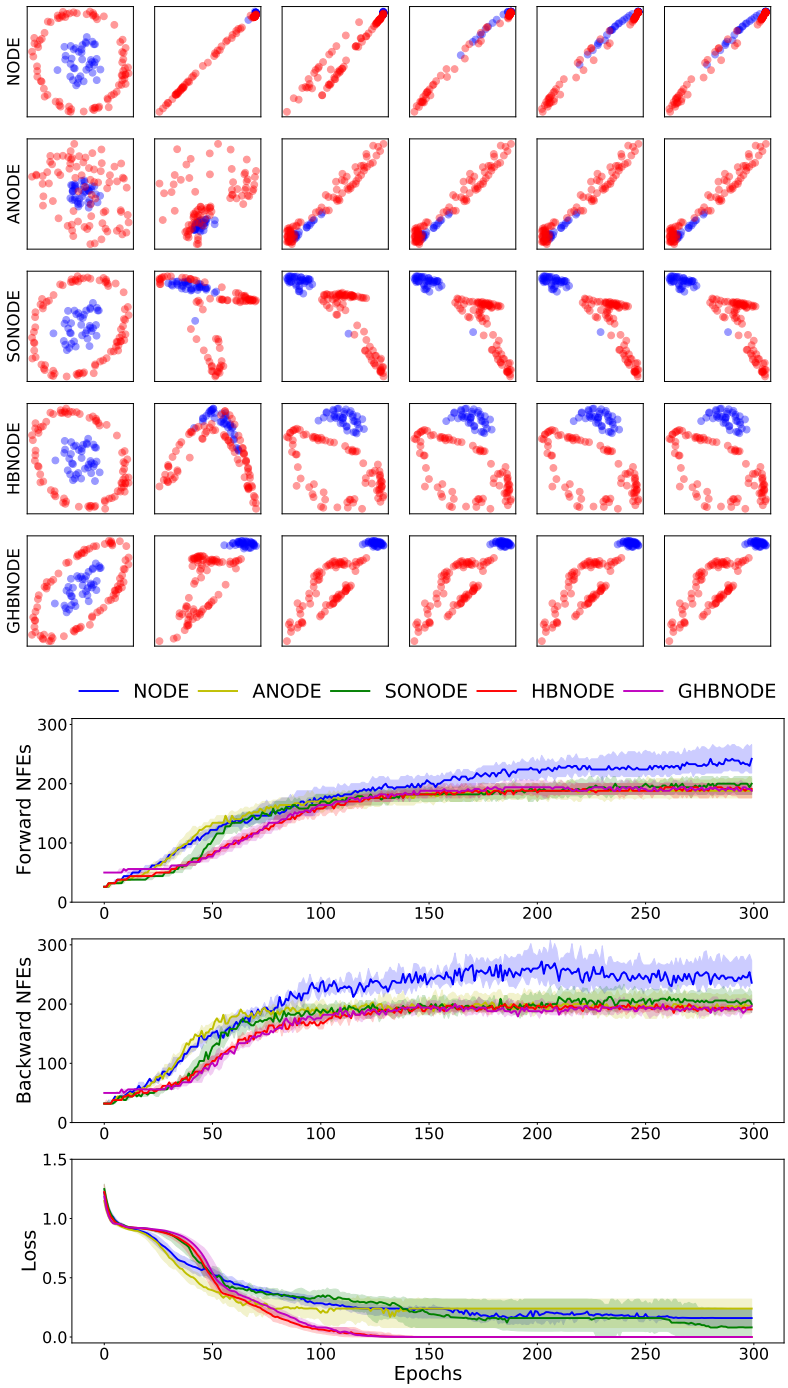


Figure 3.6: Comparison between NODE, ANODE, SONODE, HBNODE, and GHBNODE for two-dimensional point cloud separation. HBNODE and GHBNODE converge better and require less NFEs in both forward and backward propagation than the other benchmark models.

- Output : $\text{input}_{n\times 32\times 32} \to \text{fc}_{10}$.

In the specification above, $\text{conv}_{h,i}$ denotes convolution layers with h dimension and i channels of output, ReLU denotes the ReLU activation function (it is entry-wise defined as $[\text{ReLU}(v)]_i = \max\{0, v_i\}$), LReLU denotes the leaky ReLU function defined entry-wise as

$$[\text{LReLU}(v)]_i = \begin{cases} v_i & \text{if } v_i \geq 0, \\ 0.3v_i & \text{if } v_i < 0, \end{cases} \tag{3.87}$$

and tpad denotes padding with time t within ODE. i.e., transform the shape $c \times x \times y$ to $(c+1) \times x \times y$ by concatenating with a tensor of shape $1 \times x \times y$ filled with all t.

The hyper-parameters are chosen so that total number of parameters for each model are relative similar, as specified in table 3.3. For a given input image of the size $c \times h \times w$, we first augment the number of channel from c to $c + p$ with the augmentation dimension p dependent on each method[3]. Moreover, for SONODE, HBNODE and GHBNODE, we further include velocity or momentum with the same shape as the augmented state.

Table 3.3: The hyper-parameters and the number of parameters for image classification.

Model	NODE	ANODE	SONODE	HBNODE	GHBNODE
n (MNIST)	1	6	5	5	6
h (MNIST)	92	64	50	50	45
n (CIFAR)	3	13	12	12	12
h (CIFAR)	125	64	50	51	51
#Params (MNIST)	85,315	85,462	86,179	85,931	85,235
#Params (CIFAR10)	173,611	172,452	171,635	172,916	172,916

NFEs. As shown in Figs. 3.1 and 3.7, the NFEs grow rapidly with training of the NODE, resulting in an increasingly complex model with reduced performance and the possibility of blow up. Input augmentation has been verified to effectively reduce the NFEs, as both

[3]We set $p = 0, 5, 4, 4, 5/0, 10, 9, 9, 9$ on MNIST/CIFAR10 for NODE, ANODE, SONODE, HBNODE, and GHBNODE, respectively.

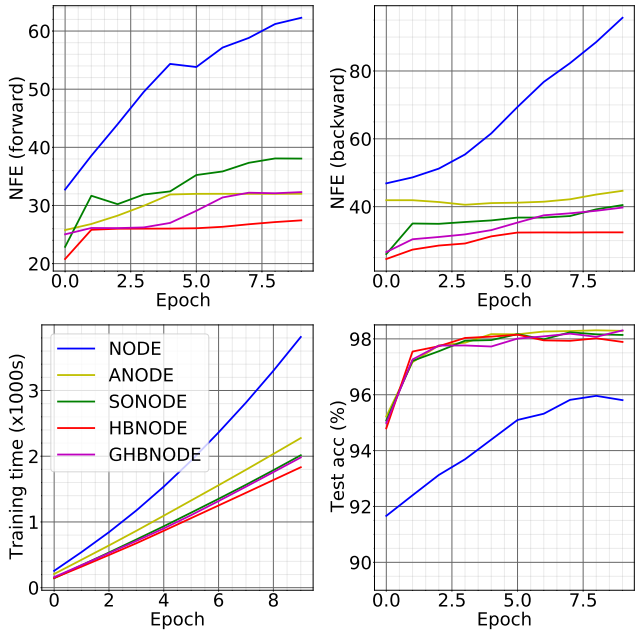


Figure 3.7: Contrasting NODE [Che+18], ANODE [DDT19], SONODE [Nor+20], HBNODE, and GHBNODE for MNIST classification in NFE, training time, and test accuracy. (Tolerance: 10^{-5}).

ANODE and SONODE require fewer forward NFEs than NODE for the MNIST and CIFAR10 classification. However, input augmentation is less effective in controlling their backward NFEs. HBNODE and GHBNODE require much fewer NFEs than the existing benchmarks, especially for backward NFEs. In practice, reducing NFEs implies reducing both training and inference time, as shown in Figs. 3.1 and 3.7.

Accuracy. We also compare the accuracy of different ODE-based models for MNIST and CIFAR10 classification. As shown in Figs. 3.1 and 3.7, HBNODE and GHBNODE have slightly better classification accuracy than the other three models; this resonates with the fact that less NFEs lead to simpler models which generalize better [DDT19; Nor+20].

NFEs vs. tolerance. We further study the NFEs for different ODE-based models under different tolerances of the ODE solver using the same approach as in [Che+18]. Figure 3.8

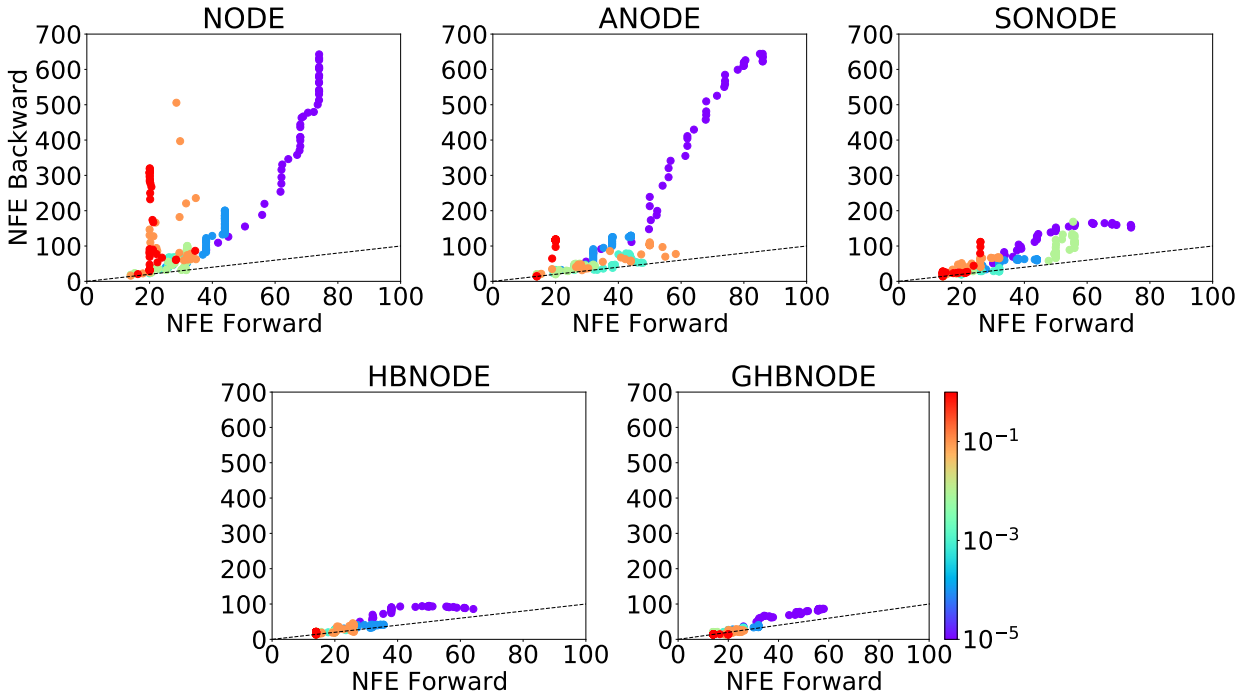


Figure 3.8: NFE vs. tolerance (shown in the colorbar) for training ODE-based models for CIFAR10 classification. Both forward and backward NFEs of HBNODE and GHBNODE grow much slower than that of NODE, ANODE, and SONODE; especially the backward NFEs.As the tolerance decreases, the advantage of HBNODE and GHBNODE in reducing NFEs becomes more significant.

depicts the forward and backward NFEs for different models under different tolerances. We see that (i) both forward and backward NFEs grow quickly when tolerance is decreased, and HBNODE and GHBNODE require much fewer NFEs than other models; (ii) under different tolerances, the backward NFEs of NODE, ANODE, and SONODE are much larger than the forward NFEs, and the difference becomes larger when the tolerance decreases. In contrast, the forward and backward NFEs of HBNODE and GHBNODE scale almost linearly with each other. This reflects that the advantage in NFEs of (G)HBNODE over the benchmarks become more significant when a smaller tolerance is used.

3.5.3 Learning dynamical systems from irregularly-sampled time series

3.5.3.1 Change of Time Intervals

Change of time interval is a technique we use in time series experiments. While training or evaluating a neural network consisting of NODEs, the time interval $[t_0, T]$ of elements within a batch might differ. In particular, for irregularly sampled time series, the time interval between two timestamps of each instance is hardly the same. Directly solving the equations results in more timestamps being recorded, which increases memory consumption and potentially introduces more interpolation error. Therefore, for the ease of computation, we align their time interval using linear change of variable. Suppose for the i-th element in the batch, we start from t_0^i and solve the equation

$$\frac{d\boldsymbol{h}^i}{dt}(t) = f(\boldsymbol{h}^i(t), t) \tag{3.88}$$

in the time interval $[t_0^i, T^i]$ to obtain $\boldsymbol{h}^i(T^i)$. In order to solve the set of equations within a batch, we define the linear change of variable $\phi^i : [0, 1] \to [t_0^i, T^i]$ such that

$$\phi^i(\tau) = t_0^i + (T^i - t_0^i)\tau, \tag{3.89}$$

and let $\boldsymbol{g}^i = \boldsymbol{h}^i \circ \phi^i$. Therefore, using the chain rule and equations (3.88) and (3.89), we have

$$\frac{d\boldsymbol{g}^i}{d\tau}(\tau) = \frac{d\boldsymbol{h}^i}{dt}(\phi^i(\tau))\frac{d\phi^i}{d\tau}(\tau) = (T^i - t_0^i) f(\boldsymbol{h}^i(\phi^i(\tau)), \phi^i(\tau)). \tag{3.90}$$

With the definition of $\boldsymbol{g}^i(\tau) = \boldsymbol{h}^i(\phi^i(\tau))$, we arrive at

$$\frac{d\boldsymbol{g}^i}{d\tau}(\tau) = (T^i - t_0^i) f(\boldsymbol{g}^i(\tau), \phi^i(\tau)). \tag{3.91}$$

Therefore, with initial conditions $\boldsymbol{g}^i(0) = \boldsymbol{h}^i(t_0^i)$, we can solve the equation (3.91) in the interval $[0, 1]$ for all elements in the batch and obtain $\boldsymbol{h}^i(T^i) = \boldsymbol{g}^i(1)$. An example is shown in Figure 3.9.

3.5.3.2 Vibrational dynamical system

In this subsection, we learn Vibrational dynamical systems from experimental measurements. In particular, we use the ODE-RNN framework [Che+18; RCD19], with the recognition

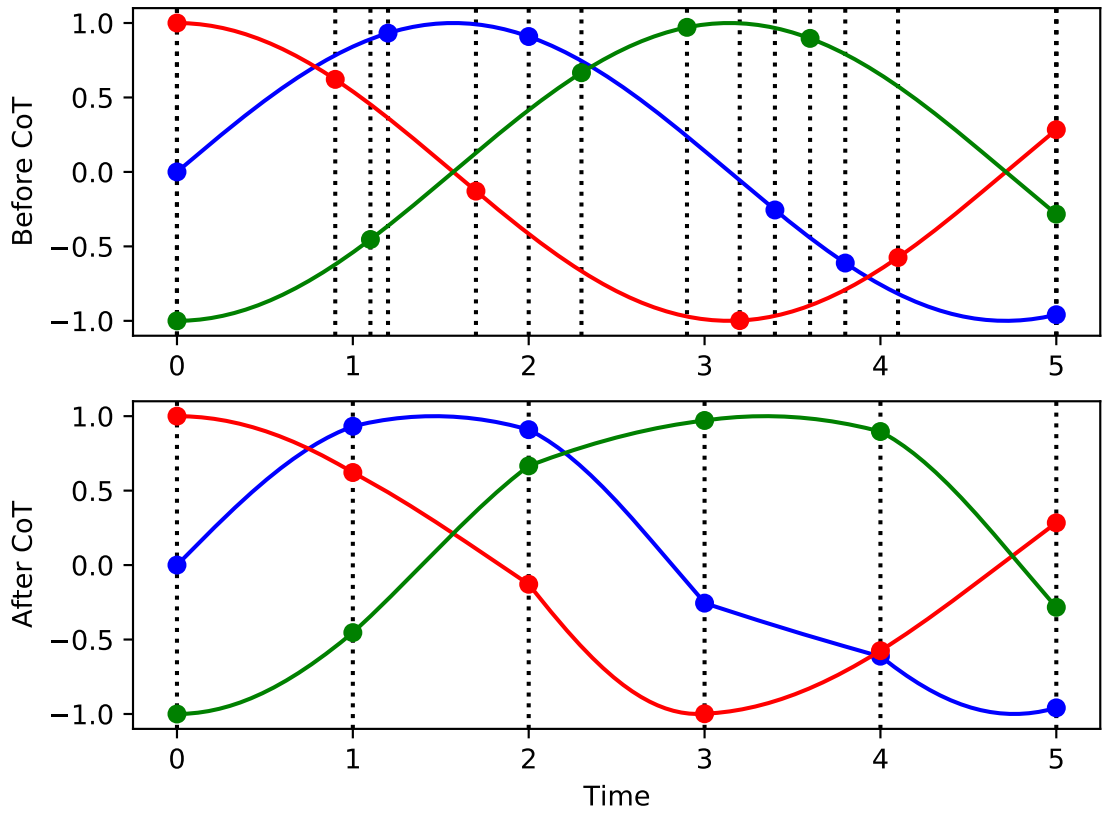


Figure 3.9: Example of Change of Time Intervals for solutions of ODEs $\frac{d^2\boldsymbol{h}}{dt^2}(t) = -\boldsymbol{h}(t)$ with different initial condition. If we compute all of the equations within the same batch, without change of time we need to capture $O(MN_T)$ output time, using $O(M^2N_T)$ memory, where M is the number of samples in batch, and N_T is the average timestamps needed for each sample, whereas with change of time we only need $O(N_T)$ output time and $O(MN_T)$ memory.

model being set to different ODE-based models, to study the vibration of an airplane dataset [NS17]. The dataset was acquired, from time 0 to 73627, by attaching a shaker underneath the right wing to provide input signals, and 5 attributes are recorded per time stamp; these attributes include voltage of input signal, force applied to aircraft, and acceleration at 3

different spots of the airplane. We randomly take out 10% of the data to make the time series irregularly-sampled. We use the first 50% of data as our train set, the next 25% as validation set, and the rest as test set. We divide each set into non-overlapping segments of consecutive 65 time stamps of the irregularly-sampled time series, with each input instance consisting of 64 time stamps of the irregularly-sampled time series, and we aim to forecast 8 consecutive time stamps starting from the last time stamp of the segment. The input is fed through the the hybrid methods in a recurrent fashion; by changing the time duration of the last step of the ODE integration, we can forecast the output in the different time stamps. The output of the hybrid method is passed to a single dense layer to generate the output time series. In our experiments, we compare different ODE-based models hybrid with RNNs. The ODE of each model is parametrized by a 3-layer network whereas the RNN is parametrized by a simple dense network as follows

- ODE : $\text{input}_{n*} \to \text{fc}_{h_1} \to \text{ReLU} \to \text{fc}_{h_2} \to \text{ReLU} \to \text{fc}_n$,
- RNN : $\text{input}_{dn+k} \to \text{fc}_{dn}$,
- Output : $\text{input}_n \to \text{fc}_5$,

where fc_n denotes fully connected layers with n dimensional output, and ReLU denotes the ReLU activation function (it is entry-wise defined as $[\text{ReLU}(v)]_i = \max\{0, v_i\}$). The hidden dimensions are specified in table 3.4 such that models contain similar number of parameters for fair comparison. The total number of parameters for ODE-RNN, ANODE-RNN, SONODE-

Table 3.4: The hyper-parameters for ODE-RNN integration models.

Model	ODE-RNN	ANODE-RNN	SONODE-RNN	HBNODE-RNN	GHBNODE-RNN
d	1	1	2	2	2
n (Plane Vibration)	21	27	19	20	20
h_1 (Plane Vibration)	63	83	19	20	20
h_2 (Plane Vibration)	84	108	19	20	20

RNN, HBNODE-RNN, and GHBNODE-RNN with 16, 22, 14, 15, 15 augmented dimensions

are 15,986, 16,730, 16,649, 16,127, and 16,127, respectively. To avoid potential error due to the ODE solver, we use a tolerance of 10^{-7}.

In training those hybrid models, we regularize the models by penalizing the L2 distance between the RNN output and the values of the next time stamp. Due to the second-order natural of the underlying dynamics [Nor+20], ODE-RNN and ANODE-RNN learn the dynamics very poorly with much larger training and test losses than the other models even they take smaller NFEs. HBNODE-RNN and GHBNODE-RNN give better prediction than SONODE-RNN using less backward NFEs as shown in figure 3.10.

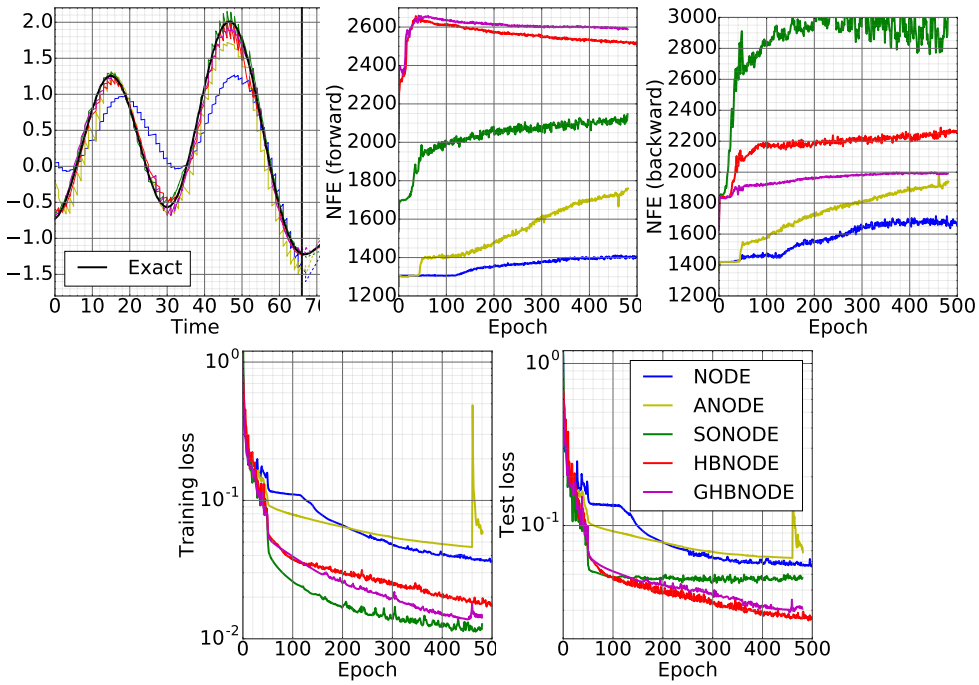


Figure 3.10: Contrasting ODE-RNN, ANODE-RNN, SONODE-RNN, HBNODE-RNN, and GHBN-ODE-RNN for learning a vibrational dynamical system. Left most: The learned curves of each model vs. the ground truth (Time: <66 for training, 66-75 for testing).

3.5.3.3 Walker2D kinematic simulation

In this subsection, we evaluate the performance of HBNODE-RNN and GHBNODE-RNN on the Walker2D kinematic simulation task, which requires learning long-term dependency

effectively [LH20]. The dataset [Bro+16] consists of a dynamical system from kinematic simulation of a person walking from a pre-trained policy, aiming to learn the kinematic simulation of the MuJoCo physics engine [TET12]. The dataset is irregularly-sampled with 10% of the data removed from the simulation. Each input consists of 64 time stamps fed though the the hybrid methods in a recurrent fashion, and the output is passed to a single dense layer to generate the output time series. The goal is to provide an auto-regressive forecast so that the output time series is as close as the input sequence shifted one time stamp to the right. We compare ODE-RNN (with 7 augmentation), ANODE-RNN (with 7 ANODE style augmentation), HBNODE-RNN (with 7 augmentation), and GHBNODE-RNN (with 7 augmentation) [4] The RNN is parametrized by a 3-layer network whereas the ODE is parametrized by a simple dense network as follows:

- ODE : $\text{input}_{n*} \to \text{fc}_n$,
- RNN : $\text{input}_{dn+k} \to \text{fc}_{h_1} \to \text{Tanh} \to \text{fc}_{h_2} \to \text{Tanh} \to \text{fc}_{dn}$,
- Output : $\text{input}_n \to \text{fc}_{17}$,

where fc_n denotes fully connected layers with n dimensional output, and $Tanh$ denotes the hyperbolic tangent function. The hidden dimensions are specified in table 3.5 such that models contain similar number of parameters for fair comparison.

Table 3.5: The hyper-parameters for ODE-RNN integration models.

Model	ODE-RNN	ANODE-RNN	SONODE-RNN	HBNODE-RNN	GHBNODE-RNN
d	1	1	2	2	2
n (Walker 2D)	24	24	23	24	24
h_1 (Walker 2D)	72	72	46	48	48
h_2 (Walker 2D)	48	48	46	48	48

[4]Here, we do not compare with SONODE-RNN since SONODE has some initialization problem on this dataset; the ODE solver encounters failure due to exponential growth over time.

The number of parameters of the above four models are 8,729, 8,815, 8,899, and 8,899, respectively. In Fig. 3.11, we compare the performance of the above four models on the Walker2D benchmark; HBNODE-RNN and GHBNODE-RNN not only require significantly less NFEs in both training (forward and backward) and in testing than ODE-RNN and ANODE-RNN, but also have much smaller training and test losses.

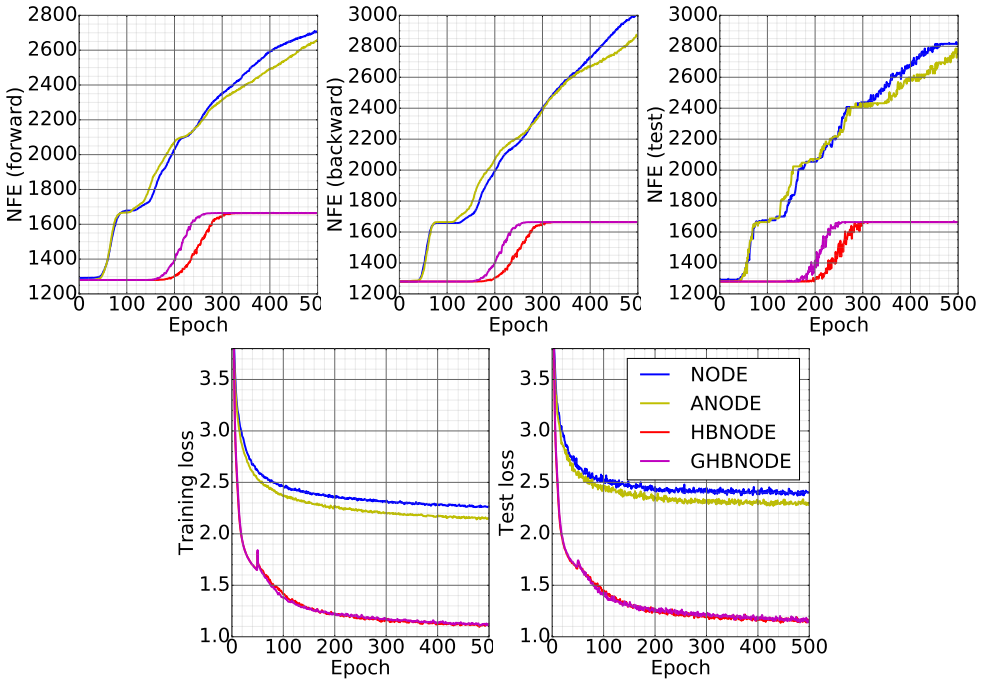


Figure 3.11: Contrasting ODE-RNN, ANODE-RNN, SONODE-RNN, HBNODE-RNN, and GHBN-ODE-RNN for the Walker-2D kinematic simulation.

3.6 Related Work

Reducing NFEs in training NODEs. Several techniques have been developed to reduce the NFEs for the forward solvers in NODEs, including weight decay [Gra+19], input augmentation [DDT19], regularizing solvers and learning dynamics [Fin+20; Kel+20; Gho+20; Pal+21], high-order ODE [Nor+20], data control [Mas+20], and depth-variance [Mas+20]. HBNODEs can reduce both forward and backward NFEs at the same time.

Second-order ODE accelerated dynamics. It has been noticed in both optimization and sampling communities that second-order ODEs with an appropriate damping term, e.g., the classical momentum and Nesterov's acceleration in discrete regime, can significantly accelerate the first-order gradient dynamics (gradient descent), e.g., [Pol64; Nes83; CFG14; SBC14; WRJ18]. Also, these second-order ODEs have been discretized via some interesting numerical schemes to design fast optimization schemes, e.g., [Shi+19].

Learning long-term dependencies. Learning long-term dependency is one of the most important goals for learning from sequential data. Most of the existing works focus on mitigating exploding or vanishing gradient issues in training RNNs, e.g., [ASB16; Wis+16; Jin+17; Vor+17; Mha+17; HWY18; Ngu+20]. Attention-based models are proposed for learning on sequential data concurrently with the effective accommodation of learning long-term dependency [Vas+17; Dev+19]. Recently, NODEs have been integrated with long-short term memory model [HS97] to learn long-term dependency for irregularly-sampled time series [LH20]. HBNODEs directly enhance learning long-term dependency from sequential data.

Momentum in neural network design. As a line of orthogonal work, the momentum has also been studied in designing neural network architecture, e.g., [MB17; Ngu+20; Li+18; San+21], which can also help accelerate training and learn long-term dependencies. These techniques can be considered as changing the neural network f in (3.1). We leave the synergistic integration of adding momentum to f with our work on changing the left-hand side of (3.1) as a future work.

3.7 Concluding Remarks

We proposed HBNODEs to reduce the NFEs in solving both forward and backward ODEs, which also improve generalization performance over the existing benchmark models. Moreover, HBNODEs alleviate vanishing gradients in training NODEs, making HBNODEs able to learn long-term dependency effectively from sequential data. In the optimization

community, Nesterov acceleration [Nes83] is also a famous algorithm for accelerating gradient descent, that achieves an optimal convergence rate for general convex optimization problems. The ODE counterpart of the Nesterov's acceleration corresponds to (3.42) with γ being replaced by a time-dependent damping parameter, e.g., $t/3$ [SBC14] or with restart [Wan+20]. The adjoint equation of the Nesterov's ODE [SBC14] is no longer a Nesterov's ODE. We notice that directly using Nesterov's ODE cannot improve the performance of the vanilla neural ODE. How to integrate Nesterov's ODE with neural ODE is an interesting future direction. Another interesting direction is connecting HBNODE with symplectic ODE-net [ZDC20] through an appropriate change of variables.

CHAPTER 4

GRAND++: Graph Neural Diffusion with A Source Term

4.1 Introduction

Graph neural networks (GNNs) are the backbone for deep learning on graphs. Recent GNN architectures include graph convolutional networks (GCNs) [KW17], ChebyNet [DBV16], GraphSAGE [HYL17], neural graph fingerprints [Duv+15], message passing neural networks [Gil+17], and graph attention networks (GATs) [Vel+]. These graph deep networks have achieved success in many applications, including computational physics and computational chemistry [Duv+15; Gil+17; Bat+16], recommender systems [MBB17; Yin+18b], and social networks [ZC18; Qiu+18]. Hyperbolic GNNs have also been proposed to enable certain kinds of data embedding with much smaller distortion [Cha+19; LNK19]. See [Bro+21] for some recent advances of GNN algorithm development and applications.

A well-known problem of GNNs is that increasing the depth of GNNs often results in a significant drop in performance on various graph learning tasks. This performance degradation has been widely interpreted as the *over-smoothing* issue of GNNs [LHW18; OS20; Che+20]. Intuitively, GNN layers update the node representation by taking a weighted average of its neighbors' features, making representations for neighboring nodes to be similar. As the GNN architecture gets deeper, all nodes' representation will become indistinguishable resulting in over-smoothing. In Sec. 4.2, we briefly show that certain GNNs have a diffusive nature which makes over-smoothing inevitable. Another interesting interpretation of the GNN performance degradation is via a *bottleneck* [AY21], since a GNN tends to represent exponentially growing information from neighbors with fixed-size vectors. Several algorithms have been proposed

to mitigate the over-smoothing of GNNs, including skip connection and dilated convolution [Li+19a], Jumping Knowledge [Xu+18], DropEdge [Ron+20], PairNorm [ZA20], graph neural diffusion (GRAND) [Cha+21a], and wave equation motivated GNNs [EHT21]. Nevertheless, developing deep GNN architectures is still in its infancy compared to the development of other deep networks.

Besides suffering from over-smoothing, we notice that the accuracy of existing GNNs drops severely when they are trained with a limited labeled data. As illustrated in Fig. 4.1, the test accuracy of several celebrated GNN architectures, including GCN, GAT, and GraphSage, drops rapidly when they are trained with fewer labeled data. Moreover, the variance of classification accuracy grows significantly as number of labeled nodes drops. Indeed, semi-supervised graph learning with very low-labeling rates has been studied in the Laplace learning and graph deep learning settings, see, e.g., [Lia+19; Cal+20; FS21]; one question is *can we develop new GNN architectures to improve the performance of graph deep learning in low-labeling rate regimes?*

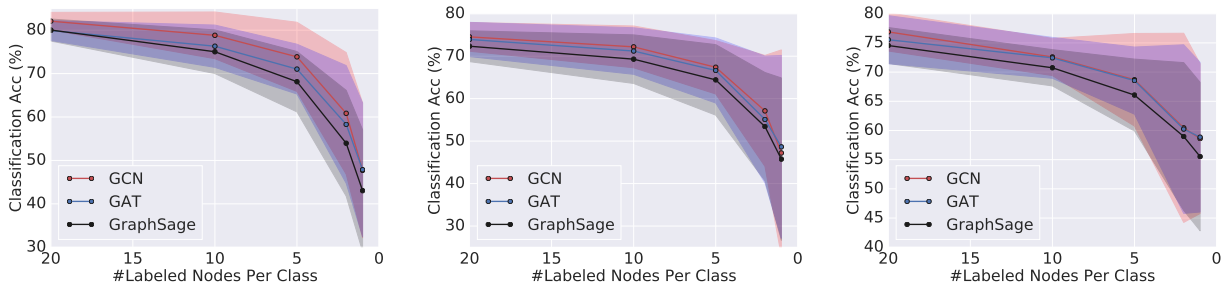


Figure 4.1: Test accuracy of GCN, GAT, and GraphSage vs. the number of labeled nodes per class. All networks have 2 layers, and each experiment is run with 100 splits and 20 random seeds following [Cha+21a]. The accuracy drops rapidly with fewer labeled data for training. CORA, CiteSeer, and PubMed have 2485, 2120, and 19717 nodes in total respectively. Results on more benchmark GNN architectures are in 4.6.2.1.

4.1.1 Our contribution

With the above GNN problems in mind, we focus on developing new continuous-depth GNNs to overcome over-smoothing and boost the accuracy of GNNs with a limited number of labeled data. We first present a random walk interpretation of the GRAND model [Cha+21a], revealing a potentially inevitable over-smoothing phenomenon when GRAND is implicitly very deep. Based on the random walk viewpoint of GRAND, we then propose graph neural diffusion with a source term (GRAND++) that corrects the bias arising from the diffusion process, see Sec. 4.5 for details. GRAND++ theoretically guarantees that: (i) under GRAND++ dynamics, the graph node features do not converge to a constant vector over all nodes even as the time goes to infinity, and (ii) GRAND++ can provide accurate prediction even when it is trained with a limited number of labeled nodes. Moreover, these theoretical results resonate with the practical advantages of GRAND++. We summarize the major practical advantages of GRAND++ below.

- GRAND++ can effectively overcome the over-smoothing issue; it is remarkably more accurate than existing GNNs when the architecture is very deep.
- GRAND++ is suitable for graph deep learning when only a few nodes are labeled as training data. Moreover, in the low-labeling rates, GRAND++ can be more accurate when the network is deeper.
- GRAND++ inherits the continuous-depth merit from GRAND, which defines the network depth implicitly and enables memory-efficient training by using the adjoint method.

4.1.2 Related work

Diffusion on graphs and continuous-depth graph neural networks. Diffusion has been defined on graphs, see, e.g., [FW93; FS00], and used in various applications, including data clustering and dimension reduction [Coi+05; BN03], image processing [GO08; ELB08; DEL13; LEL14], and semi-supervised graph nodes classification [ZGL03; Zho+04]. From the

numerical viewpoint, fast algorithms have been proposed for using diffusion on graphs to solve penalized graph cut problems [Gar+14]. The connection between GNNs and diffusion on graphs has been studied substantially. For instance, GNN has been interpreted as a diffusion process on graphs, which performs low-pass filtering on the input features [NM19]. Moreover, insights from the diffusion process on graphs have been used to improve the performance of GNNs, see, e.g., [AT16; Lia+19; KWG19; Wan+21b].

Leveraging neural ordinary differential equations (ODEs) [Che+18], continuous-depth GNNs have been proposed, see, e.g., [Pol+19; XQT20; Zhu+20b]. One recent work is GRAND [Cha+21a], which parameterizes the diffusion equation on graphs with a neural network. See Sec. 4.3 for a brief review of GRAND.

Neural ODEs. Neural ODEs [Che+18] are a class of continuous-depth neural networks whose depth is defined implicitly. Training neural ODEs using the adjoint method [Bit63] is more memory efficient than training other neural networks using backpropagation. GRANDs [Cha+21a] are a class of neural partial differential equations (PDEs) on graphs that can also be considered as a coupled system of neural ODEs. Furthermore, GRANDs are also trained by using the adjoint method.

Laplace learning and Poisson learning. Laplace learning has been used for semi-supervised data classification [ZGL03; Zho+04; Wan+06], image processing [BCM06; GO08], etc. Direct application of Laplace learning with Gaussian weights [BN04] or locally linear embedding weights [RS00] for the above tasks may cause inference inconsistency when only a limited number of graph nodes are labeled, resulting in poor performance. Several algorithms address the inference inconsistency at low labeling rate. They include up-weighting the weights of the labeled data [SOZ17] and the p-Laplacian [Cal18; RCL19; ZS05]. In [Cal+20], the authors have proposed Poisson learning for improving Laplace learning at extremely low-labeling rate regimes. Poisson learning augments Laplace learning with a Green's function

at each labeled data, enabling accurate node classification when only a few labeled data are available. Compared to Laplace learning, Poisson learning adds Green's function to the label of each labeled node and then performs label propagation to predict the label for unlabeled graph nodes. GRAND and GRAND++ both learn graph node representations and perform prediction by activating the node representations, which are fundamentally different from Laplace and Poisson learning.

4.1.3 Notation

We denote scalars by lower- or upper-case letters and vectors and matrices by lower- and upper-case boldface letters, respectively. For a matrix $\boldsymbol{A}$, we denote its transpose as $\boldsymbol{A}^\top$ and its Hadamard product with another matrix $\boldsymbol{B}$ as $\boldsymbol{A} \odot \boldsymbol{B}$, i.e., the entrywise multiplication of $\boldsymbol{A}$ and $\boldsymbol{B}$. We write the set $\{1, 2, \cdots, n\}$ as $[n]$. We denote the probability and expectation of a given random variable $\boldsymbol{x}$ as $\mathbb{P}(\boldsymbol{x})$ and $\mathbb{E}[\boldsymbol{x}]$, respectively. The meaning of other notations can be inferred from the context.

4.1.4 Organization

The chapter is organized as follows: In Sec. 4.2, we review diffusion equation on graphs and its connection to GNNs. In Secs. 4.3 and 4.4, we briefly review GRAND and present a random walk interpretation of GRAND, respectively. Leveraging the random walk viewpoint of GRAND, we propose GRAND++ for deep graph learning with theoretical guarantees in Sec. 4.5. We verify the efficacy of GRAND++ in Sec. 4.6.

4.2 Background

Background on Graph Differential Operators Let $(\boldsymbol{X}, \boldsymbol{W})$ represent a graph where $\boldsymbol{X} = ([\boldsymbol{x}^{(1)}]^\top, \ldots, [\boldsymbol{x}^{(n)}]^\top)^\top \in \mathbb{R}^{n \times d}$ is the matrix where each row $\boldsymbol{x}^{(i)} \in \mathbb{R}^d$ is a feature vector and $\boldsymbol{W} = (W_{ij})_{i,j=1}^n$ is a $n \times n$ matrix with W_{ij} representing the similarity (edge weight) between the ith and jth feature vector. We assume that we are dealing with an undirected graph, i.e., $W_{ij} = W_{ji}$. A $\mathbb{R}^{k_1}$-valued function on the nodes of the graph can be represented

as a matrix $\boldsymbol{U} \in \mathbb{R}^{n \times k_1}$ by $\boldsymbol{U} = ([\boldsymbol{u}^{(1)}]^\top, \ldots, [\boldsymbol{u}^{(n)}]^\top)^\top$ and we define the inner product

$$\langle \boldsymbol{U}, \boldsymbol{V} \rangle = \sum_{i=1}^{n} \boldsymbol{u}^{(i)} \cdot \boldsymbol{v}^{(i)}.$$

Similarly, a $\mathbb{R}^{k_2}$-valued function on the edges can be represented as a third-order tensor $\mathcal{U} \in \mathbb{R}^{n \times n \times k_2}$ which we write as

$$\mathcal{U} = \begin{pmatrix} \mathcal{U}^{(1,1)} & \cdots & \mathcal{U}^{(1,n)} \\ \vdots & \ddots & \vdots \\ \mathcal{U}^{(n,1)} & \cdots & \mathcal{U}^{(n,n)} \end{pmatrix}$$

and $\mathcal{U}^{(i,j)} \in \mathbb{R}^{k_2}$. On edge functions we use the inner product

$$\langle \mathcal{U}, \mathcal{V} \rangle = \frac{1}{2} \sum_{i,j=1}^{n} W_{ij} \mathcal{U}^{(i,j)} \cdot \mathcal{V}^{(i,j)}.$$

Multiplication between a matrix and an edge function is usually defined pointwise and we use the notation $[\boldsymbol{A} \odot \mathcal{U}]^{(i,j)} = A_{ij} \mathcal{U}^{(i,j)} \in \mathbb{R}^{k_2}$, for a matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ and an edge function $\mathcal{U} \in \mathbb{R}^{n \times n \times k_2}$, to make this clear. Similarly, pointwise multiplication between two matrices $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{n \times n}$ is defined by $[\boldsymbol{A} \odot \boldsymbol{B}]_{ij} = A_{ij} B_{ij} \in \mathbb{R}$. When a matrix is acting as a linear operator on a node function we use the usual matrix-vector notation and write $[\boldsymbol{A}\boldsymbol{U}]^{(i)} = \sum_{j=1}^{n} A_{ij} \boldsymbol{u}^{(j)} \in \mathbb{R}^{k_1}$ for a matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ and node function $\boldsymbol{U} = ([\boldsymbol{u}^{(1)}]^\top, \ldots, [\boldsymbol{u}^{(n)}]^\top)^\top \in \mathbb{R}^{n \times k_1}$. In the sequel we will have $k_1 = k_2 = d$.

The gradient of a node-function $\boldsymbol{U} = ([\boldsymbol{u}^{(1)}]^\top, \ldots, [\boldsymbol{u}^{(n)}]^\top)^\top \in \mathbb{R}^{n \times d}$ is defined as the edge-function $\nabla \boldsymbol{U} \in \mathbb{R}^{n \times n \times d}$ with $[\nabla \boldsymbol{U}]^{(i,j)} = \boldsymbol{u}^{(j)} - \boldsymbol{u}^{(i)} \in \mathbb{R}^d$. The divergence $\mathrm{div}\mathcal{V} = ([[\mathrm{div}\mathcal{V}]^{(1)}]^\top, \ldots, [[\mathrm{div}\mathcal{V}]^{(n)}]^\top)^\top \in \mathbb{R}^{n \times d}$ of an edge-function $\mathcal{V} \in \mathbb{R}^{n \times n \times d}$ is defined as

$$[\mathrm{div}\mathcal{V}]^{(i)} = \sum_{j=1}^{n} W_{ij} \mathcal{V}^{(i,j)}$$

for all $i = 1, \ldots, n$. For anti-symmetric edge functions, i.e. $\mathcal{V}^{(i,j)} = -\mathcal{V}^{(j,i)}$ for all i, j, we have that the divergence is the negative adjoint to the gradient, i.e.

$$\langle \mathrm{div}\mathcal{V}, \boldsymbol{U} \rangle = -\langle \mathcal{V}, \nabla \boldsymbol{U} \rangle.$$

Diffusion equation on graphs. Let $G = (\boldsymbol{X}, \boldsymbol{W})$ represent an undirected graph with n nodes, where $\boldsymbol{X} = \big([\boldsymbol{x}^{(1)}]^\top, \cdots, [\boldsymbol{x}^{(n)}]^\top\big)^\top \in \mathbb{R}^{n\times d}$ with each row $\boldsymbol{x}^{(i)} \in \mathbb{R}^d$ a feature vector and $\boldsymbol{W} := \big(W_{ij}\big)$ a $n \times n$ matrix with W_{ij} representing the similarity (edge weight) between the i^{th} and j^{th} feature vectors, and we assume $W_{ij} = W_{ji}$. Consider the following diffusion process that evolves the feature matrix $\boldsymbol{X}$ on the graph:

$$\frac{\partial \boldsymbol{X}(t)}{\partial t} = \mathrm{d}iv\big(\boldsymbol{G}(\boldsymbol{X}(t), t) \odot \nabla \boldsymbol{X}(t)\big), \tag{4.1}$$

where $\boldsymbol{X}(t) = \big([\boldsymbol{x}^{(1)}(t)]^\top, \cdots, [\boldsymbol{x}^{(n)}(t)]^\top\big)^\top \in \mathbb{R}^{n\times d}$ with $\boldsymbol{x}^{(i)}(0) = \boldsymbol{x}^{(i)}$, ∇ and $\mathrm{d}iv$ are the gradient and divergence operators, respectively. The matrix $\boldsymbol{G}(\boldsymbol{X}(t), t)$ is chosen such that $\boldsymbol{W} \odot \boldsymbol{G}$ is right-stochastic, i.e., each row of $\boldsymbol{W} \odot \boldsymbol{G}$ summing to 1. In the machine learning setting, we can parameterize $\boldsymbol{G}$ with learnable parameters θ which we denote by $\boldsymbol{G}(\boldsymbol{X}(t), t, \theta)$. The initial features are evolved under the diffusion dynamics (4.1) from $t = 0$ to T to learn the final representation $\boldsymbol{X}(T)$ for further machine learning tasks.

In the simplest case when $\boldsymbol{G}(\boldsymbol{X}(t), t)$ is only dependent on the initial node features $\boldsymbol{X}$, i.e., $\boldsymbol{G}$ is time-independent, right-stochasticity implies $\sum_j W_{ij}G_{ij} = 1$ for all i, and so we focus on the particular case when $G_{ij} = 1/d_i$ with $d_i = \sum_{j=1}^n W_{ij}$. In this case the right-hand side of (4.1) reduces to the negative of the random-walk Laplacian applied to $\boldsymbol{X}(t)$ and (4.1) becomes

$$\frac{\partial \boldsymbol{X}(t)}{\partial t} = \mathrm{d}iv\big(\boldsymbol{G}(\boldsymbol{X}(t), t) \odot \nabla \boldsymbol{X}(t)\big) = -\boldsymbol{L}\boldsymbol{X}(t), \tag{4.2}$$

where $\boldsymbol{L} = \boldsymbol{I} - \boldsymbol{D}^{-1}\boldsymbol{W} := \boldsymbol{I} - \boldsymbol{A}$ ($\boldsymbol{A} := \boldsymbol{A}(\boldsymbol{X})$) is the random walk Laplacian and $\boldsymbol{D}$ is diagonal with $D_{ii} = d_i$. See [CG97; FW93; FS00] for more about random walk Laplacian and diffusion on graphs.

Graph neural networks. Applying forward Euler discretization, with step size $\delta_t < 1$, of (4.2) gives

$$\boldsymbol{X}(k\delta_t) = \boldsymbol{X}((k-1)\delta_t) - \delta_t \boldsymbol{L}\boldsymbol{X}((k-1)\delta_t) := \tilde{\boldsymbol{L}}\boldsymbol{X}((k-1)\delta_t), \quad \text{for} \quad k = 1, 2, \cdots, K, \tag{4.3}$$

$T = K\delta_t$, and $\boldsymbol{X}(0) = \boldsymbol{X}$. Note that the matrix $\tilde{\boldsymbol{L}}$ is the discretization of the diffusion operator, which is a special low-pass filter. Equation (4.3) is a prototype for motivating GNNs: by introducing weights $\boldsymbol{W}^{(k)} \in \mathbb{R}^{d\times d}$ and a nonlinearity σ, e.g., ReLU, into (4.3), we have

$$\boldsymbol{X}((k+1)\delta_t) = \sigma\big(\tilde{\boldsymbol{L}}\boldsymbol{X}(k\delta_t)\boldsymbol{W}^{(k)}\big). \tag{4.4}$$

The model in (4.4) is similar to the well-established GCN architecture proposed in [KW17]. The diffusive nature of the GNN architecture in (4.4) further explains the over-smoothing issue of training deep GNNs; the deeper the network architecture is, the more the node features diffuse. Eventually, all nodes share similar features and become indistinguishable. See Sec. 4.5 for a detailed analysis.

4.3 A Brief Review of GRAND

GRAND is a new continuous-depth GNN proposed in [Cha+21a]. It integrates a learnable encoder function ϕ and a learnable decoder function ψ with the neural network parameterized graph diffusion process, resulting in the prediction $\boldsymbol{Y} = \psi(\boldsymbol{X}(T))$, where $\boldsymbol{X}(T)$ is computed as

$$\boldsymbol{X}(T) = \boldsymbol{X}(0) + \int_0^T \frac{\partial \boldsymbol{X}(t)}{\partial t} dt, \quad \text{with} \quad \boldsymbol{X}(0) = \phi(\boldsymbol{X}), \tag{4.5}$$

where $\partial \boldsymbol{X}(t)/\partial t$ is given by the graph diffusion equation (4.2). From the neural ODE perspective, we can perform forward propagation of GRAND, i.e., we solve (4.5), using numerical ODE solvers.

In the simplest case, when $\boldsymbol{G}$ is only dependent on the initial node features, we can rewrite (4.1) as

$$\frac{\partial \boldsymbol{X}(t)}{\partial t} = \big(\boldsymbol{A}(\boldsymbol{X}) - \boldsymbol{I}\big)\boldsymbol{X}(t), \tag{4.6}$$

GRAND models the diffusivity $\boldsymbol{A}(\boldsymbol{X})$ in (4.6) by the multi-head self-attention mechanism; potential choices of the attention function include the ones proposed in [Vas+17; Vel+]. More precisely, in GRAND $\boldsymbol{A}(\boldsymbol{X}) = \frac{1}{h}\sum_{l=1}^{h} \boldsymbol{A}^l(\boldsymbol{X})$ with h being the number of heads and the

attention matrix $\boldsymbol{A}^l(\boldsymbol{X}) = (a^l(\boldsymbol{x}_i, \boldsymbol{x}_j))$, for $l = 1, \cdots, h$, is computed as follows:

$$a^l(\boldsymbol{x}_i, \boldsymbol{x}_j) = \frac{\exp\big(\mathrm{L}eakyReLU(\boldsymbol{a}^{l\top}[\boldsymbol{W}^l\boldsymbol{x}_i \| \boldsymbol{W}^l\boldsymbol{x}_j])\big)}{\sum_{k\in\mathcal{N}_i}\exp\big(\mathrm{L}eakyReLU(\boldsymbol{a}^{l\top}[\boldsymbol{W}^l\boldsymbol{x}_i \| \boldsymbol{W}^l\boldsymbol{x}_k])\big)}, \tag{4.7}$$

where $\boldsymbol{W}^l$ and $\boldsymbol{a}^l$ are learned, $\|$ is the concatenation operator, and $\mathcal{N}_i$ is the index set of the nodes that are connected to the i^{th} node in the graph. GRAND with the attention in (4.7) is called GRAND-l, that is, GRAND-l is a special case of GRAND when the diffusivity is dependent only on the initial graph node features. Time-dependent attention and graph rewiring can be integrated into GRAND, resulting in GRAND-nl and GRAND-nl-rw, respectively [Cha+21a]. From the ODE viewpoint, GRAND and its variants are a class of coupled neural ODEs defined on an unweighted graph. Their merits include continuous-depth and memory-efficient training using the adjoint method [Bit63; Che+18].

4.4 Random walk viewpoint of GRAND

In this section, we present a random walk interpretation of GRAND. The connection between graph random walks and the diffusion equation has been extensively studied, but we recap the key idea here to motivate the new GRAND with a source term architecture. Let $\{\boldsymbol{B}^{(i)}(k)\}_{k\in\mathbb{N}}$ be the random walk on $\{\boldsymbol{x}^{(j)}(0)\}_{j=1}^n$ defined by, for $\delta_t \in [0,1]$,

$$\begin{aligned} \boldsymbol{B}^{(i)}(0) &= \boldsymbol{x}^{(i)}(0) \\ \mathbb{P}\big(\boldsymbol{B}^{(i)}(k+1) = \boldsymbol{x}^{(L)}(0) | \boldsymbol{B}^{(i)}(k) = \boldsymbol{x}^{(j)}(0)\big) &= \begin{cases} 1-\delta_t & \text{if } L = j \\ \frac{\delta_t W_{jL}}{d_j} & \text{if } L \neq j \end{cases} \end{aligned} \tag{4.8}$$

where $d_j = \sum_{L=1}^n W_{jL}$ (assume $W_{LL} = 0$ for all L). Proposition 5 below is well-known, see [ZGL03].

Proposition 5. *Let $\boldsymbol{X}$ solve* (4.3) *and $\boldsymbol{B}^{(i)}$ be the random walk determined by* (4.8) *where $\delta_t \in [0,1]$. Then*

$$\boldsymbol{x}^{(i)}(\delta_t k) = \mathbb{E}\big[\boldsymbol{B}^{(i)}(k)\big].$$

Proof of Proposition 5. For notational convenience let us assume that $\boldsymbol{x}^{(i)}(0) = \boldsymbol{x}^{(i)}$. Clearly

$$\mathbb{E}\left[\boldsymbol{B}^{(i)}(0)\right] = \boldsymbol{x}^{(i)} = \boldsymbol{x}^{(i)}(0)$$

for all $i = 1, \ldots, n$. Assume that

$$\mathbb{E}\left[\boldsymbol{B}^{(i)}(k)\right] = \boldsymbol{x}^{(i)}(\delta_t k)$$

for all $i = 1, \ldots, n$. Then,

$$\begin{aligned}
&\mathbb{E}\left[\boldsymbol{B}^{(i)}(k+1)\right] \\
&= \sum_{j=1}^{n} \boldsymbol{x}^{(j)} \mathbb{P}\left(\boldsymbol{B}^{(i)}(k+1) = \boldsymbol{x}^{(j)}\right) \\
&= \sum_{j=1}^{n}\sum_{L=1}^{n} \boldsymbol{x}^{(j)} \mathbb{P}\left(\boldsymbol{B}^{(L)}(k) = \boldsymbol{x}^{(j)} | \boldsymbol{B}^{(i)}(1) = \boldsymbol{x}^{(L)}\right) \mathbb{P}\left(\boldsymbol{B}^{(i)}(1) = \boldsymbol{x}^{(L)}\right) \\
&= \sum_{j=1}^{n}\sum_{L=1}^{n} \boldsymbol{x}^{(j)} \left((1-\delta_t)\,\mathbb{1}_{i=L} + \frac{\delta_t W_{iL}}{d_i}\right) \mathbb{P}\left(\boldsymbol{B}^{(i)}(1) = \boldsymbol{x}^{(L)}\right) \\
&= (1-\delta_t) \sum_{j=1}^{n} \boldsymbol{x}^{(j)} \mathbb{P}\left(\boldsymbol{B}^{(i)}(1) = \boldsymbol{x}^{(L)}\right) + \frac{\delta_t}{d_i} \sum_{L=1}^{n} W_{iL} \sum_{j=1}^{n} \boldsymbol{x}^{(j)} \mathbb{P}\left(\boldsymbol{B}^{(L)}(k) = \boldsymbol{x}^{(j)}\right) \\
&= (1-\delta_t)\mathbb{E}\left[\boldsymbol{B}^{(i)}(k)\right] + \frac{\delta_t}{d_i} \sum_{L=1}^{n} W_{iL} \mathbb{E}\left[\boldsymbol{B}^{(L)}(k)\right] \\
&= (1-\delta_t)\boldsymbol{x}^{(i)}(\delta_t k) + \frac{\delta_t}{d_i} \sum_{L=1}^{n} W_{iL} \boldsymbol{x}^{(L)}(\delta_t k) \\
&= \boldsymbol{x}^{(i)}(\delta_t k) + \frac{\delta_t}{d_i} \sum_{L=1}^{n} W_{iL} W_{iL} \left(\boldsymbol{x}^{(L)}(\delta_t k) - \boldsymbol{x}^{(i)}(\delta_t k)\right) \\
&= \boldsymbol{x}^{(i)}(\delta_t k) - \delta_t \left[\boldsymbol{L}\boldsymbol{X}(\delta_t k)\right]^{(i)} \\
&= \boldsymbol{x}^{(i)}(\delta_t (k+1)),
\end{aligned}$$

as required. □

Proposition 6 below gives the stationary distribution of the random walk $\{\boldsymbol{B}^{(i)}(k)\}_{k\in\mathbb{N}}$.

Proposition 6. *Assume the graph $G = (\boldsymbol{X}, \boldsymbol{W})$ is connected. Then, the stationary distribution of $\{\boldsymbol{B}^{(i)}(k)\}_{k\in\mathbb{N}}$ is*

$$\pi = \left(\frac{d_1}{\sum_{j=1}^{n} d_j}, \ldots, \frac{d_n}{\sum_{j=1}^{n} d_j}\right), \tag{4.9}$$

which is independent of the starting position $\boldsymbol{x}^{(i)}$.

Proof of Proposition 6. Let $\boldsymbol{P} = \big(P_{ij}\big) \in \mathbb{R}^{n\times n}$ be the probability transition kernel, so

$$P_{ij} = \begin{cases} 1-\delta_t & \text{if } i = j, \\ \frac{\delta_t W_{ij}}{d_i} & \text{if } i \neq j. \end{cases}$$

We have

$$\begin{aligned} \sum_{i=1}^{n} \pi_i P_{ij} &= \sum_{i=1}^{n} \frac{d_i}{\sum_{k=1}^{n} d_k} \left((1-\delta_t \mathbb{1}_{i=j} + \frac{\delta_t W_{ij}}{d_i} \right) \\ &= \frac{d_j(1-\delta_t)}{\sum_{k=1}^{n} d_k} + \frac{\delta_t \sum_{i=1}^{n} W_{ij}}{\sum_{k=1}^{n} d_k} \\ &= \frac{d_j}{\sum_{k=1}^{n} d_k} \\ &= \pi_j, \end{aligned}$$

as required. □

Furthermore, we have the following theoretical result on the asymptotic behavior of graph node features under the GRAND dynamics given by (4.3).

Proposition 7. *Assume the graph $G = (\boldsymbol{X}, \boldsymbol{W})$ is connected. Then for all $i = 1, \cdots, n$, we have*

$$\boldsymbol{x}^{(i)}(k\delta_t) \to \widetilde{\boldsymbol{x}} := \sum_{j=1}^{n} \boldsymbol{x}^{(j)}(0)\pi_j, \quad \textit{as} \quad k \to \infty.$$

Hence, for the case of (4.3), i.e., GRAND-l, we expect the output to be approximately independent of the input, due to over-smoothing. Of course, once we reintroduce the $\boldsymbol{X}(t)$ dependence back into $\boldsymbol{G}$ in (4.1) and (4.2) or into the operator $\boldsymbol{A}$ in (4.6) then the above arguments no longer hold. Nevertheless, the GRAND architectures are built on a principle that is ill-suited to deep networks. In the next section we introduce a source term and perform a similar random walk analysis that illustrates how the new architecture can be better suited for deep GNN architectures.

4.5 GRAND++: Graph Neural Diffusion with A Source Term

4.5.1 Algorithm and formulation

At the core of GRAND++ is the introduction of a source term into GRAND, leveraging the random walk viewpoint of the diffusion process. We take a small subset of feature vectors, indexed by $\mathcal{I} \subseteq [n]$, believed to be "trustworthy" for use as a source term. In particular, we use the features of labeled data. The GRAND++ dynamics are defined by a diffusion equation with a source term (we use the variable $\boldsymbol{z}$ for GRAND++-related dynamics and $\boldsymbol{x}$ for GRAND dynamics)

$$\frac{\partial \boldsymbol{z}^{(i)}(t)}{\partial t} = \operatorname{div}\left[\boldsymbol{G}(\boldsymbol{Z}(t), t) \odot \nabla \boldsymbol{Z}(t)\right]^{(i)} + \sum_{j \in \mathcal{I}} \delta_{ij} C_j \tag{4.10}$$

where C_j is the source at feature vector of node j. Below we motivate a particular choice of C_j.

The key idea is to first characterise the bias that arises from the diffusion and use that to propose a correction via the choice of source terms C_j. Following the simplifications in (4.2), our diffusion equation (without the source term) follows the approximate dynamics when $t \gg 1$

$$\frac{\partial \boldsymbol{x}^{(i)}(t)}{\partial t} = -\left[\boldsymbol{L}\boldsymbol{X}(t)\right]^{(i)} = -\underbrace{\boldsymbol{x}^{(i)}(t)}_{\approx \widetilde{\boldsymbol{x}}} + \frac{1}{d_i}\sum_{j=1}^{n} W_{ij} \underbrace{\boldsymbol{x}^{(j)}(t)}_{\approx \widetilde{\boldsymbol{x}}} \approx 0.$$

For $i \in \mathcal{I}$, it transpires that choosing $C_i = \boldsymbol{x}^{(i)} - \hat{\boldsymbol{x}}$ (where $\hat{\boldsymbol{x}}$ is defined below) gives rise to a random walk interpretation that allows us to prove that the oversmoothing seen in the GRAND model is avoided.

One can in fact choose $\widetilde{\boldsymbol{x}}$ with a certain degree of freedom. If we initialise $\boldsymbol{X}(0) = \boldsymbol{X}$ then we obtain $\widetilde{\boldsymbol{x}} = \sum_{j=1}^{n} \boldsymbol{x}^{(j)} \pi_j$ (as is usual in the GRAND model). However, as the similarities are encoded in the graph weights, and the diffusion dynamics will drive it towards a non-trivial state, we can choose a different initialization than $\boldsymbol{X}(0) = \boldsymbol{X}$. Through connections with

random walks we, in the next subsection, motivate an alternative initialisation

$$\sum_{i=1}^{n} \boldsymbol{z}^{(i)}(0) = \sum_{i \in \mathcal{I}} \frac{\boldsymbol{x}^{(i)} - \hat{\boldsymbol{x}}}{d_i}, \quad \text{where} \quad \hat{\boldsymbol{x}} = \frac{1}{|\mathcal{I}|} \sum_{j \in \mathcal{I}} \boldsymbol{x}^{(j)} \tag{4.11}$$

with the dynamics

$$\frac{\partial \boldsymbol{z}^{(i)}(t)}{\partial t} = \operatorname{div} \left[\boldsymbol{G}(\boldsymbol{Z}(t), t) \odot \nabla \boldsymbol{Z}(t) \right]^{(i)} + \sum_{j \in \mathcal{I}} \delta_{ij} \left(\boldsymbol{x}^{(i)} - \hat{\boldsymbol{x}} \right). \tag{4.12}$$

For example, we could choose

$$\boldsymbol{z}^{(i)}(0) = \begin{cases} \frac{1}{d_i} \left(\boldsymbol{x}^{(i)} - \hat{\boldsymbol{x}} \right) & \text{if } i \in \mathcal{I} \\ \mathbf{0} & \text{otherwise,} \end{cases} \quad \text{or} \quad \boldsymbol{z}^{(i)}(0) = \boldsymbol{x}^{(i)} - c,$$

where $c = \frac{1}{n} \Big(\sum_{i=1}^{n} \boldsymbol{x}^{(i)} - \sum_{j \in \mathcal{I}} \frac{\boldsymbol{x}^{(j)} - \hat{\boldsymbol{x}}}{d_j} \Big)$ is chosen such that (4.11) holds. We do not believe that the constant c (that shifts by a constant) is particularly important but it is included to provide a random walk interpretation which helps to understand the deep architecture (when T is big) behaviour of the model GRAND++. The justification for this choice will be made in Sec. 4.5.2. To summarize, the GRAND++ model in (4.12), with initial condition satisfying (4.11), simply adds a source term to the original GRAND model and uses a different initial condition. Therefore, the nonlinear diffusivity and graph rewiring tricks used by GRAND can be easily integrated into GRAND++. In terms of implementation, since GRAND++ merely changes the right-hand side of GRAND, which again can be regarded as a system of coupled first-order neural ODEs; we can leverage neural ODE training, testing, and inference for GRAND++ similar to GRAND.

In the next subsection we explore the random walk connection of the above model, suggesting that building a graph neural network based on the diffusion with source model does not suffer from the same degeneracy as we observed in Sec. 4.4 and is therefore better suited to build deep GNNs. In particular, we can write the diffusion with source model as the short time expected behaviour of a random walk and therefore we do not have the issue of reaching the stationary state (in other words passing the mixing time). Our experiments in Sec. 4.6 suggest the formal motivation holds and we are able to design deep GNNs.

4.5.2 The random walk perspective of GRAND++

Let us continue to consider the simplified model in the previous subsection, i.e., assume the dynamics are governed by

$$\frac{\partial \boldsymbol{z}^{(i)}(t)}{\partial t} = -\left[\boldsymbol{L}\boldsymbol{Z}(t)\right]^{(i)} + \sum_{j\in\mathcal{I}} \delta_{ij}\left(\boldsymbol{x}^{(i)} - \hat{\boldsymbol{x}}\right) \tag{4.13}$$

where the initial condition satisfies (4.11). Using the forward Euler discretisation of the above dynamics we have

$$\boldsymbol{z}^{(i)}(\delta_t k) = \boldsymbol{z}^{(i)}(\delta_t(k-1)) - \delta_t\left[\boldsymbol{L}\boldsymbol{Z}(\delta_t(k-1))\right]^{(i)} + \delta_t \sum_{j\in\mathcal{I}} \delta_{ij}\left(\boldsymbol{x}^{(i)} - \hat{\boldsymbol{x}}\right), \tag{4.14}$$

for $k = 1, 2, \ldots, K$ where again $T = K\delta_t$.

We use the same random walk as that introduced in Sec. 4.4, i.e. the random walk defined by (4.8), but we will now only consider random walks that are initialised on the nodes indexed by $\mathcal{I}$.

Proposition 8. *Let $\boldsymbol{Z}$ solve* (4.14) *with the initial condition satisfying* (4.11)*, and let $\boldsymbol{B}^{(i)}$ be the random walk determined by* (4.8)*. Then,*

$$\left|\boldsymbol{z}^{(i)}(k\delta_t) - \mathbb{E}\left[\sum_{s=0}^{k}\frac{1}{d_i}\sum_{j\in\mathcal{I}}\left(\boldsymbol{x}^{(j)} - \hat{\boldsymbol{x}}\right)\mathbb{1}_{\boldsymbol{B}^{(j)}(s)=\boldsymbol{x}^{(i)}}\right]\right| \to 0 \quad \text{as } k\to\infty.$$

Proof of Proposition 8. Let

$$\boldsymbol{y}^{(i)}(k) = \mathbb{E}\left[\sum_{s=0}^{k}\frac{1}{d_i}\sum_{j\in\mathcal{I}}\left(\boldsymbol{x}^{(j)} - \hat{\boldsymbol{x}}\right)\mathbb{1}_{\boldsymbol{B}^{(j)}(s)=\boldsymbol{x}^{(i)}}\right].$$

Notice that

$$
\begin{aligned}
&\mathbb{E}\left[\sum_{s=0}^{k}\mathbb{1}_{\boldsymbol{B}^{(j)}(s)=\boldsymbol{x}^{(i)}}\right]\\
&=\sum_{s=0}^{k}\mathbb{P}\left(\boldsymbol{B}^{(j)}(s)=\boldsymbol{x}^{(i)}\right)\\
&=\underbrace{\mathbb{P}\left(\boldsymbol{B}^{(j)}(0)=\boldsymbol{x}^{(i)}\right)}_{\delta_{ij}}+\sum_{s=1}^{k}\mathbb{P}\left(\boldsymbol{B}^{(j)}(s)=\boldsymbol{x}^{(i)}\right)\\
&=\delta_{ij}+\sum_{s=1}^{k}\sum_{L=1}^{n}\mathbb{P}\left(\boldsymbol{B}^{(j)}(s)=\boldsymbol{x}^{(i)}|\boldsymbol{B}^{(j)}(s-1)=\boldsymbol{x}^{(L)}\right)\mathbb{P}\left(\boldsymbol{B}^{(j)}(s-1)=\boldsymbol{x}^{(L)}\right)\\
&=\delta_{ij}+\sum_{s=1}^{k}\sum_{L=1}^{n}\left((1-\delta_t)\delta_{Li}+\frac{\delta_t W_{Li}}{d_L}\right)\mathbb{P}\left(\boldsymbol{B}^{(j)}(s-1)=\boldsymbol{x}^{(L)}\right)\\
&=\delta_{ij}+(1-\delta_t)\sum_{s=1}^{k}\mathbb{P}\left(\boldsymbol{B}^{(j)}(s-1)=\boldsymbol{x}^{(i)}\right)+\delta_t\sum_{L=1}^{n}\frac{W_{Li}}{d_L}\sum_{s=1}^{k}\mathbb{P}\left(\boldsymbol{B}^{(j)}(s-1)=\boldsymbol{x}^{(L)}\right)\\
&=\delta_{ij}+(1-\delta_t)\sum_{s=0}^{k-1}\mathbb{P}\left(\boldsymbol{B}^{(j)}(s)=\boldsymbol{x}^{(i)}\right)+\delta_t\sum_{L=1}^{n}\frac{W_{Li}}{d_L}\sum_{s=0}^{k-1}\mathbb{P}\left(\boldsymbol{B}^{(j)}(s)=\boldsymbol{x}^{(L)}\right)\\
&=\delta_{ij}+(1-\delta_t)\mathbb{E}\left[\sum_{s=0}^{k-1}\mathbb{1}_{\boldsymbol{B}^{(j)}(s)=\boldsymbol{x}^{(i)}}\right]+\delta_t\sum_{L=1}^{n}\frac{W_{Li}}{d_L}\mathbb{E}\left[\sum_{s=0}^{k-1}\mathbb{1}_{\boldsymbol{B}^{(j)}(s)=\boldsymbol{x}^{(L)}}\right].
\end{aligned}
$$

From the definition of $\boldsymbol{Y}$ and the above recursive relationship we have

$$
\begin{aligned}
\boldsymbol{y}^{(i)}(k)&=\frac{1}{d_i}\sum_{j\in\mathcal{I}}\left(\boldsymbol{x}^{(j)}-\hat{\boldsymbol{x}}\right)\mathbb{E}\left[\sum_{s=0}^{k}\mathbb{1}_{\boldsymbol{B}^{(j)}(s)=\boldsymbol{x}^{(i)}}\right]\\
&=\frac{1}{d_i}\sum_{j\in\mathcal{I}}\left(\boldsymbol{x}^{(j)}-\hat{\boldsymbol{x}}\right)\delta_{ij}+(1-\delta_t)\frac{1}{d_i}\sum_{j\in\mathcal{I}}\left(\boldsymbol{x}^{(j)}-\hat{\boldsymbol{x}}\right)\mathbb{E}\left[\sum_{s=0}^{k-1}\mathbb{1}_{\boldsymbol{B}^{(j)}(s)=\boldsymbol{x}^{(L)}}\right]\\
&\qquad+\frac{\delta_t}{d_i}\sum_{L=1}^{n}\frac{W_{Li}}{d_L}\sum_{j\in\mathcal{I}}\left(\boldsymbol{x}^{(j)}-\hat{\boldsymbol{x}}\right)\mathbb{E}\left[\sum_{s=0}^{k-1}\mathbb{1}_{\boldsymbol{B}^{(j)}(s)=\boldsymbol{x}^{(L)}}\right]\\
&=\frac{1}{d_i}\sum_{j\in\mathcal{I}}\left(\boldsymbol{x}^{(j)}-\hat{\boldsymbol{x}}\right)\delta_{ij}+(1-\delta_t)\boldsymbol{y}^{(i)}(k-1)+\frac{\delta_t}{d_i}\sum_{L=1}^{n}W_{Li}\boldsymbol{y}^{(L)}(k-1)\\
&=\boldsymbol{y}^{(i)}(k-1)+\frac{1}{d_i}\sum_{j\in\mathcal{I}}\left(\boldsymbol{x}^{(j)}-\hat{\boldsymbol{x}}\right)\delta_{ij}-\delta_t\left[\boldsymbol{L}\boldsymbol{Y}(k-1)\right]^{(i)}.
\end{aligned}
$$

Now we let

$$
\boldsymbol{w}^{(i)}(k)=d_i\left(\boldsymbol{z}^{(i)}(k\delta_t)-\boldsymbol{y}^{(i)}(k)\right)
$$

so that $\boldsymbol{W}$ satisfies

$$\boldsymbol{w}^{(i)}(k) = \boldsymbol{w}^{(i)}(k-1) - \delta_t \left[\boldsymbol{L}\boldsymbol{W}(k-1)\right]^{(i)} = \left[\boldsymbol{P}\boldsymbol{W}(k-1)\right]^{(i)}.$$

Hence, $\boldsymbol{W}(k) = \boldsymbol{P}^k \boldsymbol{W}(0)$. Since the stationary distribution of the random walk with transition kernel is π, we have $\lim_{k\to\infty} \boldsymbol{P}^k = \mathbb{1}\pi^\top$. Hence, as $k \to \infty$, $\boldsymbol{w}^{(i)}(k) \to \sum_{j=1}^n \pi_j \boldsymbol{w}^{(j)}(0) = 0$ since $\boldsymbol{w}^{(j)}(0) = 0$ by the choice in the initialisation of $\boldsymbol{Z}$. □

Remark 3. *In the limit $k \to \infty$ the term*

$$\mathbb{E}\Big[\sum_{s=0}^{k} \frac{1}{d_i} \sum_{j\in\mathcal{I}} \boldsymbol{x}^{(j)} \mathbb{1}_{\boldsymbol{B}^{(j)}(s)=\boldsymbol{x}^{(i)}}\Big]$$

is formally a function of the random walk at all times. Whilst if k is very large (i.e. in comparison to the mixing time) we still have that

$$\mathbb{E}\Big[\frac{1}{d_i} \sum_{j\in\mathcal{I}} \boldsymbol{x}^{(j)} \mathbb{1}_{\boldsymbol{B}^{(j)}(k)=\boldsymbol{x}^{(i)}}\Big] = \frac{1}{d_i} \sum_{j\in\mathcal{I}} \boldsymbol{x}^{(j)} \underbrace{\mathbb{P}\left(\boldsymbol{B}^{(j)}(k) = \boldsymbol{x}^{(j)}\right)}_{\approx \pi_i} \approx \frac{\pi_i}{d_i} \sum_{j\in\mathcal{I}} \boldsymbol{x}^{(j)} \tag{4.15}$$

and on the other hand

$$\mathbb{E}\Big[\frac{1}{d_i} \sum_{j\in\mathcal{I}} \hat{\boldsymbol{x}} \mathbb{1}_{\boldsymbol{B}^{(j)}(k)=\boldsymbol{x}^{(i)}}\Big] = \frac{\hat{\boldsymbol{x}}}{d_i} \sum_{j\in\mathcal{I}} \underbrace{\mathbb{P}\left(\boldsymbol{B}^{(j)}(k) = \boldsymbol{x}^{(j)}\right)}_{\approx \pi_i} \approx \frac{\pi_i |\mathcal{I}| \hat{\boldsymbol{x}}}{d_i}. \tag{4.16}$$

From the definition of $\hat{\boldsymbol{x}}$ we see that (4.15) *and* (4.16) *are approximately equal. And therefore we can understand $\mathbb{E}[\frac{1}{d_i}\sum_{j\in\mathcal{I}} \hat{\boldsymbol{x}} \mathbb{1}_{\boldsymbol{B}^{(j)}(k)=\boldsymbol{x}^{(i)}}]$ as the long time behaviour of $\mathbb{E}[\frac{1}{d_i}\sum_{j\in\mathcal{I}} \boldsymbol{x}^{(j)} \mathbb{1}_{\boldsymbol{B}^{(j)}(k)=\boldsymbol{x}^{(i)}}]$ Very formally we can see that subtracting the long-time behaviour from the all-time behaviour leaves us with the short time behaviour. This provides one explanation as to why we do not expect the deep layers to be determined by the stationary state of the random walk (at which point there is little dependence on the initial layers, causing the deep layers to be approximately constant).*

Remark 4. *The random walk interpretation*

$$\mathbb{E}\Big[\sum_{s=0}^{k} \frac{1}{d_i} \sum_{j\in\mathcal{I}} \left(\boldsymbol{x}^{(j)} - \hat{\boldsymbol{x}}\right) \mathbb{1}_{\boldsymbol{B}^{(j)}=\boldsymbol{x}^{(i)}}\Big] \tag{4.17}$$

can be considered to be dual to the random walk interpretation in Sec. 4.4: in Sec. 4.4 we released the random walker from the node of interest, whilst now we release the random walkers from nodes indexed by $\mathcal{I}$ and see how many of them hit the node of interest. We note also that we do not require a lower bound on the size of the set $\mathcal{I}$. Indeed, if $|\mathcal{I}|$ is fixed whilst one takes the number of feature vectors $n \to \infty$ we still expect many properties of GRAND++, in particular Proposition 9 below, to hold. This is due to the asymptotic well-posedness of the dual random walk in low labeling rates [Cal+20].

Proposition 7 reveals that in the simple setting of (4.2), GRAND converges to a constant when its depth goes to infinity. However, this is not true for GRAND++ since the graph node features will not converge to a constant vector driven by the GRAND++, as shown in Proposition 9 below.

Proposition 9. *Assume the graph $G = (\boldsymbol{X}, \boldsymbol{W})$ is connected. Then $\boldsymbol{z}^{(i)}(k\delta_t)$ that was defined in (4.14) does not converge to a constant vector as a function of i as $k \to \infty$. That is, the node features will not become the same across graph nodes under the GRAND++ dynamics.*

Proof of Proposition 9. We prove the result by contradiction. Assume that there exists $\overline{\boldsymbol{z}}$ such that $\boldsymbol{z}^{(i)}(k\delta_t) \to \overline{\boldsymbol{z}}$ for all $i = 1, \ldots, n$ as $k \to \infty$. Then $\boldsymbol{L}\boldsymbol{Z}(k\delta_t) \to 0$. Since we can write

$$\begin{aligned}\boldsymbol{z}^{(i)}(k\delta_t) - \boldsymbol{z}^{(j)}(k\delta_t) = \boldsymbol{z}^{(i)}((k-1)\delta_t) - \boldsymbol{z}^{(j)}((k-1)\delta_t) \\ - \delta_t \Big([\boldsymbol{L}\boldsymbol{Z}((k-1)\delta_t)]^{(i)} - [\boldsymbol{L}\boldsymbol{Z}((k-1)\delta_t)]^{(j)} \Big) \\ + \delta_t \sum_{L \in \mathcal{I}} \Big(\delta_{iL}(\boldsymbol{x}^{(i)} - \hat{\boldsymbol{x}}) - \delta_{jL}(\boldsymbol{x}^{(j)} - \hat{\boldsymbol{x}}) \Big),\end{aligned}$$

then taking the limit $k \to \infty$ implies

$$\sum_{L \in \mathcal{I}} \delta_{iL}(\boldsymbol{x}^{(i)} - \hat{\boldsymbol{x}}) = \sum_{L \in \mathcal{I}} \delta_{jL}(\boldsymbol{x}^{(j)} - \hat{\boldsymbol{x}})$$

for all i, j which is clearly not true. □

Remark 5. *Proposition 9 guarantees GRAND++ is less likely to suffer from over-smoothing than GRAND, and in particular it shows that we have a non-constant deep layer limit, i.e., as $t \to \infty$. Analysing the limit is beyond the scope of the chapter but we have seen one characterisation in Proposition 8. By construction we have $\partial \boldsymbol{z}^{(i)}(t)/\partial t \approx \mathbf{0}$ for $i \in \mathcal{I}$ so one should expect that the deep layer limit is (close to) a smooth interpolation of the feature vectors labeled by $\mathcal{I}$.*

The continuous time model (4.10) is, in the special case of (4.13), the mean field limit of the probabilistic formulation (4.17). Our proposed algorithm is formulated from the mean-field limit.

4.6 Experiments

In this section, we compare the performance of GRAND++ with GRAND and several other popular GNNs on various graph node classification tasks. We aim to show the practical advantages of GRAND++ in learning with limited labeled data and using deep architectures. Without mentioning clearly, we use the same hyperparameters that that used for GRAND in [Cha+21a] for GRAND++. We provide detailed descriptions of experimental settings and datasets that are omitted in the main text in 4.6.4. For all experiments, we run 100 splits for each dataset with 20 random seeds for each split, which are conducted on a server with four NVIDIA RTX 3090 graphics cards.

We compare the performance of GRAND++ and its nonlinear and graph rewiring variants with several popular GNNs on various graph node classification benchmarks. Except for the integration time, which measures the implicit depth of GRAND and GRAND++, we adopt the experimental settings of GRAND in [Cha+21a] for GRAND++ include numerical differential equation solvers. Following [Cha+21a], we study seven graph node classification datasets, namely CORA, CiteSeer, PubMed, CoauthorCS, Computer, Photo, and ogbn-arxiv; we describe these datasets in 4.6.4.

4.6.1 GRAND++ is more resilient to deep architectures

We first show that our introduced source term in (4.12) can improve the accuracy of GRAND-l when the architecture is deep, i.e., the integration time T in (4.5) is big. We denote GRAND-l with the source term as GRAND++-l. For each node classification task, we train all models using the same number of labeled nodes as in [Cha+21a]. Figure 4.2 contrasts the performance of GRAND-l and GRAND++-l with different depths, or T, on CORA, CiteSeer, Computer, and Photo datasets. We provide the detailed results on PubMed and CoauthorCS, together with more comparisons of GRAND++-l with GRAND-l and several other celebrated GNNs include GCN, GAT, and GraphSage in Table 4.1. The results in Fig. 4.2 and Table 4.1 confirm that GRAND-l suffers less from over-smoothing compared to GCN, GAT, and GraphSage. Moreover, GRAND++-l performs on par with GRAND-l when the depth (T) of the network is small, but GRAND++-l significantly outperforms GRAND-l when T is large. As T increases, the margin becomes wider, indicating that GRAND++-l can overcome over-smoothing much more effectively than GRAND-l. Note that we did not use uniform depth for GRAND-l and GRAND++-l on all datasets because the adaptive step-size ODE solver fails when T is large for some tasks.

Open graph benchmark with paper citation network (ogbn-arxiv). Ogbn-arxiv consists of 169, 343 nodes and 1, 166, 243 directed edges. Each node is an arxiv paper represented by a 128-dimensional features and each directed edge indicates the citation direction. This dataset is used for node property prediction and has been a popular benchmark to test the advantage of deep graph neural networks over shallow graph neural networks [Li+20a; Li+21]. We train two models using labeling rates of 3.0% and 5.0%, respectively; the corresponding test accuracy for GRAND-l/GRAND++-l are 65.26%/66.64% and 67.42%/67.77%, respectively. GRAND++-l outperforms GRAND-l in both labeling rates.We further compare GRAND and GRAND++ with different depth on the ogbn-arxiv task. Compared to the GRAND model used in [Cha+21a], we reduce the hidden dimension from 162 to 81 to fit the

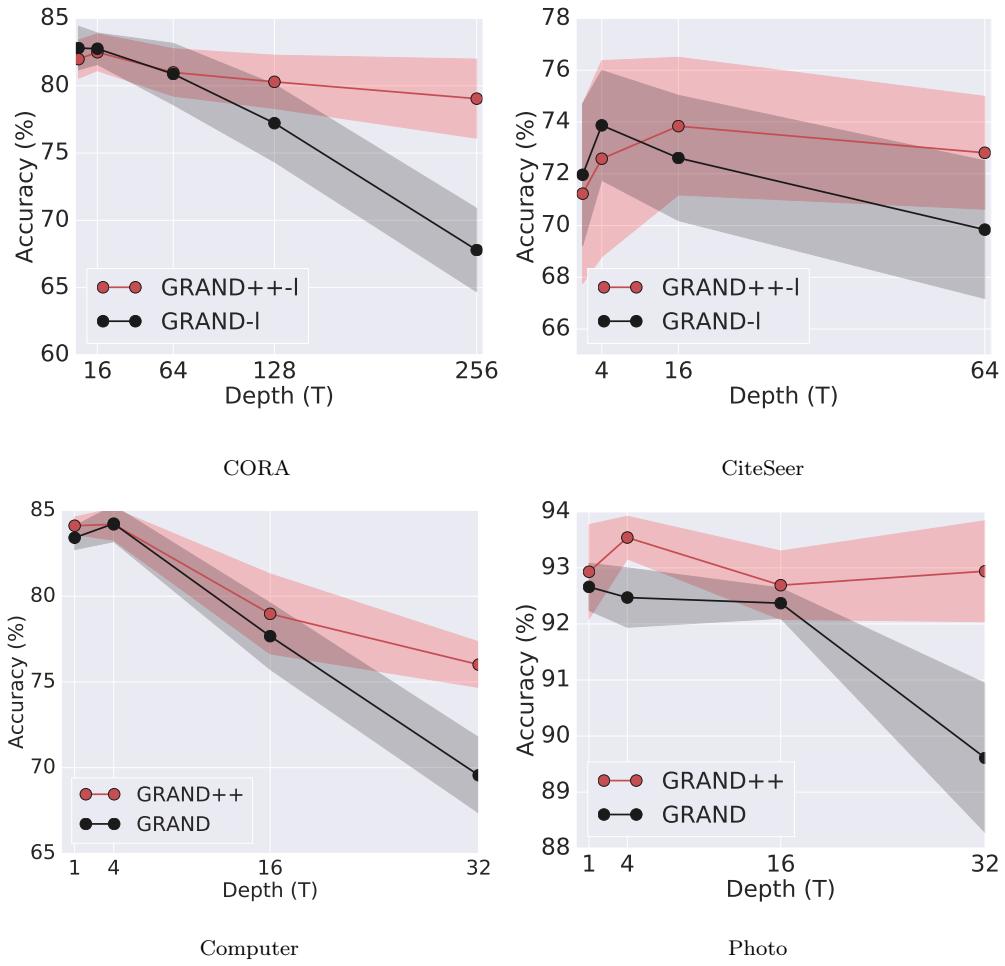


Figure 4.2: Test accuracy vs. the "depth" (T in (4.5)) of GRAND-l and GRAND++-l on the four graph node classification tasks. We see that GRAND++-l is much more resilient to deep architectures than GRAND-l. These results show that GRAND++ is better suited for learning with a very deep architecture than GRAND.

model into the GPU in our lab. We can clearly see that GRAND++ outperforms GRAND on various choices of depth in table 4.2.

4.6.2 GRAND++ is more accurate with limited labeled training data

Besides helping to overcome over-smoothing, our theory shows that the source term can boost the accuracy of GRAND-l with low-labeling rates. Table 4.3 compares the accuracy of

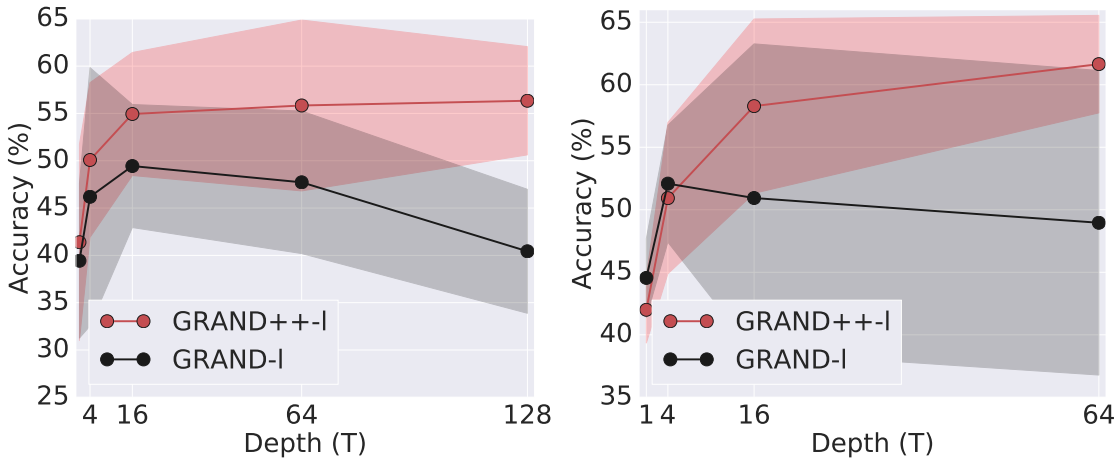


Figure 4.3: Accuracy of GRAND++-l and GRAND-l for CORA and CiteSeer, where both models, with different depth (T), are train with 1 labeled node per class. These results show that GRAND++ is more effective in learning with low-labeling rates than GRAND.

GRAND++-l with GRAND-l, GCN, GAT, GraphSage, and MoNet, trained with different numbers of labeled data. Here, we slightly tune T for GRAND++ based on the optimal value for GRAND, see Table 4.12 for their values. We see that with few labeled data, in most tasks GRAND++-l is significantly more accurate than the other GNNs include GRAND-l, confirming our theoretical insight. For CoauthorCS task, both GRAND-l and GRAND++-l are worse than GCN and GraphSage. Moreover, increasing the depth of GRAND++-l can improve the classification accuracy with limited training data, but this is not the case for GRAND-l, see Fig. 4.3. We perform the t-test in to confirm the statistical significance of the accuracy gain of GRAND++ over GRAND in Table 4.3.

t-test of the accuracy improvement of GRAND++ over GRAND To confirm the statistical significance of the accuracy improvement of GRAND++ over GRAND in Table 4.3, in this subsection, we conduct t-test experiments at 0.95 confidence to compare GRAND and GRAND++ on six different benchmark graph node classification tasks. We

Model	depth	CORA	CiteSeer	PubMed	CoauthorCS	Computer	Photo
GRAND++-l (ours)	1	77.48 ± 1.43	71.23 ± 3.47	**78.11 ± 1.47**	90.42 ± 0.76	**84.11 ± 0.51**	**92.93 ± 0.84**
	4	81.98 ± 1.42	72.58 ± 3.79	**79.20 ± 0.74**	90.89 ± 0.36	84.19 ± 0.93	**93.54 ± 0.38**
	16	82.49 ± 1.37	**73.84 ± 2.66**	**79.49 ± 0.84**	**90.24 ± 0.30**	**78.97 ± 2.33**	**92.69 ± 0.61**
	32	**82.48 ± 0.71**	**73.29 ± 1.29**	**79.81 ± 1.61**	NA	**76.01 ± 1.33**	**92.94 ± 0.90**
	64	**80.99 ± 1.76**	**72.81 ± 2.18**	NA	NA	NA	NA
	128	**80.29 ± 1.98**	NA	NA	NA	NA	NA
	256	**79.04 ± 2.94**	NA	NA	NA	NA	NA
GRAND-l [Cha+21a]	1	**78.59 ± 1.17**	71.96 ± 2.74	77.93 ± 1.26	90.79 ± 0.93	83.41 ± 0.69	92.66 ± 0.42
	4	**82.80 ± 1.62**	**73.87 ± 2.12**	78.71 ± 1.19	**90.94 ± 0.21**	**84.23 ± 1.05**	92.47 ± 0.53
	16	**82.75 ± 1.17**	72.61 ± 2.42	78.79 ± 0.93	87.66 ± 1.70	77.67 ± 1.94	92.37 ± 0.27
	32	82.19 ± 1.73	72.65 ± 3.15	78.70 ± 1.08	NA	69.56 ± 2.20	89.61 ± 1.33
	64	80.87 ± 2.28	69.84 ± 2.66	NA	NA	NA	NA
	128	77.22 ± 2.88	NA	NA	NA	NA	NA
	256	67.79 ± 3.10	NA	NA	NA	NA	NA
GCN [KW17]	1	76.92 ± 0.56	**72.80 ± 1.69**	72.78 ± 1.80	91.53 ± 0.45	81.44 ± 0.24	91.31 ± 0.19
	4	81.35 ± 1.27	70.54 ± 6.61	77.15 ± 3.00	87.84 ± 0.96	75.73 ± 1.02	90.11 ± 0.66
	16	19.70 ± 7.06	24.78 ± 1.45	41.36 ± 1.77	14.49 ± 0.91	12.86 ± 2.39	23.11 ± 1.76
	32	21.86 ± 6.09	24.23 ± 1.65	40.66 ± 1.86	12.14 ± 1.64	21.15 ± 13.10	24.30 ± 0.73
GAT [Vel+]	1	72.49 ± 2.03	71.83 ± 1.53	77.24 ± 0.72	79.22 ± 0.60	73.97 ± 1.20	87.08 ± 0.37
	4	80.95 ± 2.28	72.31 ± 2.82	77.37 ± 1.32	78.05 ± 1.10	76.67 ± 2.79	87.95 ± 1.76
	16	29.14 ± 1.02	24.84 ± 1.45	39.21 ± 0.43	24.20 ± 2.22	37.07 ± 2.99	29.97 ± 3.68
	32	29.75 ± 1.57	24.83 ± 1.45	39.02 ± 0.12	**22.73 ± 2.08**	32.53 ± 3.09	25.57 ± 4.03
GraphSage [HYL17]	1	73.47 ± 1.98	71.94 ± 1.45	72.42 ± 0.61	**91.74 ± 0.26**	75.95 ± 0.70	88.10 ± 0.87
	4	79.83 ± 2.43	50.00 ± 14.27	76.01 ± 2.35	87.94 ± 0.23	75.62 ± 2.85	90.68 ± 2.11
	16	25.52 ± 6.45	24.84 ± 1.45	37.55 ± 3.92	10.12 ± 2.21	22.79 ± 10.77	25.57 ± 3.31
	32	29.14 ± 1.02	28.38 ± 2.54	39.21 ± 4.39	7.91 ± 3.15	37.07 ± 13.22	20.09 ± 5.67

Table 4.1: Classification accuracy of different GNN models with different depths on six benchmark graph node classification tasks. NA: neural ODE solver failed. These results show that GRAND++ is better suited for learning with a very deep architecture than GRAND. (Unit: %)

first perform unpaired t-tests to show the improvement of GRAND++ over GRAND on low

Model	depth (T)	GRAND-l [Cha+21a]	GRAND++-l (ours)	Improvement from GRAND++-l (ours)
OGBN-arXiv	1	68.50 ± 0.76	**68.79 ± 0.35**	0.29
	4	69.53 ± 0.21	**69.68 ± 0.38**	0.15
	6	69.46 ± 0.43	**69.71 ± 0.24**	0.25
	8	69.44 ± 0.30	**69.61 ± 0.28**	0.17
	32	67.44 ± 0.59	**69.41 ± 0.53**	1.97
	64	63.47 ± 0.28	**68.05 ± 0.73**	4.58
	96	55.95 ± 1.24	**67.26 ± 0.61**	11.31

Table 4.2: Classification accuracy of the linear GRAND and GRAND++ models trained with different depth on the OGBN-arXiv graph node classification task. Compared to the GRAND model used in [Cha+21a], we reduce the hidden dimension from 162 to 81 to fit the model into the GPU in our lab. (Unit: %)

labeled datasets using the following t-score

$$\textbf{t-score} = \frac{\mu_{\textbf{GRAND++}} - \mu_{\textbf{GRAND}}}{\sqrt{\frac{\sigma^2_{\textbf{GRAND++}}}{n} + \frac{\sigma^2_{\textbf{GRAND++}}}{n}}}, \tag{4.18}$$

where μ and σ^2 are the mean and variance of the performances of each model, and n is the number of runs for each model. The t-test score are shown in Table 4.4.

For some entries in Table 4.4 that are not significant enough, we further conduct paired t-test between GRAND++ and GRAND on these specific datasets as shown in Table 4.5. Since a large portion of variance comes from splitting of the datasets, we pair up tests of GRAND and GRAND++ with the same splitting in this experiment. In this case, a sample of difference of size n is computed, and t-test score can be computed using the equation

$$\textbf{t-score} = \frac{\mu_{\textbf{diff}} - 0}{\sigma_{\textbf{diff}}/\sqrt{n}}. \tag{4.19}$$

4.6.2.1 Classification accuracy of GNNs with fewer number of training data

Besides the results shown in Fig. 4.1, we further test the classification accuracy of more benchmark GNN architectures trained with fewer numbers of labeled data per class.

Model	Label#	CORA	CiteSeer	PubMed	CoauthorCS	Computer	Photo
GRAND++ -l (ours)	1	**54.94 ± 16.09**	**58.95 ± 9.59**	**65.94 ± 4.87**	60.30 ± 1.50	**67.65 ± 0.37**	**83.12 ± 0.78**
	2	**66.92 ± 10.04**	**64.98 ± 8.31**	**69.31 ± 4.87**	76.53 ± 1.85	76.47 ± 1.48	**83.71 ± 0.90**
	5	**77.80 ± 4.46**	**70.03 ± 3.63**	71.99 ± 1.91	84.83 ± 0.84	**82.64 ± 0.56**	88.33 ± 1.21
	10	**80.86 ± 2.99**	**72.34 ± 2.42**	75.13 ± 3.88	86.94 ± 0.46	**82.99 ± 0.81**	90.65 ± 1.19
	20	**82.95 ± 1.37**	73.53 ± 3.31	**79.16 ± 1.37**	90.80 ± 0.34	**85.73 ± 0.50**	**93.55 ± 0.38**
GRAND-l [Cha+21a]	1	52.53 ± 16.40	50.06 ± 17.98	62.11 ± 10.58	59.15 ± 5.73	48.67 ± 1.66	81.25 ± 2.50
	2	64.82 ± 11.16	59.55 ± 10.89	69.00 ± 7.55	73.83 ± 5.58	74.77 ± 1.85	82.13 ± 3.27
	5	76.07 ± 5.08	68.37 ± 5.00	**73.98 ± 5.08**	85.29 ± 2.19	80.72 ± 1.09	88.27 ± 1.94
	10	80.25 ± 3.40	71.90 ± 7.66	**76.33 ± 3.41**	87.81 ± 1.36	82.42 ± 1.10	**90.98 ± 0.93**
	20	82.86 ± 2.39	73.02 ± 5.89	78.76 ± 1.69	91.03 ± 0.47	84.54 ± 0.90	93.53 ± 0.47
GCN [KW17]	1	47.72 ± 15.33	48.94 ± 10.24	58.61 ± 12.83	**65.22 ± 2.25**	49.46 ± 1.65	82.94 ± 2.17
	2	60.85 ± 14.01	58.06 ± 9.76	60.45 ± 16.20	**83.61 ± 1.49**	**76.90 ± 1.49**	83.61 ± 0.71
	5	73.86 ± 7.97	67.24 ± 4.19	68.69 ± 7.93	86.66 ± 0.43	82.47 ± 0.97	**88.86 ± 1.56**
	10	78.82 ± 5.38	72.18 ± 3.47	72.59 ± 3.19	88.60 ± 0.50	82.53 ± 0.74	90.41 ± 0.35
	20	82.07 ± 2.03	**74.21 ± 2.90**	76.89 ± 3.27	91.09 ± 0.35	82.94 ± 1.54	91.95 ± 0.11
GAT [Vel+]	1	47.86 ± 15.38	50.31 ± 14.27	58.84 ± 12.81	51.13 ± 5.24	37.14 ± 7.81	73.58 ± 8.15
	2	58.30 ± 13.55	55.55 ± 9.19	60.24 ± 14.44	63.12 ± 6.09	65.07 ± 8.86	76.89 ± 4.89
	5	71.04 ± 5.74	67.37 ± 5.08	68.54 ± 5.75	71.65 ± 4.53	71.43 ± 7.34	83.01 ± 3.64
	10	76.31 ± 4.87	71.35 ± 4.92	72.44 ± 3.50	74.71 ± 3.35	76.04 ± 0.35	87.42 ± 2.38
	20	79.92 ± 2.28	73.22 ± 2.90	75.55 ± 4.11	79.95 ± 2.88	80.05 ± 1.81	89.38 ± 2.48
GraphSage [HYL17]	1	43.04 ± 14.01	48.81 ± 11.45	55.53 ± 12.71	61.35 ± 1.35	27.65 ± 2.39	45.36 ± 7.13
	2	53.96 ± 12.18	54.39 ± 11.37	58.97 ± 12.65	76.51 ± 1.31	42.63 ± 4.29	51.93 ± 4.21
	5	68.14 ± 6.95	64.79 ± 5.16	66.07 ± 6.16	**89.06 ± 0.69**	64.83 ± 1.62	78.26 ± 1.93
	10	75.04 ± 5.03	68.90 ± 5.08	70.74 ± 3.11	**89.68 ± 0.39**	74.66 ± 1.29	84.38 ± 1.75
	20	80.04 ± 2.54	72.02 ± 2.82	74.55 ± 3.09	**91.33 ± 0.36**	79.98 ± 0.96	91.29 ± 0.67
MoNet [Mon+17]	1	47.72 ± 15.53	39.13 ± 11.37	56.47 ± 4.67	58.99 ± 5.17	23.78 ± 7.57	34.72 ± 8.18
	2	60.85 ± 14.01	48.52 ± 9.52	61.03 ± 6.93	76.57 ± 4.06	38.19 ± 3.72	43.03 ± 8.22
	5	73.86 ± 7.97	61.66 ± 6.61	67.92 ± 2.50	87.02 ± 1.67	59.38 ± 4.73	71.80 ± 5.02
	10	78.82 ± 5.38	68.08 ± 6.29	71.24 ± 1.54	88.76 ± 0.49	68.66 ± 3.30	78.66 ± 3.17
	20	82.07 ± 2.03	71.52 ± 4.11	76.49 ± 1.75	90.31 ± 0.41	73.66 ± 2.87	88.61 ± 1.18

Table 4.3: Classification accuracy of different GNNs trained with different number of labeled data per class (label #) on six benchmark graph node classification tasks. The highest accuracy is highlighted in bold for each number of labeled data per class. These results show that GRAND++ is more effective in learning with low-labeling rates than GRAND. (Unit: %)

#per class	CORA	CiteSeer	PubMed	CoauthorCS	Computer	Photo
1	1.05	**4.36**	**3.28**	**1.95**	**111.60**	**45.33**
2	1.39	**3.96**	0.34	**4.59**	**7.18**	**4.65**
5	**2.55**	**2.68**	-3.67	-1.96	**15.67**	0.26

Table 4.4: Unpaired t-test scores of GRAND++ v.s. GRAND on six different benchmark graph node classification tasks. With $n = 100$, over 0.95 confidence is equivalent to exceed roughly 1.66 t-test scores. Highlighted are the ones passing the test.

Dataset	#per class	Accuracy Difference	# splits	t-score	p-score
CORA	1	1.06 ± 6.24	100	1.80	0.044
CORA	2	1.45 ± 5.23	100	2.78	0.003

Table 4.5: Paired t-test scores of GRAND++ v.s. GRAND on datasets where unpaired t-test scores are not significant enough.

Tables 4.6-4.8 list the classification accuracy, on the test set, of different benchmark GNN models when they are trained with different numbers of labeled nodes per class.

#labeled nodes per class	1	2	5	10	20
GCN [KW17]	47.72 ± 15.53	60.85 ± 14.01	73.86 ± 7.97	78.82 ± 5.38	82.07 ± 2.03
GAT [Vel+]	47.86 ± 15.38	58.30 ± 13.55	71.04 ± 5.74	76.31 ± 4.87	79.92 ± 2.28
GraphSage [HYL17]	43.04 ± 14.01	53.96 ± 12.18	68.14 ± 6.95	75.04 ± 5.03	80.04 ± 2.54
MoNet [Mon+17]	47.72 ± 15.53	60.85 ± 14.01	73.86 ± 7.97	78.82 ± 5.38	82.07 ± 2.03
Lanczos [Lia+19]	47.41 ± 11.82	60.94 ± 4.00	74.28 ± 3.07	76.12 ± 0.93	79.85 ± 1.82
AdaLanczos [Lia+19]	48.23 ± 11.82	61.46 ± 4.96	74.24 ± 3.25	77.61 ± 1.36	81.03 ± 1.56
GCNN [XQT20]	43.31 ± 11.95	60.28 ± 12.89	72.75 ± 4.21	78.92 ± 1.32	81.89 ± 1.12
GRAND-l [Cha+21a]	52.53 ± 16.40	64.82 ± 11.16	76.07 ± 5.08	80.25 ± 3.40	82.86 ± 2.39
GRAND-nl [Cha+21a]	40.97 ± 14.87	50.59 ± 13.25	65.13 ± 9.14	72.55 ± 6.65	77.76 ± 4.21
GRAND-nl-rw (gdc) [Cha+21a]	52.68 ± 12.48	65.54 ± 10.01	74.94 ± 7.04	80.64 ± 6.19	82.47 ± 1.93
GRAND-nl-rw (two-hop) [Cha+21a]	53.79 ± 17.72	64.50 ± 11.88	74.33 ± 6.28	79.61 ± 4.47	82.37 ± 1.98

Table 4.6: Classification accuracy of different GNNs trained with different numbers of labeled nodes per class. Dataset: CORA.

#labeled nodes per class	1	2	5	10	20
GCN [KW17]	48.94 ± 10.24	58.06 ± 9.76	67.24 ± 4.19	72.18 ± 3.47	74.21 ± 2.90
GAT [Vel+]	50.31 ± 14.27	55.55 ± 9.19	67.37 ± 5.08	71.35 ± 4.92	73.22 ± 2.90
GraphSage [HYL17]	48.81 ± 11.45	54.39 ± 11.37	64.79 ± 5.16	68.90 ± 5.08	72.02 ± 2.82
MoNet [Mon+17]	39.13 ± 11.37	48.52 ± 9.52	61.66 ± 6.61	68.08 ± 6.29	71.52 ± 4.11
Lanczos [Lia+19]	49.16 ± 3.63	57.65 ± 7.60	66.72 ± 9.38	71.01 ± 4.90	72.14 ± 2.00
AdaLanczos [Lia+19]	50.32 ± 7.42	58.35 ± 7.97	67.39 ± 8.20	72.15 ± 4.85	74.33 ± 2.83
GCNN [XQT20]	40.58 ± 15.32	51.71 ± 13.87	63.16 ± 12.26	67.06 ± 5.65	69.84 ± 1.77
GRAND-l [Cha+21a]	50.06 ± 17.98	59.55 ± 10.89	68.37 ± 5.00	71.90 ± 7.66	73.02 ± 5.89
GRAND-nl [Cha+21a]	49.96 ± 18.62	59.57 ± 11.03	68.21 ± 7.08	71.88 ± 6.94	72.84 ± 6.61
GRAND-nl-rw (gdc) [Cha+21a]	50.35 ± 17.74	59.98 ± 10.32	68.39 ± 5.81	71.83 ± 7.26	72.81 ± 6.94
GRAND-nl-rw (two-hop) [Cha+21a]	50.20 ± 17.90	59.95 ± 10.48	68.05 ± 5.49	71.92 ± 7.34	72.72 ± 6.85

Table 4.7: Classification accuracy of different GNNs trained with different numbers of labeled nodes per class. Dataset: CiteSeer.

#labeled nodes per class	1	2	5	10	20
GCN [KW17]	58.61 ± 12.83	60.45 ± 16.20	68.69 ± 7.93	72.59 ± 3.19	76.89 ± 3.27
GAT [Vel+]	58.84 ± 12.81	60.24 ± 14.44	68.54 ± 5.75	72.44 ± 3.50	75.55 ± 4.11
GraphSage [HYL17]	55.53 ± 12.71	58.97 ± 12.65	66.07 ± 6.16	70.74 ± 3.11	74.55 ± 3.09
MoNet [Mon+17]	56.47 ± 4.67	61.03 ± 6.93	67.92 ± 2.50	71.24 ± 1.54	76.49 ± 1.75
Lanczos [Lia+19]	60.12 ± 6.37	63.65 ± 6.97	70.61 ± 4.50	73.01 ± 3.27	78.35 ± 1.84
AdaLanczos [Lia+19]	61.07 ± 5.16	64.11 ± 6.88	69.05 ± 3.00	72.79 ± 2.74	78.10 ± 1.91
GCNN [XQT20]	60.78 ± 20.64	65.14 ± 19.45	72.72 ± 10.82	76.47 ± 6.03	79.24 ± 3.45
GRAND-l [Cha+21a]	62.11 ± 10.58	69.00 ± 7.55	73.98 ± 5.08	76.33 ± 3.41	78.76 ± 1.69
GRAND-nl [Cha+21a]	61.75 ± 11.12	69.16 ± 8.46	72.35 ± 5.35	76.03 ± 3.72	78.55 ± 1.59
GRAND-nl-rw (gdc) [Cha+21a]	61.70 ± 10.74	69.42 ± 8.21	72.39 ± 5.25	75.32 ± 3.45	78.30 ± 1.43
GRAND-nl-rw (two-hop) [Cha+21a]	61.65 ± 12.09	68.49 ± 8.99	72.68 ± 5.92	75.72 ± 3.50	78.77 ± 1.88

Table 4.8: Classification accuracy of different GNNs trained with different numbers of labeled nodes per class. Dataset: PubMed.

4.6.3 Time-dependent attention and graph rewiring

The previous experimental results show that GRAND++-l enhances the accuracy of GRAND-l in the cases when the labeled training data is limited and when the network is deep. Here, we explore the same strategy for GRAND-nl and GRAND-nl-rw; we name the corresponding models with the new source term GRAND++-nl and GRAND++-nl-rw,

respectively. Table 4.9 compares GRAND-nl and GRAND-nl-rw with the corresponding model with a source term. We see that overall GRAND++-nl (GRAND++-nl-rw) outperforms GRAND-nl (GRAND-nl-rw) when the network is deep, i.e., T is big.

Model	Depth (T)	GRAND-nl [Cha+21a]	GRAND-nl-rw [Cha+21a]	GRAND++-nl (ours)	GRAND++-nl-rw (ours)
CORA	1	**79.70 ± 1.88**	79.07 ± 3.05	79.24 ± 1.48	79.24 ± 1.48
	4	82.31 ± 0.91	82.47 ± 1.32	**82.64 ± 0.89**	82.23 ± 1.14
	16	82.11 ± 1.42	82.05 ± 1.31	**83.24 ± 0.20**	81.48 ± 1.07
	32	79.42 ± 0.64	81.01 ± 0.81	81.21 ± 0.37	**82.20 ± 1.15**
CiteSeer	1	71.84 ± 2.98	**71.84 ± 2.66**	70.45 ± 2.12	71.74 ± 1.37
	16	72.65 ± 2.42	73.06 ± 2.98	72.48 ± 1.10	**73.29 ± 1.37**
	64	70.29 ± 2.58	69.65 ± 2.50	72.64 ± 0.93	**73.38 ± 0.95**
	128	65.19 ± 6.77	65.45 ± 7.18	**74.24 ± 0.70**	74.23 ± 0.70
PubMed	1	77.93 ± 1.27	77.93 ± 1.26	**78.01 ± 0.68**	78.01 ± 0.68
	4	77.95 ± 1.28	78.02 ± 1.14	**78.41 ± 0.88**	78.17 ± 0.93
	16	76.51 ± 2.73	76.88 ± 2.57	**78.43 ± 0.78**	78.12 ± 0.87

Table 4.9: Classification accuracy of GRAND and GRAND++ variants of different depth trained 20 labeled data per class. The highest accuracy is highlighted in bold for each of the depths $T = 1, 4, 16, 32, 64$, and 128. We test T only up to 16 for PubMed and up to 32 for 32 since the neural ODE solver failed for larger T. (Unit: %)

We further explore the effects of the source term for GRAND-nl and GRAND-nl-rw in the low-labeling rate regimes. Table 4.10 compares GRAND-nl and GRAND-nl-rw with the corresponding model with a source term. We see that GRAND-nl and GRAND-nl-rw are almost always worse than the vanilla GRAND-l, consistent with the results reported in [Cha+21a]. GRAND++-nl and GRAND++-nl-rw cannot help learning at low labeling rates anymore. However, when the labeling rates are not low, GRAND++-nl or GRAND++-nl-rw can outperform GRAND-nl and GRAND-nl-rw, even outperform GRAND++.

4.6.4 Datasets and experimental settings

Graph node classification dataset. Following [Cha+21a], we consider the largest connected component of seven graph node classification datasets, including CORA, CiteSeer,

Model	label #	GRAND-nl [Cha+21a]	GRAND-nl-rw [Cha+21a]	GRAND++-nl (ours)	GRAND++-nl-rw (ours)
CORA	1	50.55 ± 15.68	50.63 ± 17.71	48.89 ± 11.51	47.94 ± 11.06
	2	65.06 ± 9.35	61.24 ± 16.19	59.96 ± 7.90	58.25 ± 11.97
	5	76.93 ± 3.10	76.50 ± 3.91	74.01 ± 1.73	74.25 ± 1.99
	10	79.60 ± 2.69	79.38 ± 3.25	80.14 ± 0.69	80.18 ± 0.40
	20	82.22 ± 1.93	82.14 ± 2.49	83.24 ± 0.20	81.48 ± 1.07
CiteSeer	1	50.25 ± 17.66	50.20 ± 17.90	49.65 ± 5.45	53.10 ± 5.51
	2	59.87 ± 10.89	59.95 ± 10.48	59.16 ± 8.13	60.26 ± 5.10
	5	68.21 ± 5.08	68.05 ± 5.48	66.13 ± 2.09	67.81 ± 1.97
	10	71.88 ± 6.94	71.92 ± 7.34	68.84 ± 2.84	71.45 ± 1.64
	20	72.84 ± 6.61	72.72 ± 6.85	72.52 ± 1.24	73.87 ± 1.35
PubMed	1	66.97 ± 10.07	67.69 ± 7.89	63.85 ± 4.86	67.45 ± 3.88
	2	69.17 ± 2.46	69.42 ± 2.13	66.98 ± 5.30	69.11 ± 1.80
	5	72.56 ± 3.36	72.68 ± 2.52	71.49 ± 1.53	72.05 ± 3.67
	10	76.03 ± 3.73	75.32 ± 3.45	74.94 ± 2.15	75.09 ± 2.88
	20	78.55 ± 1.59	78.30 ± 1.43	78.41 ± 0.99	79.44 ± 0.56

Table 4.10: Classification accuracy of the variants of GRAND and GRAND++ models trained with different numbers of labeled data per class (#per class) on graph node classification tasks. (Unit: %)

PubMed, coauthor graph CoauthorCS, and Amazon co-purchasing graphs Computer and Photo, and a large scale ogbn-arxiv dataset. For completeness, we list the number of classes, the number of features, and the number of nodes and edges of each dataset in Table 4.11. More detailed information can be found in [Cha+21a].

Dataset	Classes	Features	#Nodes	#Edges
CORA	7	1433	2485	5069
CiteSeer	6	3703	2120	3679
PubMed	3	500	19717	44324
CoauthorCS	15	6805	18333	81894
Computer	10	767	13381	245778
Photo	8	745	7487	119043
ogbn-arxiv	40	128	169343	1166243

Table 4.11: Summary of the graph node classification datasets.

Depth of GRAND and GRAND++ for the results in Table 4.3. Table 4.12 lists the fine-tuned T for the results in Table 4.3. Due to the limited time, we only search around the value of optimal T for GRAND with grid spacing 0.1.

Model	CORA	CiteSeer	PubMed	CoauthorCS	Computer	Photo
GRAND++-l	18.3	8.0	13.0	4.0	3.2	3.6
GRAND-l	18.2948	7.8741	12.9423	3.2490	3.5824	3.6760

Table 4.12: The value of the fine-tuned T, i.e. depth of the continuous-depth GNNs, for GRAND and GRAND++ in learning with different labeling results, and the corresponding accuracy are reported in Table 4.3. The values of T for GRAND++ are adopted from the paper [Cha+21a].

4.7 Concluding Remarks

We propose GRAND++, which augments graph neural diffusion with a source term. We present some theory that connects the model to a random walk formulation on graphs. GRAND++ outperforms many existing GNNs for graph deep learning with very deep architectures and when the number of labeled data is limited. GRAND++ can be regarded as coupled ODE system in which each ODE has an external force term. As such, it is natural to consider if advanced techniques in accelerating training, test, and inference of neural ODEs can be leveraged to improve the efficiency and accuracy of GRAND++, in particular high-order neural ODEs [DDT19; YHL19; Nor+20; Xia+21] and noise injection [Wan+19]. It is interesting to note that the second-order neural ODE can be connected to the wave equation in the graph setting, which can automatically bypass over-smoothing. We leave studying the second-order neural ODE on graphs as future work.

CHAPTER 5

Proximal Implicit ODE Solvers for Accelerating Learning Neural ODEs

5.1 Introduction

Neural ODEs [Che+18] are a class of continuous-depth neural networks, which can be considered as the continuous limit of the ODE-motivated neural networks [HR17; Lu+18]. Neural ODEs are particularly suitable for learning complex dynamics from irregularly-sampled sequential data, see, e.g., [Che+18; RCD19; DDT19; Mas+20; Nor+20]. Neural ODEs are used in many applications, including image classification and generation [Che+18], learning dynamical systems [RCD19], modeling probabilistic distributions of complex data [Gra+19; Yan+19; Jia+20], and scientific computing [Kim+21; DRF21; Bak+22a]. Recently, neural ODEs have been employed for building continuous-depth graph neural networks (GNNs), achieving remarkable results for deep graph learning [Pol+19; XQT20; Cha+21a; Tho+22] and providing great potential for scientific simulation [San+20; Umm+20; Li+19b; Bap+20; DEK20; Pfa+21]. Mathematically, a neural ODE is given by the following first-order ODE:

$$\frac{d\boldsymbol{h}(t)}{dt} = \boldsymbol{f}(\boldsymbol{h}(t), t, \theta), \quad \boldsymbol{h}(0) = \boldsymbol{h}_0, \tag{5.1}$$

where $\boldsymbol{f}(\boldsymbol{h}(t), t, \theta) \in \mathbb{R}^d$ is specified by a neural network parameterized by θ, e.g., a two-layer feed-forward neural network. Starting from the input $\boldsymbol{h}(0)$, neural ODEs learn the representation of the input and perform prediction by solving (5.1) from the initial time $t = 0$ to the terminal time T using a numerical ODE solver with a given error tolerance, often with adaptive step size solver or adaptive solver for short [DP80a]. Solving (5.1) from $t = 0$ to T in

a single pass with an adaptive solver requires evaluating $\boldsymbol{f}(\boldsymbol{h}(t), t, \theta)$ at various timestamps, with the computational complexity counted by the number of forward function evaluations (forward NFEs), which is nearly proportional to the computational time, see [Che+18] for details.

The adjoint method [Bit63], is a memory-efficient algorithm for training neural ODEs. For the sake of presentation, if we regard the output $\boldsymbol{h}(T)$ as the prediction and denote the loss between $\boldsymbol{h}(T)$ and the ground truth as $\mathcal{L} := \mathcal{L}(\boldsymbol{h}(T))$. Let $\boldsymbol{a}(t) := \partial\mathcal{L}/\partial\boldsymbol{h}(t)$ be the adjoint state, then we have (see e.g., [Che+18; Bit63] for details)

$$\frac{d\mathcal{L}}{d\theta} = \int_0^T \boldsymbol{a}(t)^\top \frac{\partial \boldsymbol{f}(\boldsymbol{h}(t), t, \theta)}{\partial \theta} dt, \tag{5.2}$$

with $\boldsymbol{a}(t)$ satisfying the following adjoint ODE

$$\frac{d\boldsymbol{a}(t)}{dt} = -\boldsymbol{a}(t)^\top \frac{\partial}{\partial \boldsymbol{h}} \boldsymbol{f}(\boldsymbol{h}(t), t, \theta), \tag{5.3}$$

which is solved numerically from $t = T$ to 0 and also requires the evaluation of the right-hand side of (5.3) at various timestamps, with the computational complexity, or number of evaluation of the function $-\boldsymbol{a}(t)^\top \frac{\partial}{\partial \boldsymbol{h}} \boldsymbol{f}(\boldsymbol{h}(t), t, \theta)$, measured by the backward NFEs.

5.1.1 Computational bottlenecks of neural ODEs

Since both neural ODE (5.1) and its adjoint ODE (5.3) are usually high dimensional; direct application of implicit ODE solvers requires solving a system of high dimensional nonlinear equations, which is computationally inefficient, see Section 5.4.1 for an illustration. As such, explicit solvers — especially the explicit adaptive solvers, e.g., the Dormand-Prince method [DP80a] — are the current default numerical solvers for neural ODE's training and testing.

Dormand-Prince Method: In this section, we briefly review the scheme, error control, and step size rule of the Dormand-Prince method, which is an explicit adaptive numerical ODE solver. The one step calculation, from t_k to t_{k+1} with step size s, in the Dormand-Prince

method for solving (5.1) is summarized below: First, we update $\boldsymbol{h}_k$ to $\boldsymbol{h}_{k+1}$ using Runge-Kutta method of order 4.

$$
\begin{aligned}
\boldsymbol{k}_1 &= s\boldsymbol{f}(t_k, \boldsymbol{h}_k) \\
\boldsymbol{k}_2 &= s\boldsymbol{f}\Big(t_k + \frac{1}{5}s, \boldsymbol{h}_k + \frac{1}{5}\boldsymbol{k}_1\Big) \\
\boldsymbol{k}_3 &= s\boldsymbol{f}\Big(t_k + \frac{3}{10}s, \boldsymbol{h}_k + \frac{3}{40}\boldsymbol{k}_1 + \frac{9}{40}\boldsymbol{k}_2\Big) \\
\boldsymbol{k}_4 &= s\boldsymbol{f}\Big(t_k + \frac{4}{5}s, \boldsymbol{h}_k + \frac{44}{45}\boldsymbol{k}_1 - \frac{56}{15}\boldsymbol{k}_2 + \frac{32}{9}\boldsymbol{k}_3\Big) \\
\boldsymbol{k}_5 &= s\boldsymbol{f}\Big(t_k + \frac{8}{9}s, \boldsymbol{h}_k + \frac{19372}{6561}\boldsymbol{k}_1 - \frac{25360}{2187}\boldsymbol{k}_2 + \frac{64448}{6561}\boldsymbol{k}_3 - \frac{212}{729}\boldsymbol{k}_4\Big) \\
\boldsymbol{k}_6 &= s\boldsymbol{f}\Big(t_k + s, \boldsymbol{h}_k + \frac{9017}{3168}\boldsymbol{k}_1 - \frac{355}{33}\boldsymbol{k}_2 - \frac{46732}{5247}\boldsymbol{k}_3 + \frac{49}{176}\boldsymbol{k}_4 - \frac{5103}{18656}\boldsymbol{k}_5\Big) \\
\boldsymbol{k}_7 &= s\boldsymbol{f}\Big(t_k + s, \boldsymbol{h}_k + \frac{35}{384}\boldsymbol{k}_1 + \frac{500}{1113}\boldsymbol{k}_3 + \frac{125}{192}\boldsymbol{k}_4 - \frac{2187}{6784}\boldsymbol{k}_5 + \frac{11}{84}\boldsymbol{k}_6\Big)
\end{aligned}
$$

And the $\boldsymbol{h}_{k+1}$ is calculated as

$$
\boldsymbol{h}_{k+1} = \boldsymbol{h}_k + \frac{35}{384}\boldsymbol{k}_1 + \frac{500}{1113}\boldsymbol{k}_3 + \frac{500}{192}\boldsymbol{k}_4 - \frac{2187}{6784}\boldsymbol{k}_5 + \frac{11}{84}\boldsymbol{k}_6.
$$

Second, we update $\boldsymbol{h}_k$ to $\boldsymbol{h}'_{k+1}$ by Runge-Kutta method of order 5 as

$$
\boldsymbol{h}'_{k+1} = \boldsymbol{h}_k + \frac{5179}{57600}\boldsymbol{k}_1 + \frac{7571}{16695}\boldsymbol{k}_3 + \frac{393}{640}\boldsymbol{k}_4 - \frac{92097}{339200}\boldsymbol{k}_5 + \frac{187}{2100}\boldsymbol{k}_6 + \frac{1}{40}\boldsymbol{k}_7.
$$

We consider $\|\boldsymbol{h}'_{k+1} - \boldsymbol{h}_{k+1}\|$ as the error in $\boldsymbol{h}_{k+1}$, and given error tolerance ϵ we select the adaptive step size at this step to be

$$
s_{opt} = s\left(\frac{\epsilon s}{2\|\boldsymbol{h}_{k+1} - \boldsymbol{h}'_{k+1}\|}\right)^{\frac{1}{5}}.
$$

Many other adaptive ODE solvers exist, and some are also used to learn neural ODEs, e.g., the adaptive Heun's method, which is a second-order ODE solver. More adaptive step solvers can be found at [AHS11].

The step size of explicit solvers is constrained by numerical accuracy and numerical stability, and the latter is often the dominating factor when the ODE system is stiff. The high-order method usually has a smaller stability region than the low-order method; the typical behavior of explicit ODE solvers when applied to solve the stiff ODE system is that

the high-order solver requires a smaller step size than the low-order method. In contrast, the high-order method can take a much larger step size than the low-order method for solving non-stiff ODEs. To demonstrate this issue, we consider training a recently proposed neural ODE-based GNN, named graph neural diffusion (GRAND) [Cha+21a], for the benchmark CoauthorCS graph node classification; the detailed experimental settings can be found in Section 5.4.3. GRAND parametrizes the right-hand side of (5.1) with the Laplacian operator, resulting in a class of diffusion models. The diffusion model is stiff from the numerical ODE viewpoint since the ratio between the magnitude of the largest and smallest eigenvalues is infinite. Figure 5.1 plots the error tolerance of the adaptive solver vs. forward and backward NFEs of different adaptive solvers — including adaptive Heun [SM03] and two Dormand-Prince methods [DP80a] (DOPRI5 and DOPRI8) — for training GRAND for CoauthorCS node classification. We see that 1) all three adaptive solvers require significant NFEs in solving both neural ODE and its adjoint ODE; as the error tolerance reduces both forward and backward NFEs increase rapidly. And 2) the high-order scheme, e.g., DOPRI8 requires more NFEs than the low-order scheme, indicating that the stability is a dominating factor in choosing the step size. Moreover, it is worth mentioning that both DOPRI5 and DOPRI8 often fail to solve the adjoint ODE when a large tolerance is used, based on our experiments. Even worse, the explicit ODE solvers can be computationally prohibitive even using a very large error tolerance, and this happens particularly when numerical stability constraints the step size; see Section 5.4.1 for a numerical illustration. The stability constraint of learning stiff neural ODEs using explicit solvers and the high computational cost of direct application of implicit solvers motivate us to study the question: *Can we modify the implicit solvers to adapt them for learning high dimensional neural ODEs with significantly reduced computational costs than the existing benchmark solvers?*

5.1.2 Our contribution

We answer the above question affirmatively by considering learning neural ODE-style models using the scalable proxy of a few celebrated implicit ODE solvers — including

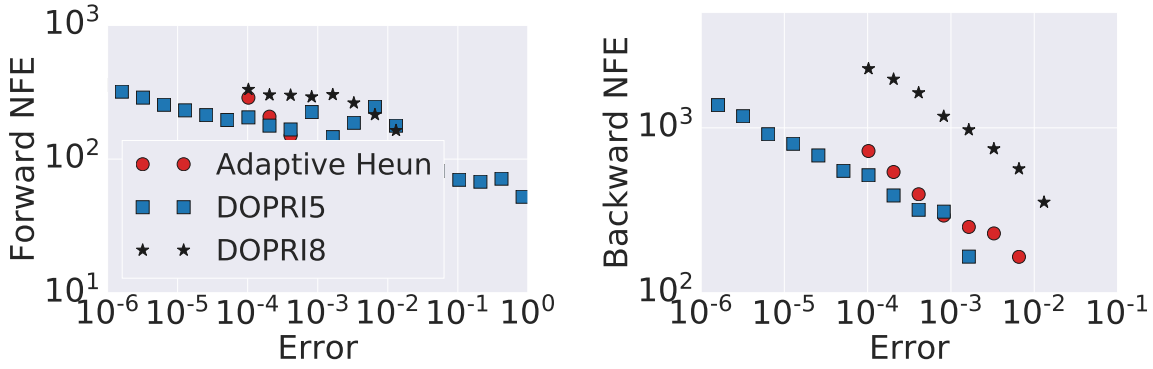


Figure 5.1: Error tolerance vs. forward and backward NFEs of different adaptive solvers for training the GRAND model for CoauthorCS graph node classification.

backward Euler, backward differentiation formulas (BDFs), and Crank-Nicolson — leveraging the proximal operator [PB14]. These proximal solvers reformulate implicit ODE solvers as variational problems. Each solver contains inner-outer iterations, where the inner iterations approximate a one-step update of the implicit solver, and the outer iterations solve the ODE over time. Leveraging fast optimization algorithms for solving the inner optimization problem, these proximal solvers are remarkably faster than explicit adaptive ODE solvers in learning certain benchmark neural ODEs. We summarize the major benefits of proximal solvers below:

- The proximal implicit ODE solvers are scalable for solving very high dimensional ODEs due to using scalable optimization algorithms for solving the corresponding optimization problems.
- Due to the implicit nature of proximal algorithms, they are much less encumbered by the numerical stability issue than explicit solvers. Therefore, the proximal algorithms allow the use of very large step sizes for neural ODEs' training and testing.
- To achieve the same numerical accuracy in solving stiff neural ODEs, proximal algorithms can save significant NFEs and computational time in both forward and backward propagation.

- Training neural ODE-style models using proximal algorithms maintains and often improves the generalization accuracy of the model compared to using benchmark adaptive solvers.

5.1.3 More related works

In this part, we discuss some representative related works in three directions: reducing NFEs in learning neural ODEs, advances in proximal algorithms, and algorithms for learning neural ODEs.

Reducing NFEs in learning neural ODEs by model design and regularization Several algorithms have been developed to reduce the NFEs for learning neural ODEs. They can be classified into three categories: 1. Improving the ODE model and neural network architecture, and notable works in this direction include augmented neural ODEs [DDT19], high-order neural ODEs [Nor+20], heavy-ball neural ODEs [Xia+21], and neural ODEs with depth variance [Mas+20]. These models can reduce the forward NFEs significantly, and heavy-ball neural ODEs can also reduce the backward NFEs remarkably. 2. Learning neural ODEs with regularization, including weight decay [Gra+19], regularizing ODE solvers and learning dynamics [Fin+20; Kel+20; Pol+20; Gho+20; Pal+21]. And 3. Input augmentation [DDT19] and data control [Mas+20]. Our work focuses on accelerating learning neural ODEs using the proximal form of implicit ODE solvers and can be used jointly with the above acceleration methods.

Advances of proximal algorithms Proximal algorithms have been the workhorse for solving nonsmooth, large-scale, or distributed optimization problems [PB14]. The core matter is the evaluation of proximal operators [BC+11]. The proximal algorithms are widely used in statistical computing and machine learning [PSW15], image processing [BT09], matrix completion [MS12; ZYX17], computational optimal transport [SKL20; PC+19], game theory and optimal control [Att+08], etc. Proximal operators can be viewed as backward Euler

method, see Section 5.2 for a brief mathematical introduction. From an implicit solver viewpoint, a proximal formulation of the backward Euler method has been applied to the stochastic gradient descent training of neural networks [Cha+18] and solving clustering problems [Yin+18a]. We consider accelerating learning neural ODEs using proximal implicit ODE solvers, especially when the efficiency of explicit adaptive solvers is limited by numerical stability.

Developments of efficient neural ODE learning algorithms Solving a high dimensional ODE is required in both forward and backward propagation of learning neural ODEs. The default numerical solvers are explicit adaptive Runge-Kutta schemes [PT92; Che+18], especially the Dormand-Prince method [DP80a]. To solve both forward and backward ODEs accurately, the adaptive solvers will evaluate the right-hand side of ODEs at many intermediate timestamps, causing tremendous computational burden. Checkpoint schemes have been proposed to reduce the computational cost [GKB19; Zhu+20a], often reducing computational cost by compromising memory efficiency. Recently, the symplectic adjoint method has been proposed [MMY21], which solves neural ODEs using a symplectic integrator and obtains the exact gradient (up to rounding error) with memory efficiency. Approximating gradients using interpolation instead of the adjoint method [Dau+20] has also been used to accelerate learning neural ODEs. Our proposed proximal solvers can be integrated with these new algorithms for efficient training of neural ODEs.

5.1.4 Notation

We denote scalars by lower or upper case letters; vectors and matrices by lower and upper case boldface letters, respectively. We denote the magnitude of a complex number z as $|z|$. For a vector $\boldsymbol{x} = (x_1, \ldots, x_d)^\top \in \mathbb{R}^d$, we use $\|\boldsymbol{x}\| := (\sum_{i=1}^d |x_i|^2)^{1/2}$ and $\|\boldsymbol{x}\|_\infty := \max_{i=1}^d |x_i|$ to denote its L_2-norm and L_∞-norm, respectively. We denote the vector whose entries are all 0s as $\mathbf{0}$. For two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, we denote their inner product as $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$. For a matrix $\boldsymbol{A}$, we use $\boldsymbol{A}^\top$ and $\boldsymbol{A}^{-1}$, and $\|\boldsymbol{A}\|$ to denote its transpose, inverse, and spectral norm, respectively.

We denote the identity matrix as $\boldsymbol{I}$. For a function $f(\boldsymbol{x}) : \mathbb{R}^d \to \mathbb{R}$, we denote $\nabla f(\boldsymbol{x})$ and $\nabla^2 f(\boldsymbol{x})$ as its gradient and Hessian, respectively. We denote $a = \mathcal{O}(b)$, if there is a constant C such that $a \leq Cb$.

5.2 Proximal Algorithms for Learning Neural ODEs

In this section, we formulate the proximal formulation of several celebrated implicit ODE solvers, including backward Euler, Crank-Nicolson, and BDFs.

5.2.1 A proximal viewpoint of the backward Euler solver

We first consider solving neural ODE (5.1) using the proximal backward Euler solver. Directly discretizing (5.1) using the backward Euler scheme with a constant step size s gives the following system of linear equations

$$\boldsymbol{h}_{k+1} = \boldsymbol{h}_k + s\boldsymbol{f}(\boldsymbol{h}_{k+1}). \tag{5.4}$$

When $\boldsymbol{f}$ is a nonlinear function, the solution $\boldsymbol{h}_{k+1}$ is often non-unique for a given $\boldsymbol{h}_k$, and we only need to find one particular solution of (5.4). In particular, the system of nonlinear equations (5.4) can be solved by the Newton–Raphson method (NR), with the numerical updates for $\boldsymbol{h}_{k+1}$ being initialized by $\boldsymbol{h}_k$.

But when the dimension of $\boldsymbol{f}$ is very high, solving (5.4) using NR can be computationally prohibitive. Nevertheless, suppose $\boldsymbol{f}(\boldsymbol{z})$ is the gradient of a function $-F(\boldsymbol{z})$. In that case, we can find a solution $\boldsymbol{h}_{k+1}$ of the nonlinear equation (5.4) by finding a local minimum of the following optimization problem

$$\boldsymbol{h}_{k+1} = \arg\min_{\boldsymbol{z}} \Big\{ \frac{1}{2s}\|\boldsymbol{z} - \boldsymbol{h}_k\|_2^2 + F(\boldsymbol{z}) \Big\}. \tag{5.5}$$

To see why a local minimum of the minimization problem (5.5) is a solution to (5.4), we notice that if $\boldsymbol{h}_{k+1}$ is a local minimum of $G(\boldsymbol{z}) = \frac{1}{2s}\|\boldsymbol{z} - \boldsymbol{h}_k\|_2^2 + F(\boldsymbol{z})$, then we have

$$\frac{d}{d\boldsymbol{z}} \Big(\frac{1}{2s}\|\boldsymbol{z} - \boldsymbol{h}_k\|_2^2 + F(\boldsymbol{z}) \Big) \Big|_{\boldsymbol{z}=\boldsymbol{h}_{k+1}} = 0,$$

replacing $\boldsymbol{z}$ with $\boldsymbol{h}_{k+1}$ in the above equation, we see that the local minimum of $G(\boldsymbol{z})$ satisfies the backward Euler scheme (5.4) (note that $\nabla F(\boldsymbol{z}) = -\boldsymbol{f}(\boldsymbol{z})$). We call the scheme (5.5) proximal backward Euler.

Based on the proximal formulation of the backward Euler method (5.4), we can solve neural ODE (5.1) using an inner-outer iteration scheme. The outer iterations perform backward Euler iterations over time, and the inner iterations solve the optimization problem formulated in (5.5). We summarize the inner-outer iteration scheme for the proximal backward Euler solver in the algorithm below.

Algorithm 1 Proximal backward Euler for solving (5.1)

Require: Step size $s > 0$, inner iteration number n

for $k = 1, 2, \ldots$

step 1: Let $\boldsymbol{z}^0 = \boldsymbol{h}_k$

step 2: Start from $\boldsymbol{z}^0$ solve inner problem

$$\arg\min_{\boldsymbol{z}} \Big\{ \frac{1}{2s} \|\boldsymbol{z} - \boldsymbol{h}_k\|_2^2 + F(\boldsymbol{z}) \Big\}.$$

step 3: Let $\boldsymbol{h}_{k+1} = \boldsymbol{z}^n$ (solution to the inner problem).

end for

5.2.1.1 Solving the inner minimization problem

Another piece of the recipe is how to effectively solve the inner minimization problem of proximal backward Euler (5.5), i.e., the following optimization problem

$$\arg\min_{\boldsymbol{z}} G(\boldsymbol{z}) := \frac{1}{2s} \|\boldsymbol{z} - \boldsymbol{h}_k\|_2^2 + F(\boldsymbol{z}). \tag{5.6}$$

A simple yet efficient algorithm for solving (5.6) is gradient descent (GD), which updates as follows:

$$\boldsymbol{z}^{i+1} = \boldsymbol{z}^i - \eta \nabla G(\boldsymbol{z}^i) \text{ for } i = 1, 2, \ldots, n-1,$$

where $\eta > 0$ is the step size and $\nabla G(\boldsymbol{z}^i) = (\boldsymbol{z}^i - \boldsymbol{h}_k)/s - \boldsymbol{f}(\boldsymbol{z}^i)$. Here, we do not need to know the exact form of $F(\boldsymbol{z})$ since $\nabla G(\boldsymbol{z}^i)$ can be represented by $\boldsymbol{f}(\boldsymbol{z}^i)$ itself, and the GD

update becomes

$$\boldsymbol{z}^{i+1} = \boldsymbol{z}^i - \eta\Big(\frac{\boldsymbol{z}^i - \boldsymbol{h}_k}{s} - \boldsymbol{f}(\boldsymbol{z}^i)\Big). \tag{5.7}$$

There are various off-the-shelf acceleration schemes that exist to accelerate the convergence of (5.7), e.g., Nesterov accelerated gradient (NAG) [Nes83], NAG with adaptive restart (Restart) [Rd17], and GD with nonlinear conjugate gradient style momentum, e.g., Fletcher–Reeves momentum (FR) [FR64; WY20]. In particular, FR, which will be used in our experiments, updates $\boldsymbol{z}^i$ as follows:

$$\boldsymbol{p}^i = \Big(\frac{\boldsymbol{z}^i - \boldsymbol{h}_k}{s} - \boldsymbol{f}(\boldsymbol{z}^i)\Big) + \beta_i \boldsymbol{p}^{i-1}, \quad \boldsymbol{z}^{i+1} = \boldsymbol{z}^i - \eta \boldsymbol{p}^i, \tag{5.8}$$

where $\boldsymbol{p}_0 = \boldsymbol{0}$ and the scalar $\beta_0 = 0$ and

$$\beta_i = \frac{\nabla G(\boldsymbol{z}^i)^\top \nabla G(\boldsymbol{z}^i)}{\nabla G(\boldsymbol{z}^{i-1})^\top \nabla G(\boldsymbol{z}^{i-1})} \text{ if } i \geq 1. \tag{5.9}$$

We can also employ L-BFGS [LN89] to solve the inner minimization problem (5.6). Based on our testing, GD with FR momentum usually outperforms the other gradient-based methods listed above, see Section 5.4.1 for a comparison of different optimization algorithms for solving the 1D diffusion problem.

Remark 6. *The above discussion of gradient-based optimization algorithms assumes the existence of $F(\boldsymbol{z})$ whose gradient is $-\boldsymbol{f}(\boldsymbol{z})$. When such a function $F(\boldsymbol{z})$ does not exist, we can still use the iteration* (5.7) *to solve the inner minimization problem, which we can regard as a fixed point iteration (FP). Moreover, we can accelerate the convergence of* (5.7) *using the Anderson acceleration [And65; WN11], and we leave it as future work. Based on our numerical tests, gradient-based optimization algorithms work quite well in solving the inner optimization problem* (5.6) *across all studied benchmark tasks.*

Stopping criterion of inner solvers Given an error tolerance ϵ of the inner optimization solver, we stop the inner iteration if $\|\boldsymbol{z}^{i+1} - \boldsymbol{z}^i\| \leq \epsilon$.

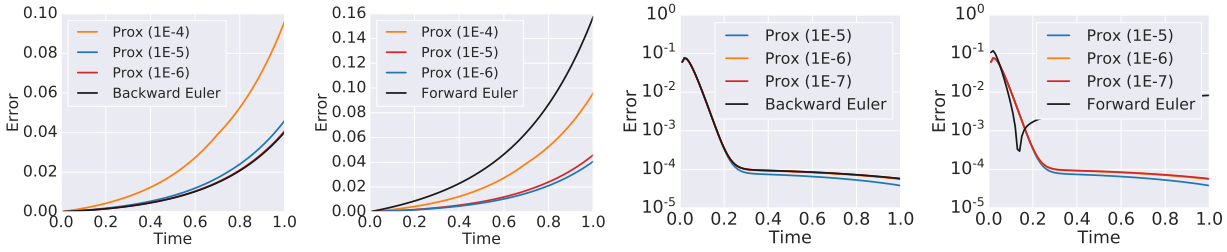


Figure 5.2: (a)/(c) Time vs. numerical errors of the backward Euler method and the proximal backward Euler (Prox) with different inner error tolerances for solving ODE (5.10)/(5.11). As the error of the inner solver decreases, the proximal backward Euler approaches the backward Euler. (b)/(d) Comparison of proximal backward Euler using different inner solver accuracy against the forward Euler for solving the same problem in (a)/(c). We see that the proximal backward Euler method remarkably outperforms the forward Euler scheme. In (d), the error of proximal and backward Euler decays as time increases dues to the ODE's stiff behavior and the solution profile.

5.2.1.2 Implicit, explicit vs. proximal solvers

Before presenting more proximal implicit solvers, we compare forward Euler, backward Euler, and the proximal backward Euler for solving the following benchmark 1D ODE, which comes from [AHS11]

$$\frac{dh(t)}{dt} = 3h(t) - 2\cos(t) - 4\sin(t) \tag{5.10}$$

with initial condition $h(0) = 1$. We use the integration time step size 0.01 for all the above ODE solvers. Figure 5.2 (a) plots the error between numerical and analytic solutions of proximal backward Euler with different inner solver tolerances and the backward Euler solver. It is evident that the *proximal backward Euler approximates backward Euler quite well, and the approximation becomes more accurate as the accuracy of the inner solver enhances*. Figure 5.2 (b) contrasts the proximal backward Euler with the forward Euler solver. We see that with an appropriate tolerance of the inner solver, proximal backward Euler accumulates error

slower than the forward Euler solver as the time increases. These advantages of the proximal backward Euler solver shed light on improving learning neural ODEs and alleviating error accumulation.

We further compare forward Euler, backward Euler, and proximal backward Euler solvers for solving the following benchmark stiff ODE [EG96]

$$\frac{dh}{dt} = -50(h(t) - \cos(t)). \tag{5.11}$$

with initial condition $h(0) = 1$. We use the integration time step size 0.01 for all the above ODE solvers. Figure 5.2 (c) plots the error between the numerical and exact solutions of proximal backward Euler with different inner solver tolerances and the backward Euler solver. Again, proximal backward Euler approximates backward Euler quite well. Figure 5.2 (d) contrasts the proximal backward Euler with the forward Euler solver, and we see that forward Euler performs much worse than the proximal backward Euler solver.

5.2.2 Proximal form of Crank-Nicolson

A single-step, second-order extension for backward Euler is the Crank-Nicolson scheme, given by

$$\boldsymbol{h}_{k+1} = \boldsymbol{h}_k + \frac{s}{2}(\boldsymbol{f}\left(\boldsymbol{h}_{k+1}\right) + \boldsymbol{f}\left(\boldsymbol{h}_k\right)).$$

The proximal formulation of the Crank-Nicolson scheme is given as follows [DF20]

$$\boldsymbol{h}_{k+1} = \arg\min_{\boldsymbol{z}} \left\{ \frac{\|\boldsymbol{z} - \boldsymbol{h}_k\|^2}{s} + F(\boldsymbol{z}) + \langle \boldsymbol{z} - \boldsymbol{h}_k, \nabla F(\boldsymbol{h}_k) \rangle \right\} \tag{5.12}$$

5.2.3 The proximal backward differentiation formula (BDF) methods

The backward Euler scheme has first-order accuracy, i.e., discretizing the ODE (5.1) using (5.4) with step size s has error $\mathcal{O}(s)$. BDFs are higher-order multi-step methods that can be formulated as follows

$$a_s \boldsymbol{h}_{k+1} = \boldsymbol{A}_s(\boldsymbol{h}_k, \boldsymbol{h}_{k-1}, \ldots) - s\boldsymbol{f}(\boldsymbol{h}_{k+1}),$$

where a_s is a constant, and $\boldsymbol{A}_s$ is a linear function of $\boldsymbol{h}_k, \boldsymbol{h}_{k-1}, \ldots$. The backward Euler method corresponds to the case when $s = 1$, and $a_s = 1$ and $A_1(\boldsymbol{h}_k) = \boldsymbol{h}_k$. The second-, third-, and fourth-order BDFs are given as follows:

- BDF2:
$$\boldsymbol{h}_{k+1} = \arg\min_{\boldsymbol{z}} \left\{ \frac{\|\boldsymbol{z} - \boldsymbol{h}_k\|^2}{s} - \frac{\|\boldsymbol{z} - \boldsymbol{h}_{k-1}\|^2}{4s} + F(\boldsymbol{z}) \right\}.$$

- BDF3:
$$\boldsymbol{h}_{k+1} = \arg\min_{\boldsymbol{z}} \left\{ \frac{3\|\boldsymbol{z} - \boldsymbol{h}_k\|^2}{2s} - \frac{3\|\boldsymbol{z} - \boldsymbol{h}_{k-1}\|^2}{4s} + \frac{\|\boldsymbol{z} - \boldsymbol{h}_{k-2}\|^2}{6s} + F(\boldsymbol{z}) \right\}.$$

- BDF4:
$$\boldsymbol{h}_{k+1} = \arg\min_{\boldsymbol{z}} \left\{ \frac{2\|\boldsymbol{z} - \boldsymbol{h}_k\|^2}{s} - \frac{3\|\boldsymbol{z} - \boldsymbol{h}_{k-1}\|^2}{2s} + \frac{2\|\boldsymbol{z} - \boldsymbol{h}_{k-2}\|^2}{3s} - \frac{\|\boldsymbol{z} - \boldsymbol{h}_{k-2}\|^2}{8s} + F(\boldsymbol{z}) \right\}$$

5.3 Stability and Convergence Analysis

In this section, we analyze the stability of proximal implicit ODE solvers. In particular, we first compare the stability region between explicit and implicit solvers. Then we bound the gap between solutions of implicit and proximal implicit ODE solvers, indicating that proximal implicit solvers allow the use of a larger step size than explicit solvers, thus saving computational cost over explicit solvers for solving stiff problems. Moreover, we will analyze the convergence of proximal implicit solvers.

5.3.1 Linear stability: Implicit vs. explicit solvers

Explicit solvers require a small step size for numerical stability guarantee in solving stiff ODEs. Consider the linear ODE

$$\boldsymbol{h}' = \boldsymbol{A}\boldsymbol{h}, \quad \boldsymbol{A} \in \mathbb{R}^{d \times d}. \tag{5.13}$$

Assume $\boldsymbol{A}$ has spectrum $\sigma(\boldsymbol{A}) = \{\lambda_j\}_{j=1}^d$ and $\mathrm{R}e(\lambda_j) < 0$ for $j = 1, \dots d$ [1], one can define the stiffness ratio as

$$Q = \frac{\max_{\lambda \in \sigma(\boldsymbol{A})} |\mathrm{R}e(\lambda)|}{\min_{\lambda \in \sigma(\boldsymbol{A})} |\mathrm{R}e(\lambda)|}$$

We say the ODE is stiff if $Q \gg 1$. The *linear stability domain* $\mathcal{D}$ of the underlying numerical method is the set of all numbers $z := s\lambda_j$ for $j = 1, \dots, d$, such that $\lim_{k\to\infty} \boldsymbol{h}_k = \boldsymbol{0}$, where $\boldsymbol{h}_k$ is the numerical solution of (5.13) at the k-th step. Table 5.1 lists the linear stability domain of some single-step numerical schemes described in Section 5.2. In general, the implicit method has a much larger linear stability domain than the explicit method.

Numerical methods	Linear stability domain
Forward Euler	$\mathcal{D}_{\mathrm{F}E} = \{z \in \mathbb{C} \mid \lvert 1 + z\rvert < 1\}$
Backward Euler	$\mathcal{D}_{\mathrm{B}E} = \{z \in \mathbb{C} \mid \lvert 1 - z\rvert > 1\}$
Crank-Nicolson	$\mathcal{D}_{CR} = \{z \in \mathbb{C} \mid \lvert\frac{1+z/2}{1-z/2}\rvert < 1\}$
DOPRI5	$\mathcal{D}_{\mathrm{D}P} = \{z \in \mathbb{C} \mid \lvert F_{1/2}(z)\rvert < 1\}$

Table 5.1: Linear stability domains of several single step numerical ODE solvers. $|F_{1/2}(z)| < 1$ stands for $|F_1(z)| < 1$ and $|F_2(z) < 1|$ where $F_1(z) = \sum_{r=0}^5 \frac{z^r}{r!} + \frac{z^6}{600}$ and $F_2(z) = \sum_{r=0}^4 \frac{z^r}{r!} + \frac{1097z^5}{120000} + \frac{161z^6}{120000} + \frac{z^7}{24000}$, see [Ise09; DP80a] for details.

5.3.2 Effects of the error of the inner solver

Compared to the explicit methods, implicit methods obtain their next step by solving an implicit equation with certain error tolerance in each step. In this subsection, we consider the effects of the error of the inner solver on the proximal backward Euler solver for an illustration. In particular, we consider solving the equation $\frac{d\boldsymbol{h}}{dt} = \boldsymbol{f}(\boldsymbol{h})$ where $\boldsymbol{f}$ is Lipschitz

[1] The analysis can be generalized to the case when the system has eigenvalues of zero real part. In particular, in the direction of eigenvectors that belong to zero eigenvalues simply stay the same. In other words, these components simply do not affect the ODE's stiffness.

continuous with constant $L_{\boldsymbol{f}}$. We define

$$u := \sup_{\boldsymbol{y},\boldsymbol{h}} \frac{\langle \boldsymbol{y}-\boldsymbol{h}, \boldsymbol{f}(\boldsymbol{y})-\boldsymbol{f}(\boldsymbol{h})\rangle}{\|\boldsymbol{y}-\boldsymbol{h}\|^2},$$

and u exists since it is bounded above, i.e.,

$$\begin{aligned}\frac{\langle \boldsymbol{y}-\boldsymbol{h}, \boldsymbol{f}(\boldsymbol{y})-\boldsymbol{f}(\boldsymbol{h})\rangle}{\|\boldsymbol{y}-\boldsymbol{h}\|^2} &= \frac{1}{\|\boldsymbol{y}-\boldsymbol{h}\|}\left\langle \frac{\boldsymbol{y}-\boldsymbol{h}}{\|\boldsymbol{y}-\boldsymbol{h}\|}, \boldsymbol{f}(\boldsymbol{y})-\boldsymbol{f}(\boldsymbol{h})\right\rangle \\ &\leq \frac{\|\boldsymbol{f}(\boldsymbol{y})-\boldsymbol{f}(\boldsymbol{h})\|}{\|\boldsymbol{y}-\boldsymbol{h}\|} \leq L_{\boldsymbol{f}}.\end{aligned}$$

Unlike Lipschitz constant, u remains small on stiff problems as it is not affected by swift decays which usually occurs in stiff problems. Backward Euler with error in each step can be written as

$$\frac{\boldsymbol{h}_{k+1}-\boldsymbol{h}_k}{s} = \boldsymbol{f}(\boldsymbol{h}_{k+1}) + \boldsymbol{\epsilon}_k,$$

where $\|\boldsymbol{\epsilon}_k\| \leq \epsilon$ is the error for the inner loops of proximal backward Euler with $\epsilon > 0$. Also, let $\boldsymbol{y}_k$ be the exact solution of backward Euler, i.e., $\boldsymbol{y}_k$ satisfies

$$\frac{\boldsymbol{y}_{k+1}-\boldsymbol{y}_k}{s} = \boldsymbol{f}(\boldsymbol{y}_{k+1}).$$

Subtract the above two equations gives

$$\boldsymbol{h}_{k+1}-\boldsymbol{y}_{k+1}-s\big(\boldsymbol{f}(\boldsymbol{h}_{k+1})-\boldsymbol{f}(\boldsymbol{y}_{k+1})\big) = \boldsymbol{h}_k-\boldsymbol{y}_k+s\boldsymbol{\epsilon}_k. \tag{5.14}$$

By triangle inequality and Lipschitz continuous assumption on $\boldsymbol{f}$, we have

$$\big\|\boldsymbol{h}_{k+1}-\boldsymbol{y}_{k+1}-s\big(\boldsymbol{f}(\boldsymbol{h}_{k+1})-\boldsymbol{f}(\boldsymbol{y}_{k+1})\big)\big\| \leq \|\boldsymbol{h}_k-\boldsymbol{y}_k\|+s\|\boldsymbol{\epsilon}_k\|. \tag{5.15}$$

Let $\boldsymbol{y} = \boldsymbol{y}_{k+1}$ and $\boldsymbol{h} = \boldsymbol{h}_{k+1}$ in the definition of u, we have

$$u \geq \frac{\big\langle \boldsymbol{y}_{k+1}-\boldsymbol{h}_{k+1}, \boldsymbol{f}(\boldsymbol{y}_{k+1})-\boldsymbol{f}(\boldsymbol{h}_{k+1})\big\rangle}{\|\boldsymbol{y}_{k+1}-\boldsymbol{h}_{k+1}\|^2},$$

i.e.,

$$u\|\boldsymbol{y}_{k+1}-\boldsymbol{h}_{k+1}\|^2 \geq \big\langle \boldsymbol{y}_{k+1}-\boldsymbol{h}_{k+1}, \boldsymbol{f}(\boldsymbol{y}_{k+1})-\boldsymbol{f}(\boldsymbol{h}_{k+1})\big\rangle.$$

Therefore, we have

$$\begin{aligned}
&\big\|\boldsymbol{h}_{k+1}-\boldsymbol{y}_{k+1}-s\big(\boldsymbol{f}(\boldsymbol{h}_{k+1})-\boldsymbol{f}(\boldsymbol{y}_{k+1})\big)\big\|^2\\
=&\|\boldsymbol{h}_{k+1}-\boldsymbol{y}_{k+1}\|^2+s^2\|\boldsymbol{f}(\boldsymbol{h}_{k+1})-\boldsymbol{f}(\boldsymbol{y}_{k+1})\|^2\\
&-2s\big\langle\boldsymbol{h}_{k+1}-\boldsymbol{y}_{k+1},\boldsymbol{f}(\boldsymbol{y}_{k+1})-\boldsymbol{f}(\boldsymbol{h}_{k+1})\big\rangle \qquad (5.16)\\
\geq&\|\boldsymbol{h}_{k+1}-\boldsymbol{y}_{k+1}\|^2-2s\big\langle\boldsymbol{h}_{k+1}-\boldsymbol{y}_{k+1},\boldsymbol{f}(\boldsymbol{y}_{k+1})-\boldsymbol{f}(\boldsymbol{h}_{k+1})\big\rangle\\
\geq&(1-2su)\|\boldsymbol{h}_{k+1}-\boldsymbol{y}_{k+1}\|^2.
\end{aligned}$$

Combining (5.15) and (5.16), we have

$$(1-2su)\|\boldsymbol{h}_{k+1}-\boldsymbol{y}_{k+1}\|^2\leq\big(\|\boldsymbol{h}_k-\boldsymbol{y}_k\|^2+s\|\boldsymbol{\epsilon}_k\|\big)^2.$$

i.e.,

$$\|\boldsymbol{h}_{k+1}-\boldsymbol{y}_{k+1}\|\leq\frac{1}{\sqrt{1-2su}}\Big(\|\boldsymbol{h}_k-\boldsymbol{y}_k\|^2+s\epsilon\Big).$$

It follows that

$$\|\boldsymbol{h}_k-\boldsymbol{y}_k\|\leq\sum_{j=0}^{k-1}\left(\prod_{i=j}^{k-1}\frac{1}{\sqrt{1-2su}}\right)s\epsilon\leq\frac{(1-2su)^{-k/2}}{u}\epsilon.$$

Notice that $k=(t_k-t_0)/s$, where t_0 and t_k are the initial time and the time at the k-th integration step, thus we have

$$\frac{(1-2su)^{-k/2}}{u}\epsilon=\Big((1-2su)^{-\frac{1}{2su}}\Big)^{(t_k-t_0)u}\frac{\epsilon}{u}=\mathcal{O}(\epsilon).$$

Therefore, the error between $\boldsymbol{h}_k$ and $\boldsymbol{y}_k$ is bounded by $\mathcal{O}(\epsilon)$. The above argument can be generalized to the proximal BDFs and proximal Crank-Nicolson.

5.3.3 Energy stability and convergence

Besides the linear stability, there is another notion of stability, known as *energy stability*. We say a numerical method is energy stable if $F(\boldsymbol{h}_{k+1})\leq F(\boldsymbol{h}_k)$ for all k [SXY19]. As an advantage of the proximal formulation, one can show that the backward Euler is unconditionally energy stable [Xu+19] for arbitrary s even for non-convex $F(\boldsymbol{z})$ (here we assume that there exists $F(\boldsymbol{z})$ such that $\boldsymbol{f}(\boldsymbol{z})=-\nabla F(\boldsymbol{z})$). *The energy stability guarantees that the numerical*

solver takes each step toward minimizing $F(\boldsymbol{z})$, which further lead to the convergence of the numerical scheme [Wan+21c]. More precisely, we have the following result

Proposition 10. *If $F(\boldsymbol{z})$ is continuous, coercive and bounded from below, for any choice of $s > 0$, there exists $\boldsymbol{h}_{k+1}$ solves (5.4), such that $F(\boldsymbol{h}_{k+1}) \leq F(\boldsymbol{h}_k)$. Moreover, for non-convex $F(\boldsymbol{z})$, $\boldsymbol{h}_{k+1}$ is unique provided $s \leq -1/\lambda_1$, where $\lambda_1 < 0$ is the smallest eigenvalue of $\nabla^2 F$.*

Proof. For any given $\boldsymbol{h}_k$, recall the definition of $G(\boldsymbol{z})$ in (5.6).

If $F(\boldsymbol{z})$ is a convex function, clearly, $G(\boldsymbol{z})$ admits a unique minimizer. If $F(\boldsymbol{z})$ is non-convex, denote $\lambda_1 \leq 0$ be the smallest eigenvalue of $\nabla^2 F$. Direct computation shows that

$$\nabla^2 G(\boldsymbol{z}) = \frac{1}{s}\boldsymbol{I} + \nabla^2 F(\boldsymbol{z}), \tag{5.17}$$

which indicates that $G(\boldsymbol{z})$ is a convex function if $s \leq -1/\lambda_1$. Notice that

$$\lim_{|\boldsymbol{z}|\to\infty} G(\boldsymbol{z}) = \infty,$$

$G(\boldsymbol{z})$ admits a unique minimizer in $\mathbb{R}^d$.

If $G(\boldsymbol{z})$ is non-convex, we define $\mathcal{S} = \{G(\boldsymbol{z}) \leq G(\boldsymbol{h}_k)\}$. By the coerciveness and continuity of $F(\boldsymbol{z})$, $\mathcal{S}$ is a non-empty, bounded, and closed set. Hence, $G(\boldsymbol{z})$ admits a minimizer (not unique) $\boldsymbol{z}^*$ in $\mathcal{S}$. We can take $\boldsymbol{h}_{k+1} = \boldsymbol{z}^*$, then

$$\frac{1}{2s}\|\boldsymbol{h}_{k+1} - \boldsymbol{h}_k\|^2 + F(\boldsymbol{h}_{k+1}) \leq F(\boldsymbol{h}_k),$$

which gives us $F(\boldsymbol{h}_{k+1}) \leq F(\boldsymbol{h}_k)$. Hence, the scheme is unconditionally energy stable.

For the sequence $\{\boldsymbol{h}_k\}$, since

$$\|\boldsymbol{h}_k - \boldsymbol{h}_{k-1}\|^2 \leq 2s(F(\boldsymbol{h}_{k-1}) - F(\boldsymbol{h}_k)),$$

we have

$$\sum_{k=1}^{N} \|\boldsymbol{h}_k - \boldsymbol{h}_{k-1}\|^2 \leq 2s(F(\boldsymbol{h}_0) - F(\boldsymbol{h}_N)) \leq C,$$

for some constant C that is independent with N. Hence

$$\lim_{N\to\infty} \|\boldsymbol{h}_N - \boldsymbol{h}_{N-1}\| = 0.$$

For the sequence $\{\boldsymbol{h}_k\}$, since

$$\|\boldsymbol{h}_k - \boldsymbol{h}_{k-1}\|^2 \le 2s(F(\boldsymbol{h}_{k-1}) - F(\boldsymbol{h}_k)),$$

we have

$$\sum_{k=1}^{N} \|\boldsymbol{h}_k - \boldsymbol{h}_{k-1}\|^2 \le 2s(F(\boldsymbol{h}_0) - F(\boldsymbol{h}_N)) \le C,$$

for some constant C that is independent with N. Hence

$$\lim_{N\to\infty} \|\boldsymbol{h}_N - \boldsymbol{h}_{N-1}\| = 0.$$

Moreover, since

$$\boldsymbol{h}_N = \boldsymbol{h}_{N-1} - s\nabla F(\boldsymbol{h}_N),$$

we have

$$\lim_{N\to\infty} \nabla F(\boldsymbol{h}_N) = 0,$$

so every limit of a subsequence of $\{\boldsymbol{h}_k\}$ is a stationary point of $F(\boldsymbol{h})$. □

Similar energy stability analysis holds for other proximal solvers, e.g., for BDF2, one can show there exists $\boldsymbol{h}_{k+1}$ such that

$$F(\boldsymbol{h}_{k+1}) + \frac{\|\boldsymbol{h}_{k+1} - \boldsymbol{h}_k\|^2}{4s} \le F(\boldsymbol{h}_k) + \frac{\|\boldsymbol{h}_k - \boldsymbol{h}_{k-1}\|^2}{4s}.$$

For Crank-Nicolson, the proximal formulation (5.12) leads to

$$F(\boldsymbol{h}_{k+1}) + \langle \boldsymbol{h}_{k+1} - \boldsymbol{h}_k, \nabla F(\boldsymbol{h}_k)\rangle \le F(\boldsymbol{h}_k).$$

These energy stability result can further lead to a certain convergence of these schemes [MP19].

Figure 5.3: Final step error vs. NFEs of different solvers in solving the 1D diffusion equation (5.18). Adaptive solvers require many more NFEs than proximal solvers, and NFEs required by explicit solvers are almost independent of the error since the step sizes here are constrained by numerical stability.

5.4 Experimental Results

In this section, we validate the efficacy of proximal solvers and contrast their performances with benchmark adaptive neural ODE solvers in solving the 1D diffusion problem and learning continuous normalizing flows (CNFs) [Gra+19] and GRAND [Cha+21a]. All experiments are conducted on a server with 4 NVIDIA RTX 3090 GPUs.

5.4.1 Solving 1D diffusion equation

We consider solving the 1D diffusion equation $\frac{\partial h}{\partial t} = \frac{\partial^2 h}{\partial x^2}$ for $t \in [0, 1]$ and $x \in [0, 1]$ with a periodic boundary condition. We initialize $u(x, 0)$ with the standard Gaussian. We partition $[0, 1]$ into uniform grids $\{x_i\}_{i=1}^{128}$ and discretize $\partial^2 h / \partial x^2$ using central finite difference, resulting in the following coupled ODEs

$$\frac{d\boldsymbol{h}}{dt} = -\boldsymbol{L}\boldsymbol{h}, \tag{5.18}$$

where $\boldsymbol{L} \in \mathbb{R}^{128\times128}$ is the Laplacian matrix of a cyclic graph scaled by $1/\Delta x^2$ with $\Delta x = 1/127$ being the spatial resolution, and $\boldsymbol{h} = [h(x_1), \ldots, h(x_{128})]$. Then we apply different ODE solvers to solve (5.18). It is worth noting that the diffusion equation has an infinite domain of dependence, thus in method of lines discretization, the time step to be taken for explicit methods has to be much finer compared to the spatial discretization to guarantee the numerical stability.

We compare our proximal methods against the predominantly used Adaptive Heun and DOPRI5 in solving (5.18). As the matrix $\boldsymbol{L}$ is circulant, we can compute the exact solution of (5.18) using the discrete Fourier transform for comparison. Figure 5.3 compares the NFEs of proximal methods and two benchmark adaptive solvers with different errors at the final time. Here, the reported NFE of proximal solvers is the product of inner and outer iteration numbers and the number of function evaluations of the scheme per update. That is, the reported NFE is the absolute count of the function evaluations performed by the solver. Similar for all the experiments in the following. For each final error, we use a fixed step size for each proximal solver with FR for solving the inner minimization problem. A list of the configuration of each ODE solver can be found in Table 5.2. Figure 5.3 shows that adaptive solvers require many more NFEs than proximal solvers. *NFEs required by explicit solvers are almost independent of the discretization error of the solver, revealing that the step sizes are constrained by the numerical stability.* Each iteration of higher-order BDFs requires more NFEs than lower-order BDF schemes, showing a tradeoff between controlling numerical error and using high-order schemes.

We list the numerical integration step size and the corresponding final step error of each ODE solver in Table 5.2. Here, we use FR optimizer to solve the inner optimization problem with step size 0.1. We stop the inner solver when $\|\boldsymbol{z}^{i+1} - \boldsymbol{z}^i\| \leq 5 \times 10^{-9}$.

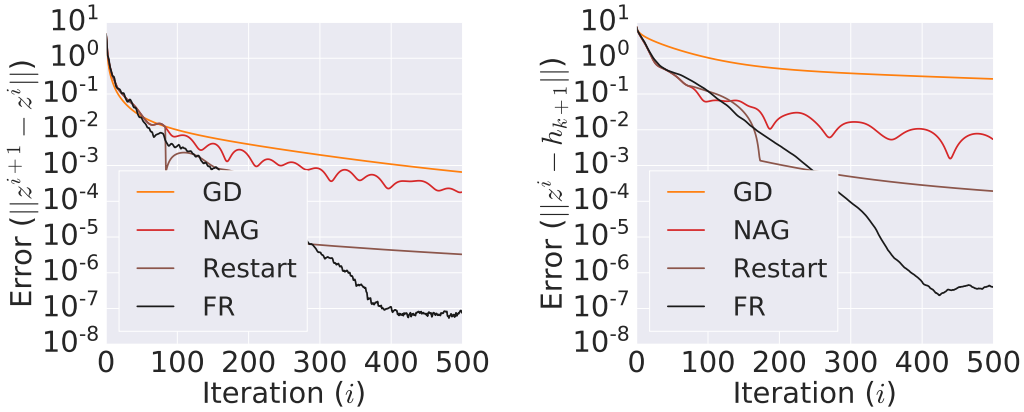
Convergence of inner solvers An inner optimization algorithm is required for the proximal solver. We compare the efficiency of a few gradient-based optimization algorithms, including

Proximal Algorithm	Numerical integration step size s	Optimization step size η	Final step error
Backward Euler	1/2000	0.1	4.68e−7
Crank Nicolson	1/2000	0.1	1.75e−6
BDF2	1/2000	0.1	1.36e−6
BDF3	1/2000	0.1	5.71e−7
BDF4	1/2000	0.1	1.15e−6

Table 5.2: The configuration of different solvers for solving the 1D Diffusion equation in Section 5.4.1. We use GD with FR optimization to solve the inner optimization problem with the step size 0.1. Figure 5.3 is generated by considering a range of inner optimization tolerances from 10^{-3} to 10^{-7}. The final step error is the error between the numerical solution at $t = 1$ and the exact solution. We report in the following figure the smallest final step error for each proximal algorithm.

GD, NAG, Restart, and FR, for one pass of proximal backward Euler. The comparison of different optimizers using the same step size 0.1 (Figure 5.4) shows that different solvers perform similarly to each other at the beginning of iterations, while as the iteration goes, FR performs best among the four considered solvers. The performance of different optimization algorithms further show the rationale for solving the inner problem using gradient descent with FR momentum. We will use FR as the default inner solver for all experiments below.

Proximal vs. nonlinear solvers We further verify the computational advantages of proximal backward Euler over solving nonlinear equations (5.4) using other nonlinear root-finding algorithms, including FP and NR. Figure 5.5 shows the time required by different solvers when the spatial interval $[0, 1]$ is discretized into different number of grids, controlling



(a) (b)

Figure 5.4: Convergence comparison of different inner solvers for the proximal backward Euler. (a) Convergence of $\|\boldsymbol{z}^{i+1} - \boldsymbol{z}^i\|$, and (b) convergence of $\|\boldsymbol{z}^i - \boldsymbol{h}_{k+1}\|$ for $k = 0$.

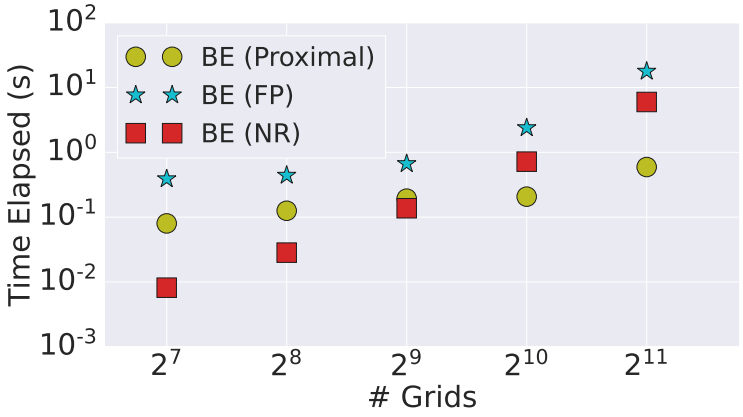


Figure 5.5: Computational time of solving 1D diffusion equation with different number of grids by backward Euler (BE) using proximal, FP, and NR solvers.

the scale of the problem. Proximal solver outperforms FP and NR in computational efficiency, and the margin gets wider as the size of the problem increases. NR is not scalable to high dimensional scenarios since it requires evaluating Jacobian, which is computationally prohibitive.

5.4.2 Learning CNFs

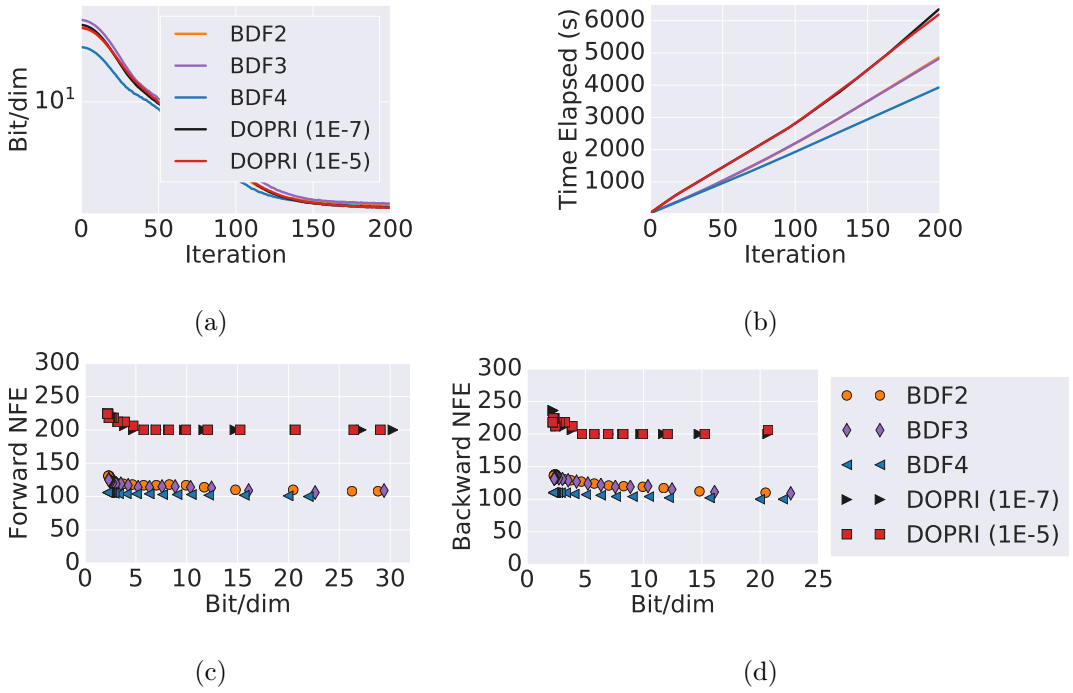


Figure 5.6: Contrasting BDFs with DOPRI5 using different error tolerances for training CNFs for MNIST image generation. BDFs converge as well as DOPRI5 using very small error tolerances (a) but require much fewer NFEs (c) and (d) and take less computational time (b) in solving both neural ODE and its adjoint ODE.

We train CNFs for MNIST [LC10] generation using proximal solvers and contrast them to adaptive solvers. In particular, we use the FFJORD approach outlined in [Gra+19], including converting the log-likelihood objective function into the pixel-wise measurement bits/dim. We use an architecture of a multi-scale encoder and two CNF blocks, containing convolutional layers of dimension $64 \to 64 \to 64 \to 64$ with a uniform stride length of 1 and a softplus activation function. We run the model using the proximal BDFs with FR and DOPRI5 with

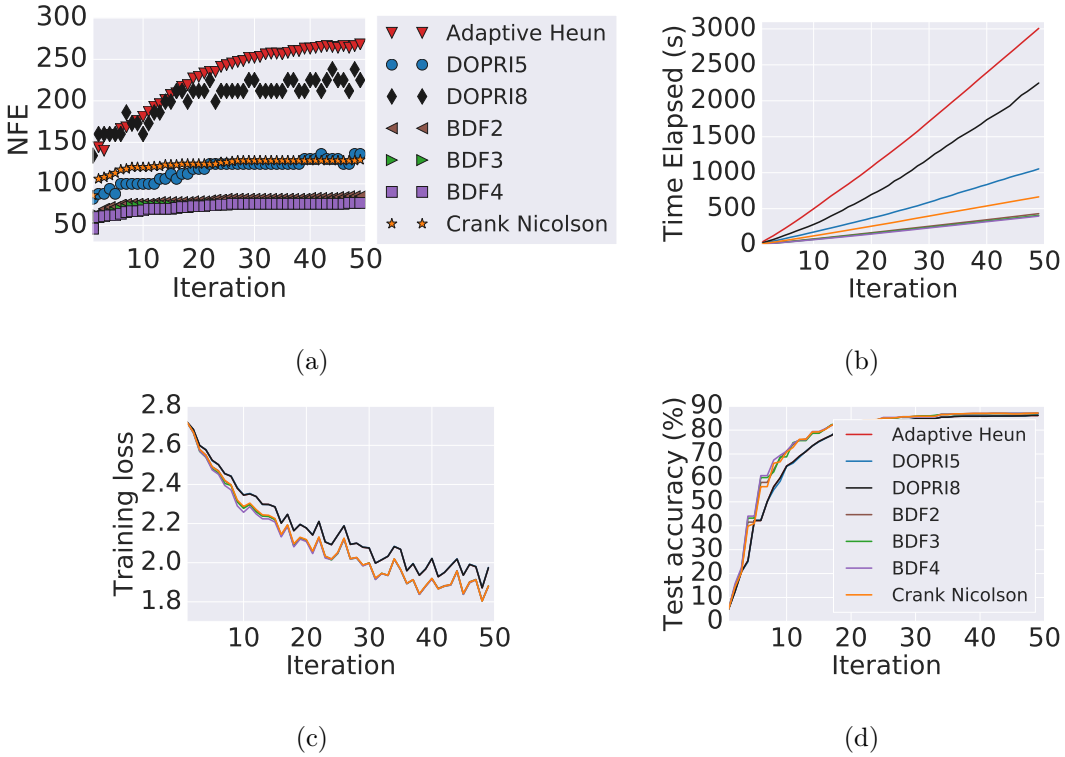


Figure 5.7: Comparison of proximal solvers with adaptive solvers in training GRAND for the CoauthorCS node classification. Proximal solvers require much fewer NFEs in each iteration (a) and takes less total computational time (b) than adaptive solvers but converges as well as adaptive solvers in training loss (c) and validation accuracy (d).

relative tolerance of 10^{-5} and 10^{-7}, respectively. All the other experimental settings are adapted from [Gra+19]. The proximal BDFs are tuned to have an error profile which was consistent with the DOPRI5 as shown in Figure 5.6 (a). We choose $t_0 = 0$ and $T = 1$ and the step sizes for numerical integration and the inner optimizer are 0.2 and 0.3, respectively. We terminate the inner optimization solver once $\|\boldsymbol{z}^{i+1} - \boldsymbol{z}^i\| \leq 2 \times 10^{-3}$. A comparison of computational time and the forward and backward NFEs are depicted in Figure 5.6 (b), (c), and (d), respectively. We see that proximal BDF2, BDF3, and BDF4 require approximately half of the NFEs as DOPRI5 to reach similar training curves.

5.4.3 Training GRAND

GRAND is a continuous-depth graph neural network proposed in [Cha+21a]. It consists of a dense embedding layer, a diffusion layer, and a dense output layer, with ReLU activation function in between. The diffusion layer takes $\boldsymbol{u}(t_0)$ as input and solves (5.19) below to time T

$$\frac{d\boldsymbol{u}(t)}{dt} = \boldsymbol{A}(\boldsymbol{u}, \theta)\boldsymbol{u} - \boldsymbol{u}, \tag{5.19}$$

where $\boldsymbol{A}$ is the graph attention function [Vel+] with learnable parameter set θ. This ODE to be solved in GRAND is nonlinear and diffusive, which is particularly challenging to solve numerically.

We compare the performance of proximal solvers with three major adaptive ODE solver options (Adaptive Heun, DOPRI5, DOPRI8) on training the GRAND model for CoauthorCS node classification. The CoauthorCS graph contains 18333 nodes and 81894 edges, and each node is represented by a 6805 dimensional vector. The graph nodes contains 15 classes, and we aim to classify the unlabelled nodes. We choose $t_0 = 0$ and $T = 10$ as our starting and ending times of the ODE (5.19). For each solver, we run the experiment for 50 epochs and record the average NFEs per epoch and accuracy of trained models. For adaptive solvers we choose tolerance 10^{-4}, and for proximal solvers we choose the integration step size 1.25 (8 steps in total) so that for the first iteration with same input admits output with error within 10^{-3}. For optimization method we use Fletcher-Reeves with learning rate 0.3, and stop when error is below 10^{-4} tolerance. All the other experimental settings are adapted from [Cha+21a]. Figure 5.7 shows the advantage of proximal methods over adaptive solvers in training efficiency and maintaining the convergence of the training.

5.5 Single-step, Multi-stage Implicit Schemes

Another natural extension of the backward Euler scheme is the single-step, multi-stage schemes whose proximal form are proposed in [ZEG20], and we formulated those schemes in Algorithm A.1. Clearly, the 1-stage scheme is the backward Euler scheme. The authors

of [ZEG20] give conditions on the coefficient $\gamma_{m,i}$. However, it is quite tedious to get those coefficients. For example, a second-order unconditionally stable scheme can be constructed by taking the matrix $\mathbf{\Lambda}$ — whose entries are $\{\gamma_{m,i}\}_{m,i=1}^{3}$ that appeared in Algorithm A.1 — as follows

$$\begin{pmatrix} \gamma_{1,0} & 0 & 0 \\ \gamma_{2,0} & \gamma_{2,1} & 0 \\ \gamma_{3,0} & \gamma_{3,1} & \gamma_{3,2} \end{pmatrix} = \begin{pmatrix} 5 & 0 & 0 \\ -2 & 6 & 0 \\ -2 & \frac{3}{14} & \frac{44}{7} \end{pmatrix}.$$

Furthermore, the matrix $\mathbf{\Lambda}$ of a third-order scheme is

$$\begin{pmatrix} 11.17 & 0 & 0 & 0 & 0 & 0 \\ -7.5 & 19.43 & 0 & 0 & 0 & 0 \\ -1.05 & -4.75 & 13.98 & 0 & 0 & 0 \\ 1.8 & 0.05 & -7.83 & 13.8 & 0 & 0 \\ 6.2 & -7.17 & -1.33 & 1.63 & 11.52 & 0 \\ -2.83 & 4.69 & 2.46 & -11.55 & 6.68 & 11.95 \end{pmatrix}$$

These multi-stage algorithms suffer from computational inefficiency in training neural ODEs. Note that for an M-stage scheme, one need to solve M optimization problems in each iteration.

Algorithm 2 Proximal single-step, multi-stage scheme for solving (5.1)

Require: Step size $s > 0$, stage M

for $k = 1, 2, \ldots$

step 1: Let $\boldsymbol{z}^0 = \boldsymbol{h}_k$

for $m = 1, \ldots, M$ solve the inner problem

$$\boldsymbol{z}^m = \arg\min_{\boldsymbol{z}} \left\{ \sum_{i=0}^{m-1} \frac{\gamma_{m,i}}{2s} \|\boldsymbol{z} - \boldsymbol{z}^i\|^2 + F(\boldsymbol{z}) \right\},$$

step 3: Let $\boldsymbol{h}_{k+1} = \boldsymbol{z}^M$.

end for

5.6 Concluding Remarks

We propose accelerating learning neural ODEs using scalable proximal implicit ODE solvers, including proximal BDFs and proximal Crank-Nicolson. These proximal schemes approach the corresponding implicit schemes as the accuracy of the inner solver enhances. Compared to the existing adaptive explicit ODE solvers, the proximal solvers are better suited for solving stiff ODEs for numerical stability guarantees. We validate the efficiency of proximal solvers on learning benchmark graph neural diffusion and continuous normalizing flows. A particular limit of the studied proximal solvers is that we need to control the step size rather than the tolerance, making the user interface different from existing adaptive explicit solvers. There are several interesting future directions. In particular, 1) accelerating the inner solver from the fixed point iteration viewpoint, e.g., leveraging Anderson acceleration, and 2) developing adaptive step size proximal solvers by integrating proximal solvers of different orders.

CHAPTER 6

Conclusion

Performance and computation cost play important roles in deep neural networks, especially Neural ODEs. This thesis focuses on developing methods to relieve computation costs and enhance the performance of Neural ODEs-based models.

In chapter 3, we propose Heavy Ball Neural ODEs (HBNODEs), a new model that incorporates heavy ball structure into Neural ODEs to improve performance and reduce the number of function evaluations (NFEs) in solving both forward and backward ODEs. We prove that the adjoint of HBNODEs is also Heavy Ball ODEs, providing insights into the improved performance of backward ODEs. We also provide theoretical guarantees that HBNODEs alleviate vanishing gradient problems in training NODEs, further improving performance on time-series modeling tasks. We also propose Generalized Heavy Ball Neural ODEs to alleviate the exploding initialization problem for long time series.

In chapter 4, we propose GRAND++, a new model that adds a source term to Graph Neural Diffusion (GRAND). We provide theoretical guarantees on the limiting behaviors of GRAND++ through a random walk formulation on graphs and perform experiments to show that the source term design, as in Poisson Learning [Cal+20], provides improved performance when networks become deeper. We further show that GRAND++ outperforms existing GNNs, including GRAND, on datasets with limited labels.

In chapter 5, we research how proximal solvers can reduce NFEs requirements and speed up training and inference of NODEs. We provide numerical experiments on 1) solving 1-D diffusion equations, 2) solving continuous normalizing flows, and 3) training GRAND, in

order to show that, in particular NODEs applications, the learned ODEs can be stiff, and proximal solvers can reduce NFEs.

There are various directions to extend our research. As Heavy Ball Neural ODEs can be viewed as a design inspired by Heavy Ball acceleration methods in optimization, other optimization methods are also worth investigating. Nguyen et al [Ngu+22] combined Neural ODEs with Nesterov's Accelerated Gradient Method [Nes83], and Cho et al [Cho+22] developed AdamNODE by combining Neural ODEs with Adam stochastic optimization method [KB15].

GRAND++ can be regarded as a coupled ODE system in which each ODE has an external force term. As such, it is natural to consider whether advanced techniques in accelerating training, testing, and inference of neural ODEs can be leveraged to improve the efficiency and accuracy of GRAND++, particularly for high-order neural ODEs [DDT19; YHL19; Nor+20; Xia+21] and noise injection [Wan+19]. In particular, we observe that with a change of variable $\boldsymbol{w} = e^{\frac{1}{2}\gamma t}\boldsymbol{h}$, there is

$$\frac{d^2\boldsymbol{w}(t)}{dt^2} = e^{\frac{1}{2}\gamma t}\frac{d^2\boldsymbol{h}(t)}{dt^2} + \gamma e^{\frac{1}{2}\gamma t}\frac{d\boldsymbol{h}(t)}{dt} + \frac{\gamma^2}{4}e^{\frac{1}{2}\gamma t}\boldsymbol{h}(t) = e^{\frac{1}{2}\gamma t}f(\boldsymbol{h}(t), t, \theta) + \frac{\gamma^2}{4}e^{\frac{1}{2}\gamma t}\boldsymbol{h}(t), \quad (6.1)$$

which is equivalently

$$\frac{d^2\boldsymbol{w}(t)}{dt^2} = e^{\frac{1}{2}\gamma t}f(e^{-\frac{1}{2}\gamma t}\boldsymbol{w}(t), t, \theta) + \frac{\gamma^2}{4}\boldsymbol{w}(t). \quad (6.2)$$

This leads to reaction wave equation when f is the laplacian operator, which might be interesting to investigate for its performance on graphs, as it naturally overcomes over-smoothing problem. We may also potentially investigate how other higher order NODEs [DDT19; YHL19; Nor+20] could be applied to GRAND and GRAND++. As GRAND++ applies to low labeling rate regime, it is also worth exploring how it applies to active learning scenarios where labeling is costly.

To accelerate Neural ODEs training and inference in numerical solver level, we can extend our study in implicit solvers beyond Proximal Implicit ODE Solvers. It is worth studying

how to switch between explicit and implicit solvers, adapt PDE solvers based on different scenarios, and various methods to solve the optimization problems for each proximal steps. It is also worth noting that the adjoint of optimization problem is also an optimization problem [Agr+19], which could potentially allows us to compute the gradients of Neural ODEs through solving optimization problems, without dealing with either the vanilla back-propagation method that needs to store all of the layers information, or the adjoint method that requires to solve backward state equations which could be unstable.

Bibliography

[Agr+19] Akshay Agrawal et al. "Differentiable convex optimization layers". In: *Advances in neural information processing systems* 32 (2019).

[Ahm+20] Imtiaz Ahmad et al. "Solution of multi-term time-fractional PDE models arising in mathematical biology and physics by local meshless method". In: *Symmetry* 12.7 (2020), p. 1195.

[AHS11] Kendall Atkinson, Weimin Han, and David E Stewart. *Numerical solution of ordinary differential equations*. Vol. 108. John Wiley & Sons, 2011.

[And65] Donald G Anderson. "Iterative procedures for nonlinear integral equations". In: *Journal of the ACM (JACM)* 12.4 (1965), pp. 547–560.

[ASB16] Martin Arjovsky, Amar Shah, and Yoshua Bengio. "Unitary evolution recurrent neural networks". In: *International Conference on Machine Learning*. 2016, pp. 1120–1128.

[ASS20] Khalid Anwar, Jamshed Siddiqui, and Shahab Saquib Sohail. "Machine learning-based book recommender system: a survey and new perspectives". In: *International Journal of Intelligent Information and Database Systems* 13.2-4 (2020), pp. 231–248.

[AT16] James Atwood and Don Towsley. "Diffusion-convolutional neural networks". In: *Advances in neural information processing systems*. 2016, pp. 1993–2001.

[Att+08] Hédy Attouch et al. "Alternating proximal algorithms for weakly coupled convex minimization problems. Applications to dynamical games and PDE's". In: *Journal of Convex Analysis* 15.3 (2008), p. 485.

[AY21] Uri Alon and Eran Yahav. "On the Bottleneck of Graph Neural Networks and its Practical Implications". In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=i80OPhOCVH2.

[Bak+22a] Justin Baker et al. “Learning POD of Complex Dynamics Using Heavy-ball Neural ODEs”. In: *arXiv preprint arXiv:2202.12373* (2022).

[Bak+22b] Justin Baker et al. “Proximal Implicit ODE Solvers for Accelerating Learning Neural ODEs”. In: *CoRR* abs/2204.08621 (2022). DOI: 10.48550/arXiv.2204.08621. arXiv: 2204.08621. URL: https://doi.org/10.48550/arXiv.2204.08621.

[Bap+20] Victor Bapst et al. “Unveiling the predictive power of static structure in glassy systems”. In: *Nature Physics* 16.4 (2020), pp. 448–454.

[Bat+16] Peter Battaglia et al. “Interaction Networks for Learning about Objects, Relations and Physics”. In: *Advances in Neural Information Processing Systems*. 2016, pp. 4502–4510.

[BC+11] Heinz H Bauschke, Patrick L Combettes, et al. *Convex analysis and monotone operator theory in Hilbert spaces*. Vol. 408. Springer, 2011.

[BCM06] Antoni Buades, Bartomeu Coll, and Jean M. Morel. “Neighborhood filters and PDE’s”. In: *Numerische Mathematik* 105.1 (2006), pp. 1–34.

[BCP95] Kathryn Eleda Brenan, Stephen L Campbell, and Linda Ruth Petzold. *Numerical solution of initial-value problems in differential-algebraic equations*. SIAM, 1995.

[Bit63] L. Bittner. “L. S. Pontryagin, V. G. Boltyanskii, R. V. Gamkrelidze, E. F. Mishechenko, The Mathematical Theory of Optimal Processes. VIII + 360 S. New York/London 1962. John Wiley & Sons. Preis 90/–”. In: *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik* 43.10-11 (1963), pp. 514–515. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/zamm.19630431023.

[BKK19] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. “Deep equilibrium models”. In: *Advances in Neural Information Processing Systems* 32 (2019).

[BN03] Mikhail Belkin and Partha Niyogi. "Laplacian Eigenmaps for Dimensionality Reduction and Data Representation". In: *Neural Computation* 15.6 (2003), pp. 1373–1396. DOI: 10.1162/089976603321780317.

[BN04] Mikhail Belkin and Partha Niyogi. "Semi-supervised learning on Riemannian manifolds". In: *Machine learning* 56.1-3 (2004), pp. 209–239.

[Bro+16] Greg Brockman et al. *OpenAI Gym*. cite arxiv:1606.01540. 2016. URL: http://arxiv.org/abs/1606.01540.

[Bro+21] Michael M Bronstein et al. "Geometric deep learning: Grids, groups, graphs, geodesics, and gauges". In: *arXiv preprint arXiv:2104.13478* (2021).

[BSF94] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166.

[BT09] Amir Beck and Marc Teboulle. "A fast iterative shrinkage-thresholding algorithm for linear inverse problems". In: *SIAM Journal on Imaging Sciences* 2.1 (2009), pp. 183–202.

[Cal+20] Jeff Calder et al. "Poisson Learning: Graph Based Semi-Supervised Learning At Very Low Label Rates". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 1306–1316. URL: https://proceedings.mlr.press/v119/calder20a.html.

[Cal18] Jeff Calder. "The game theoretic p-Laplacian and semi-supervised learning with few labels". In: *Nonlinearity* 32.1 (2018).

[CFG14] Tianqi Chen, Emily Fox, and Carlos Guestrin. "Stochastic gradient hamiltonian monte carlo". In: *International conference on machine learning*. 2014, pp. 1683–1691.

[CG97] Fan RK Chung and Fan Chung Graham. *Spectral graph theory*. 92. American Mathematical Soc., 1997.

[Cha+18] Pratik Chaudhari et al. "Deep relaxation: partial differential equations for optimizing deep neural networks". In: *Research in the Mathematical Sciences* 5.3 (2018), pp. 1–30.

[Cha+19] Ines Chami et al. "Hyperbolic graph convolutional neural networks". In: *Advances in neural information processing systems* 32 (2019), pp. 4868–4879.

[Cha+21a] Ben Chamberlain et al. "GRAND: Graph Neural Diffusion". In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, July 2021, pp. 1407–1418.

[Cha+21b] Benjamin Paul Chamberlain et al. "Beltrami Flow and Neural Diffusion on Graphs". In: *Proceedings of the Thirty-fifth Conference on Neural Information Processing Systems (NeurIPS) 2021, Virtual Event* (2021).

[Che+18] Ricky T. Q. Chen et al. "Neural Ordinary Differential Equations". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018.

[Che+20] Deli Chen et al. "Measuring and relieving the over-smoothing problem for graph neural networks from the topological view". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 2020, pp. 3438–3445.

[Cho+14] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[Cho+22] Suneghyeon Cho et al. "AdamNODEs: When Neural ODE Meets Adaptive Moment Estimation". In: *ArXiv* abs/2207.06066 (2022).

[Chu] F.R.K. Chung. *Spectral Graph Theory*. CBMS Regional Conference Series no. 92. Conference Board of the Mathematical Sciences. ISBN: 9780821889367. URL: https://books.google.com/books?id=YUc38%5C_MCuhAC.

[Coi+05] Ronald R Coifman et al. "Geometric diffusions as a tool for harmonic analysis and structure definition of data: Diffusion maps". In: *Proceedings of the National Academy of Sciences* 102.21 (2005), pp. 7426–7431. ISSN: 0027-8424. DOI: 10.1073/pnas.0500334102. eprint: https://www.pnas.org/content/102/21/7426.full.pdf. URL: https://www.pnas.org/content/102/21/7426.

[Dau+20] Talgat Daulbaev et al. "Interpolation Technique to Speed Up Gradients Propagation in Neural ODEs". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 16689–16700.

[DBV16] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. "Convolutional neural networks on graphs with fast localized spectral filtering". In: *Advances in neural information processing systems* 29 (2016), pp. 3844–3852.

[DD03] Paul D Dobson and Andrew J Doig. "Distinguishing enzyme structures from non-enzymes without alignments". In: *Journal of molecular biology*. Vol. 330. 4. Elsevier, 2003, pp. 771–783.

[DDT19] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. "Augmented neural odes". In: *Advances in Neural Information Processing Systems* 32 (2019).

[DEK20] Filipe De Avila Belbute-Peres, Thomas Economon, and Zico Kolter. "Combining Differentiable PDE Solvers and Graph Neural Networks for Fluid Flow Prediction". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 2402–2411. URL: https://proceedings.mlr.press/v119/de-avila-belbute-peres20a.html.

[DEL13] Xavier Desquesnes, Abderrahim Elmoataz, and Lézoray, Olivier. "Eikonal equation adaptation on weighted graphs: fast geometric diffusion process for local and non-local image and data processing". In: *Journal of Mathematical Imaging and Vision* 46.2 (2013), pp. 238–257.

[Dev+19] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Ed. by Jill Burstein, Christy Doran, and Thamar Solorio. Association for Computational Linguistics, 2019, pp. 4171–4186. DOI: 10.18653/v1/n19-1423. URL: https://doi.org/10.18653/v1/n19-1423.

[DF20] Qiang Du and Xiaobing Feng. "The phase field method for geometric moving interfaces and their numerical approximations". In: *Handbook of Numerical Analysis* 21 (2020), pp. 425–508.

[DFD20] Jianzhun Du, Joseph Futoma, and Finale Doshi-Velez. "Model-based Reinforcement Learning for Semi-Markov Decision Processes with Neural ODEs". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 19805–19816.

[DHS11] Mehdi Dehghan, Jalil Manafian Heris, and Abbas Saadatmandi. "Application of the Exp-function method for solving a partial differential equation arising in biology and population genetics". In: *International Journal of Numerical Methods for Heat & Fluid Flow* (2011).

[DP78] JR Dormand and PJ Prince. "New Runge-Kutta algorithms for numerical simulation in dynamical astronomy". In: *Celestial mechanics* 18.3 (1978), pp. 223–232.

[DP80a] J.R. Dormand and P.J. Prince. "A family of embedded Runge-Kutta formulae". In: *Journal of Computational and Applied Mathematics* 6.1 (1980), pp. 19–26. ISSN: 0377-0427.

[DP80b] John R Dormand and Peter J Prince. "A family of embedded Runge-Kutta formulae". In: *Journal of computational and applied mathematics* 6.1 (1980), pp. 19–26.

[DRF21] Sourav Dutta, Peter Rivera-Casillas, and Matthew W Farthing. "Neural Ordinary Differential Equations for Data-Driven Reduced Order Modeling of Environmental Hydrodynamics". In: *arXiv preprint arXiv:2104.13962* (2021).

[Duv+15] David K Duvenaud et al. "Convolutional Networks on Graphs for Learning Molecular Fingerprints". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes et al. Vol. 28. Curran Associates, Inc., 2015.

[EG96] Hairer Ernst and W Gerhard. *Solving ordinary differential equations II: Stiff and differentialalgebraic problems*. 1996.

[EHT21] Moshe Eliasof, Eldad Haber, and Eran Treister. "PDE-GCN: Novel Architectures for Graph Neural Networks Motivated by Partial Differential Equations". In: *arXiv preprint arXiv:2108.01938* (2021).

[ELB08] Abderrahim Elmoataz, Olivier Lezoray, and Sébastien Bougleux. "Nonlocal Discrete Regularization on Weighted Graphs: A Framework for Image and Manifold Processing". In: *IEEE Transactions on Image Processing* 17.7 (2008), pp. 1047–1060. DOI: 10.1109/TIP.2008.924284.

[Eri+21] N. Benjamin Erichson et al. "Lipschitz Recurrent Neural Networks". In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=-N7PBXqOUJZ.

[Fin+20] Chris Finlay et al. “How to Train Your Neural ODE: the World of Jacobian and Kinetic Regularization”. In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 3154–3164.

[For98] Robin Forman. “Combinatorial vector fields and dynamical systems”. In: *Mathematische Zeitschrift* 228.4 (1998), pp. 629–681.

[FR64] Reeves Fletcher and Colin M Reeves. “Function minimization by conjugate gradients”. In: *The computer journal* 7.2 (1964), pp. 149–154.

[FS00] Mark Freidlin and Shuenn-Jyi Sheu. “Diffusion processes on graphs: stochastic differential equations, large deviation principle”. In: *Probability theory and related fields* 116.2 (2000), pp. 181–220.

[FS21] Francesco Farina and Emma Slade. “Data efficiency in graph networks through equivariance”. In: *arXiv preprint arXiv:2106.13786* (2021).

[FW93] Mark I. Freidlin and Alexander D. Wentzell. “Diffusion Processes on Graphs and the Averaging Principle”. In: *The Annals of Probability* 21.4 (1993), pp. 2215–2245. DOI: 10.1214/aop/1176989018. URL: https://doi.org/10.1214/aop/1176989018.

[Gar+14] Cristina Garcia-Cardona et al. “Multiclass Data Segmentation Using Diffuse Interface Methods on Graphs”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.8 (2014), pp. 1600–1613. DOI: 10.1109/TPAMI.2014.2300478.

[GB10] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *AISTATS*. 2010.

[Ghi+81] Michael Ghil et al. “Applications of estimation theory to numerical weather prediction”. In: *Dynamic meteorology: Data assimilation methods*. Springer, 1981, pp. 139–224.

[Gho+20] Arnab Ghosh et al. “STEER : Simple Temporal Regularization For Neural ODE”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 14831–14843.

[Gil+17] Justin Gilmer et al. “Neural Message Passing for Quantum Chemistry”. In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, Aug. 2017, pp. 1263–1272. URL: https://proceedings.mlr.press/v70/gilmer17a.html.

[GKB19] Amir Gholami, Kurt Keutzer, and George Biros. “Anode: Unconditionally accurate memory-efficient gradients for neural odes”. In: *arXiv preprint arXiv:1902.10298* (2019).

[GO08] Guy Gilboa and Stanley Osher. “Nonlocal operators with applications to image processing”. In: *Multiscale Modeling & Simulation* 7.3 (2008), pp. 1005–1028.

[Gon71] Amilcar dos Santos Gonçalves. “A Version of Beale’s Method Avoiding the Free-Variables”. In: *Proceedings of the 1971 26th Annual Conference*. ACM ’71. New York, NY, USA: Association for Computing Machinery, 1971, pp. 433–441. ISBN: 9781450374842. DOI: 10.1145/800184.810512. URL: https://doi.org/10.1145/800184.810512.

[Gra+18] Will Grathwohl et al. “Ffjord: Free-form continuous dynamics for scalable reversible generative models”. In: *arXiv preprint arXiv:1810.01367* (2018).

[Gra+19] Will Grathwohl et al. “Scalable Reversible Generative Models with Free-form Continuous Dynamics”. In: *International Conference on Learning Representations*. 2019.

[GRK10] V Gontis, J Ruseckas, and A Kononovičius. “A long-range memory stochastic model of the return in financial markets”. In: *Physica A: Statistical Mechanics and its Applications* 389.1 (2010), pp. 100–106.

[He+15] Kaiming He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[He+16a] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 770–778.

[He+16b] Kaiming He et al. “Identity mappings in deep residual networks”. In: *European Conference on Computer Vision*. 2016, pp. 630–645.

[Hil13] David Hilditch. “An introduction to well-posedness and free-evolution”. In: *International Journal of Modern Physics A* 28.22n23 (2013), p. 1340015.

[Hoc+01] Sepp Hochreiter et al. “Gradient flow in recurrent nets: the difficulty of learning long-term dependencies”. In: *A field guide to dynamical recurrent neural networks*. IEEE Press, 2001.

[Hoc91] Sepp Hochreiter. “Untersuchungen zu dynamischen neuronalen Netzen”. In: *Diploma, Technische Universität München* 91.1 (1991).

[HR17] Eldad Haber and Lars Ruthotto. “Stable architectures for deep neural networks”. In: *Inverse Problems* 34.1 (Dec. 2017), p. 014004. DOI: 10.1088/1361-6420/aa9a90. URL: https://doi.org/10.1088/1361-6420/aa9a90.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[HSW20] Zijie Huang, Yizhou Sun, and Wei Wang. “Learning Continuous System Dynamics from Irregularly-Sampled Partial Observations”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 16177–16187. URL: https://proceedings.neurips.cc/paper/2020/file/ba4849411c8bbdd386150e5e32204198-Paper.pdf.

[HWY18] Kyle Helfrich, Devin Willmott, and Qiang Ye. "Orthogonal Recurrent Neural Networks with Scaled Cayley Transform". In: *Proceedings of the 35th International Conference on Machine Learning*. 2018, pp. 1969–1978.

[HYL17] Will Hamilton, Zhitao Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017.

[IS15] S. Ioffe and C. Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *Proceedings of the 32nd International Conference on Machine Learning*. 2015, pp. 448–456.

[Ise09] Arieh Iserles. *A first course in the numerical analysis of differential equations*. 44. Cambridge university press, 2009.

[JB19] Junteng Jia and Austin R Benson. "Neural jump stochastic differential equations". In: *Advances in Neural Information Processing Systems* 32 (2019).

[Jia+20] Chiyu Jiang et al. "ShapeFlow: Learnable Deformations Among 3D Shapes". In: *Advances in Neural Information Processing Systems*. 2020.

[Jin+17] Li Jing et al. "Tunable efficient unitary neural networks (EUNN) and their application to RNNs". In: *Proceedings of the 34th International Conference on Machine Learning*. 2017, pp. 1733–1741.

[KB15] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *Proceedings of the 3rd International Conference on Learning Representations*. 2015.

[Kel+20] Jacob Kelly et al. "Learning Differential Equations that are Easy to Solve". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 4370–4380.

[Kid+20] Patrick Kidger et al. “Neural controlled differential equations for irregular time series”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 6696–6707.

[Kim+21] Suyong Kim et al. “Stiff neural ordinary differential equations”. In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 31.9 (2021), p. 093122.

[KSH17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[KW17] Thomas N. Kipf and Max Welling. “Semi-Supervised Classification with Graph Convolutional Networks”. In: *Proceedings of the 5th International Conference on Learning Representations* (ICLR). ICLR ’17. Palais des Congrès Neptune, Toulon, France, 2017. URL: https://openreview.net/forum?id=SJU4ayYgl.

[KWG19] Johannes Klicpera, Stefan Weißenberger, and Stephan Günnemann. “Diffusion improves graph learning”. In: *Advances in Neural Information Processing Systems* 32 (2019), pp. 13354–13366.

[LC10] Yann LeCun and Corinna Cortes. “MNIST handwritten digit database”. In: *ATT Labs [Online]* (2010). URL: http://yann.lecun.com/exdb/mnist/.

[LEL14] François Lozes, Abderrahim Elmoataz, and Olivier Lézoray. “Partial Difference Operators on Weighted Graphs for Image Processing on Surfaces and Point Clouds”. In: *IEEE Transactions on Image Processing* 23.9 (2014), pp. 3896–3909. DOI: 10.1109/TIP.2014.2336548.

[LH20] Mathias Lechner and Ramin Hasani. “Learning Long-Term Dependencies in Irregularly-Sampled Time Series”. In: *arXiv preprint arXiv:2006.04418* (2020).

[LHE22] Stephanie Lin, Jacob Hilton, and Owain Evans. “Teaching Models to Express Their Uncertainty in Words”. In: *Transactions on Machine Learning Research* (2022). URL: https://openreview.net/forum?id=8s8K2UZGTZ.

[LHW18] Qimai Li, Zhichao Han, and Xiao-Ming Wu. "Deeper insights into graph convolutional networks for semi-supervised learning". In: *Thirty-Second AAAI conference on artificial intelligence*. 2018.

[Li+18] Huan Li et al. "Optimization algorithm inspired deep neural network structure design". In: *arXiv preprint arXiv:1810.01638* (2018).

[Li+19a] Guohao Li et al. "Deepgcns: Can gcns go as deep as cnns?" In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 9267–9276.

[Li+19b] Yunzhu Li et al. "Learning Particle Dynamics for Manipulating Rigid Bodies, Deformable Objects, and Fluids". In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=rJgbSn09Ym.

[Li+20a] Guohao Li et al. "Deepergcn: All you need to train deeper gcns". In: *arXiv preprint arXiv:2006.07739* (2020).

[Li+20b] Xuechen Li et al. "Scalable Gradients for Stochastic Differential Equations". In: *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*. Ed. by Silvia Chiappa and Roberto Calandra. Vol. 108. Proceedings of Machine Learning Research. PMLR, Aug. 2020, pp. 3870–3882. URL: http://proceedings.mlr.press/v108/li20i.html.

[Li+21] Guohao Li et al. "Training Graph Neural Networks with 1000 Layers". In: *arXiv preprint arXiv:2106.07476* (2021).

[Lia+19] Renjie Liao et al. "LanczosNet: Multi-Scale Deep Graph Convolutional Networks". In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=BkedznAqKQ.

[Liu+19] Xuanqing Liu et al. "Neural sde: Stabilizing neural ode networks with stochastic noise". In: *arXiv preprint arXiv:1906.02355* (2019).

[LN89] Dong C Liu and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization". In: *Mathematical programming* 45.1 (1989), pp. 503–528.

[LNK19] Qi Liu, Maximilian Nickel, and Douwe Kiela. "Hyperbolic Graph Neural Networks". In: *Advances in Neural Information Processing Systems* 32 (2019), pp. 8230–8241.

[LP21] Kookjin Lee and Eric J Parish. "Parameterized neural ordinary differential equations: Applications to computational physics problems". In: *Proceedings of the Royal Society A* 477.2253 (2021), p. 20210162.

[Lu+18] Yiping Lu et al. "Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations". In: *International Conference on Machine Learning*. 2018, pp. 3276–3285.

[Mas+20] Stefano Massaroli et al. "Dissecting Neural ODEs". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 3952–3963.

[MB17] Thomas Moreau and Joan Bruna. "Understanding the learned iterative soft thresholding algorithm with matrix factorization". In: *arXiv preprint arXiv:1706.01338* (2017).

[MBB17] Federico Monti, Michael M Bronstein, and Xavier Bresson. "Geometric matrix completion with recurrent multi-graph neural networks". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 2017, pp. 3700–3710.

[Mha+17] Zakaria Mhammedi et al. "Efficient orthogonal parametrisation of recurrent neural networks using householder reflections". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. 2017, pp. 2401–2409.

[MKB13] Ekaterina Merkurjev, Tijana Kostic, and Andrea L Bertozzi. "An MBO scheme on graphs for classification and image processing". In: *SIAM Journal on Imaging Sciences* 6.4 (2013), pp. 1903–1930.

[MMY21] Takashi Matsubara, Yuto Miyatake, and Takaharu Yaguchi. "Symplectic Adjoint Method for Exact Gradient of Neural ODE with Minimal Memory". In: *Advances in Neural Information Processing Systems*. Ed. by A. Beygelzimer et al. 2021.

[Moh+91] Bojan Mohar et al. "The Laplacian spectrum of graphs". In: *Graph theory, combinatorics, and applications* 2.871-898 (1991), p. 12.

[Mon+17] Federico Monti et al. "Geometric Deep Learning on Graphs and Manifolds Using Mixture Model CNNs". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2017, pp. 5425–5434. DOI: 10.1109/CVPR.2017.576. URL: https://doi.ieeecomputersociety.org/10.1109/CVPR.2017.576.

[MP19] Daniel Matthes and Simon Plazotta. "A variational formulation of the BDF2 method for metric gradient flows". In: *ESAIM: Mathematical Modelling and Numerical Analysis* 53.1 (2019), pp. 145–172.

[MS12] Goran Marjanovic and Victor Solo. "On L_q optimization and matrix completion". In: *IEEE Transactions on signal processing* 60.11 (2012), pp. 5714–5724.

[Nam+12] Galileo Namata et al. "Query-driven active surveying for collective classification". In: *10th international workshop on mining and learning with graphs*. Vol. 8. 2012, p. 1.

[Nes83] Yurii E Nesterov. "A method for solving the convex programming problem with convergence rate $O(1/k^2)$". In: *Proceedings of the Russian Academy of Sciences* 269 (1983), pp. 543–547.

[Ngu+20] Tan Nguyen et al. "MomentumRNN: Integrating Momentum into Recurrent Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1924–1936.

[Ngu+22] Nghia Nguyen et al. “Improving Neural Ordinary Differential Equations with Nesterov’s Accelerated Gradient Method”. In: *Advances in Neural Information Processing Systems*. Ed. by Alice H. Oh et al. 2022. URL: https://openreview.net/forum?id=-OfK_B9Q5hI.

[NM19] Hoang Nt and Takanori Maehara. “Revisiting graph neural networks: All we have is low-pass filters”. In: *arXiv preprint arXiv:1905.09550* (2019).

[Nor+20] Alexander Norcliffe et al. “On Second Order Behaviour in Augmented Neural ODEs”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 5911–5921.

[Nor+21] Alexander Norcliffe et al. “Neural {ODE} Processes”. In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=27acGyyI1BY.

[NS17] Jean-Philippe Noël and M Schoukens. “F-16 aircraft benchmark based on ground vibration test data”. In: *2017 Workshop on Nonlinear System Identification Benchmarks*. 2017, pp. 19–23.

[OS20] Kenta Oono and Taiji Suzuki. “Graph Neural Networks Exponentially Lose Expressive Power for Node Classification”. In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=S1ldO2EFPr.

[PAC18] Ivens Portugal, Paulo Alencar, and Donald Cowan. “The use of machine learning algorithms in recommender systems: A systematic review”. In: *Expert Systems with Applications* 97 (2018), pp. 205–227.

[Pal+20] Aditya Pal et al. “Pinnersage: Multi-modal user embedding framework for recommendations at pinterest”. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2020, pp. 2311–2320.

[Pal+21] Avik Pal et al. "Opening the Blackbox: Accelerating Neural Differential Equations by Regularizing Internal Solver Heuristics". In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, July 2021, pp. 8325–8335.

[Pas+17] Adam Paszke et al. "Automatic differentiation in pytorch". In: (2017).

[PAS17] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. "Predicting multivariate time series of interdependent metrics at scale: a graph attention based approach". In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM. 2017, pp. 555–564.

[PB14] Neal Parikh and Stephen Boyd. "Proximal algorithms". In: *Foundations and Trends in optimization* 1.3 (2014), pp. 127–239.

[PC+19] Gabriel Peyré, Marco Cuturi, et al. "Computational optimal transport: With applications to data science". In: *Foundations and Trends® in Machine Learning* 11.5-6 (2019), pp. 355–607.

[Pfa+21] Tobias Pfaff et al. "Learning Mesh-Based Simulation with Graph Networks". In: *International Conference on Learning Representations*. 2021. URL: `https://openreview.net/forum?id=roNqYL0_XP`.

[PM90] Pietro Perona and Jitendra Malik. "Scale-space and edge detection using anisotropic diffusion". In: *IEEE Transactions on pattern analysis and machine intelligence* 12.7 (1990), pp. 629–639.

[PMB13] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks". In: *International Conference on Machine Learning*. 2013, pp. 1310–1318.

[Pol+19] Michael Poli et al. "Graph neural ordinary differential equations". In: *arXiv preprint arXiv:1911.07532* (2019).

[Pol+20] Michael Poli et al. “Hypersolvers: Toward Fast Continuous-Depth Models”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 21105–21117.

[Pol64] Boris T Polyak. “Some methods of speeding up the convergence of iteration methods”. In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17.

[PP10] K Parand and A Pirkhedri. “Sinc-collocation method for solving astrophysics equations”. In: *New Astronomy* 15.6 (2010), pp. 533–537.

[PSW15] Nicholas G Polson, James G Scott, and Brandon T Willard. “Proximal algorithms in statistics and machine learning”. In: *Statistical Science* 30.4 (2015), pp. 559–581.

[PT92] William H Press and Saul A Teukolsky. “Adaptive Stepsize Runge-Kutta Integration”. In: *Computers in Physics* 6.2 (1992), pp. 188–191.

[Qiu+18] Jiezhong Qiu et al. “Deepinf: Social influence prediction with deep learning”. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2018, pp. 2110–2119.

[Qua+20] Alessio Quaglino et al. “SNODE: Spectral Discretization of Neural ODEs for System Identification”. In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=Sye0XkBKvS.

[RCD19] Yulia Rubanova, Ricky T. Q. Chen, and David K Duvenaud. “Latent Ordinary Differential Equations for Irregularly-Sampled Time Series”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019.

[RCL19] Mauricio Flores Rios, Jeff Calder, and Gilad Lerman. “Algorithms for lp-based semi-supervised learning on graphs”. In: *arXiv preprint arXiv:1901.05031* (2019).

[Rd17] Vincent Roulet and Alexandre d'Aspremont. "Sharpness, restart and acceleration". In: *Advances in Neural Information Processing Systems*. 2017, pp. 1119–1129.

[RG15] Chang-han Rhee and Peter W Glynn. "Unbiased estimation with square root convergence for SDE models". In: *Operations Research* 63.5 (2015), pp. 1026–1043.

[RK95] Carl Runge and Martin Kutta. "On the numerical solution of differential equations". In: *Mathematische Annalen* 46.2 (1895), pp. 167–178.

[Ron+20] Yu Rong et al. "DropEdge: Towards Deep Graph Convolutional Networks on Node Classification". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=Hkx1qkrKPr.

[Ros60] H. H. Rosenbrock. "An Automatic Method for Finding the Greatest or Least Value of a Function". In: *The Computer Journal* 3.3 (Jan. 1960), pp. 175–184. ISSN: 0010-4620. DOI: 10.1093/comjnl/3.3.175. URL: https://doi.org/10.1093/comjnl/3.3.175.

[RS00] Sam T Roweis and Lawrence K Saul. "Nonlinear dimensionality reduction by locally linear embedding". In: *science* 290.5500 (2000), pp. 2323–2326.

[Rus+15] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[San+20] Alvaro Sanchez-Gonzalez et al. "Learning to Simulate Complex Physics with Graph Networks". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 8459–8468. URL: https://proceedings.mlr.press/v119/sanchez-gonzalez20a.html.

[San+21] Michael E. Sander et al. “Momentum Residual Neural Networks”. In: *arXiv preprint arXiv:2102.07870* (2021). arXiv: 2102.07870 [cs.LG].

[SBC14] Weijie Su, Stephen Boyd, and Emmanuel Candes. “A differential equation for modeling Nesterov’s accelerated gradient method: Theory and insights”. In: *Advances in Neural Information Processing Systems*. 2014, pp. 2510–2518.

[Sha+22] Rulin Shao et al. “On the Adversarial Robustness of Vision Transformers”. In: *Transactions on Machine Learning Research* (2022). URL: https://openreview.net/forum?id=lE7K4n1Esk.

[Shc+18] Oleksandr Shchur et al. “Pitfalls of graph neural network evaluation”. In: *arXiv preprint arXiv:1811.05868* (2018).

[Shi+19] Bin Shi et al. “Acceleration via Symplectic Discretization of High-Resolution Differential Equations”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019.

[SKL20] Adil Salim, Anna Korba, and Giulia Luise. “The Wasserstein Proximal Gradient Algorithm”. In: *arXiv preprint arXiv:2002.03035* (2020).

[SM03] Endre Süli and David F Mayers. *An introduction to numerical analysis*. Cambridge university press, 2003.

[Son+20] Yang Song et al. “Score-based generative modeling through stochastic differential equations”. In: *arXiv preprint arXiv:2011.13456* (2020).

[SOZ17] Zuoqiang Shi, Stanley J. Osher, and Wei. Zhu. “Weighted nonlocal Laplacian on interpolation from sparse data”. In: *Journal of Scientific Computing* 73.2-3 (2017), pp. 1164–1177.

[Sut+13] Ilya Sutskever et al. “On the importance of initialization and momentum in deep learning”. In: *International Conference on Machine Learning*. 2013, pp. 1139–1147.

[SXY19] Jie Shen, Jie Xu, and Jiang Yang. "A new class of efficient and robust energy stable schemes for gradient flows". In: *SIAM Review* 61.3 (2019), pp. 474–506.

[SZ15] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." In: *ICLR*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://dblp.uni-trier.de/db/conf/iclr/iclr2015.html#SimonyanZ14a.

[Sze+15] Christian Szegedy et al. "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[Sze+16] Christian Szegedy et al. "Rethinking the inception architecture for computer vision". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2818–2826.

[TET12] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[Tho+22] Matthew Thorpe et al. "GRAND++: Graph Neural Diffusion with A Source Term". In: *International Conference on Learning Representations*. 2022. URL: https://openreview.net/forum?id=EMxu-dzvJk.

[TR19] Belinda Tzen and Maxim Raginsky. "Neural stochastic differential equations: Deep latent gaussian models in the diffusion limit". In: *arXiv preprint* (2019).

[TT13] Nizar Touzi and Agnès Tourin. *Optimal stochastic control, stochastic target problems, and backward SDE*. Vol. 29. Springer, 2013.

[Umm+20] Benjamin Ummenhofer et al. "Lagrangian Fluid Simulation with Continuous Convolutions". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=B1lDoJSYDH.

[Vas+17] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[Vel+] Petar Veličković et al. "Graph Attention Networks". In: *International Conference on Learning Representations*.

[Vor+17] Eugene Vorontsov et al. "On orthogonality and learning recurrent networks with long term dependencies". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. 2017, pp. 3570–3578.

[Wan+06] Fei Wang et al. "Semi-supervised classification using linear neighborhood propagation". In: *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*. Vol. 1. IEEE. 2006, pp. 160–167.

[Wan+19] Bao Wang et al. "ResNets Ensemble via the Feynman-Kac Formalism to Improve Natural and Robust Accuracies". In: *Advances in Neural Information Processing Systems*. 2019, pp. 1655–1665.

[Wan+20] Bao Wang et al. "Scheduled Restart Momentum for Accelerated Stochastic Gradient Descent". In: *arXiv preprint arXiv:2002.10583* (2020).

[Wan+21a] Luyu Wang et al. "WikiGraphs: A Wikipedia text-knowledge graph paired dataset". In: *arXiv preprint arXiv:2107.09556* (2021).

[Wan+21b] Yifei Wang et al. "Dissecting the Diffusion Process in Linear Graph Convolutional Networks". In: *arXiv preprint arXiv:2102.10739* (2021).

[Wan+21c] Yiwei Wang et al. "Particle-based energetic variational inference". In: *Statistics and Computing* 31.3 (2021), pp. 1–17.

[Wan+22] Jianfeng Wang et al. "GIT: A Generative Image-to-text Transformer for Vision and Language". In: *Transactions of Machine Learning Research* (2022). URL: https://openreview.net/forum?id=b4tMhpN0JC.

[Wis+16] Scott Wisdom et al. “Full-capacity unitary recurrent neural networks”. In: *Advances in Neural Information Processing Systems*. 2016, pp. 4880–4888.

[WN11] Homer F Walker and Peng Ni. “Anderson acceleration for fixed-point iterations”. In: *SIAM Journal on Numerical Analysis* 49.4 (2011), pp. 1715–1735.

[WRJ18] Ashia C. Wilson, Benjamin Recht, and Michael I. Jordan. “A Lyapunov Analysis of Momentum Methods in Optimization”. In: *arXiv preprint arXiv:1611.02635* (2018). arXiv: `1611.02635 [math.OC]`.

[WY20] Bao Wang and Qiang Ye. “Stochastic Gradient Descent with Nonlinear Conjugate Gradient-Style Adaptive Momentum”. In: *arXiv preprint arXiv:2012.02188* (2020).

[XGC20] Mingtao Xia, Chris D. Greenman, and Tom Chou. “PDE Models of Adder Mechanisms in Cellular Proliferation”. In: *SIAM Journal on Applied Mathematics* 80.3 (2020), pp. 1307–1335. DOI: `10.1137/19M1246754`.

[Xia+21] Hedi Xia et al. “Heavy ball neural ordinary differential equations”. In: *Advances in Neural Information Processing Systems* 34 (2021).

[XQT20] Louis-Pascal Xhonneux, Meng Qu, and Jian Tang. “Continuous Graph Neural Networks”. In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 10432–10441.

[Xu+18] Keyulu Xu et al. “Representation learning on graphs with jumping knowledge networks”. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 5453–5462.

[Xu+19] Jinchao Xu et al. “On the stability and accuracy of partially and fully implicit schemes for phase field modeling”. In: *Computer Methods in Applied Mechanics and Engineering* 345 (2019), pp. 826–853.

[Yan+19] Guandao Yang et al. “PointFlow: 3D Point Cloud Generation with Continuous Normalizing Flows”. In: *arXiv* (2019).

[YHL19] Cagatay Yildiz, Markus Heinonen, and Harri Lahdesmaki. “ODE2VAE: Deep generative second order ODEs with Bayesian neural networks”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/99a401435dcb65c4008d3ad22c8cdad0-Paper.pdf.

[Yin+18a] Penghang Yin et al. “Stochastic backward Euler: an implicit gradient descent algorithm for k-means clustering”. In: *Journal of Scientific Computing* 77.2 (2018), pp. 1133–1146.

[Yin+18b] Rex Ying et al. “Graph convolutional neural networks for web-scale recommender systems”. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2018, pp. 974–983.

[YML19] Liang Yao, Chengsheng Mao, and Yuan Luo. “Graph convolutional networks for text classification”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 33. 01. 2019, pp. 7370–7377.

[ZA20] Lingxiao Zhao and Leman Akoglu. “PairNorm: Tackling Oversmoothing in GNNs”. In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=rkecl1rtwB.

[ZC18] Muhan Zhang and Yixin Chen. “Link prediction based on graph neural networks”. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 2018, pp. 5171–5181.

[ZDC20] Yaofeng Desmond Zhong, Biswadip Dey, and Amit Chakraborty. “Symplectic ODE-Net: Learning Hamiltonian Dynamics with Control”. In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=ryxmb1rKDS.

[ZEG20] Alexander Zaitzeff, Selim Esedoglu, and Krishna Garikipati. "Variational Extrapolation of Implicit Schemes for General Gradient Flows". In: *SIAM Journal on Numerical Analysis* 58.5 (2020), pp. 2799–2817.

[ZF13] Matthew D. Zeiler and Rob Fergus. "Visualizing and Understanding Convolutional Networks". In: *European Conference on Computer Vision*. 2013.

[ZGL03] Xiaojin Zhu, Zoubin Ghahramani, and John D. Lafferty. "Semi-Supervised Learning Using Gaussian Fields and Harmonic Functions". In: *International Conference on Machine Learning*. 2003.

[Zha+19a] Si Zhang et al. "Graph convolutional networks: a comprehensive review". In: *Computational Social Networks* 6.1 (2019), pp. 1–23.

[Zha+19b] Tianjun Zhang et al. "ANODEV2: A Coupled Neural ODE Framework". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/227f6afd3b7f89b96c4bb91f95d50f6d-Paper.pdf.

[Zho+04] Dengyong Zhou et al. "Learning with local and global consistency". In: *Advances in neural information processing systems*. 2004, pp. 321–328.

[Zhu+20a] Juntang Zhuang et al. "Adaptive Checkpoint Adjoint Method for Gradient Estimation in Neural ODE". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 11639–11649.

[Zhu+20b] Juntang Zhuang et al. "Ordinary differential equations on graph networks". In: https://openreview.net/forum?id=SJg9z6VFDr, 2020.

[Zhu+21] Juntang Zhuang et al. "MALI: A memory efficient and reverse accurate integrator for Neural ODEs". In: *arXiv preprint arXiv:2102.04668* (2021).

[ZS05] Dengyong Zhou and Bernhard Schölkopf. "Regularization on Discrete Spaces". In: *27th DAGM Conference on Pattern Recognition*. 2005, pp. 361–368.

[ZYX17] S. Zhang, P. Yin, and J. Xin. “Transformed Schatten-1 Iterative Thresholding Algorithms for Low Rank Matrix Completion”. In: *Comm. Math Sci* 15.3 (2017), pp. 839–862.