Modeling Intelligence via Graph Neural Networks

by

Keyulu Xu

B.Sc., University of British Columbia (2016)
S.M., Massachusetts Institute of Technology (2019)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2021

Certified by . . . .
Stefanie Jegelka
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

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

Modeling Intelligence via Graph Neural Networks

by

Keyulu Xu

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Artificial intelligence can be more powerful than human intelligence. Many problems are perhaps challenging from a human perspective. These could be seeking statistical patterns in complex and structured objects, such as drug molecules and the global financial system. Advances in deep learning have shown that the key to solving such tasks is to learn a good representation. Given the representations of the world, the second aspect of intelligence is reasoning. Learning to reason implies learning to implement a correct reasoning process, within and outside the training distribution.

In this thesis, we address the fundamental problem of modeling intelligence that can learn to represent and reason about the world. We study both questions from the lens of graph neural networks, a class of neural networks acting on graphs. First, we can abstract many objects in the world as graphs and learn their representations with graph neural networks. Second, we shall see how graph neural networks exploit the algorithmic structure in reasoning processes to improve generalization.

This thesis consists of four parts. Each part also studies one aspect of the theoretical landscape of learning: the representation power, generalization, extrapolation, and optimization. In Part I, we characterize the expressive power of graph neural networks for representing graphs, and build maximally powerful graph neural networks. In Part II, we analyze generalization and show implications for what reasoning a neural network can sample-efficiently learn. Our analysis takes into account the training algorithm, the network structure, and the task structure. In Part III, we study how neural networks extrapolate and under what conditions they learn the correct reasoning outside the training distribution. In Part IV, we prove global convergence rates and develop normalization methods that accelerate the training of graph neural networks. Our techniques and insights go beyond graph neural networks, and extend broadly to deep learning models.

Thesis Supervisor: Stefanie Jegelka
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

This section is devoted to my advisors, committee members, co-authors, and almost co-authors. Their wonderful work and insightful discussions are the keys to shaping this thesis.

Let us start with Stefanie Jegelka, my advisor at MIT. She has granted much intellectual freedom, which is arguably the real value of a PhD, but has also managed to counterbalance that with valuable, precise advice and insight. These all helped shape the thesis. Working with her has truly been a joy, and I am grateful to have had the opportunity.

I am fortunate to have Tommi Jaakkola and Antonio Torralba as my committee members. This thesis is incomplete without their invaluable advice, ranging from broader viewpoints on modeling intelligence, to the technical aspects of statistics, learning, and implications in real applications.

Next, let us appreciate the work with Ken-ichi Kawarabayashi, my host at NII and the ERATO Kawarabayashi Large Graph Project. Despite being a theorist, he prefers instead seeking the truth and making a big impact on both real business and general intelligence. Such ambition and optimism have inspired me.

We shall appreciate the time with Nick Harvey, my undergraduate advisor at UBC. His enthusiasm and elegant, theoretical thinking have inspired me to explore the world with new, my own perspectives, while complementing these with mathematical guarantees.

It is now the time for our amazing co-authors. The time spent together with them has really defined this PhD journey and thesis. In an alphabetical order, let us thank the wonderful peers and friends: Tianle Cai, Simon S. Du, Di He, Weihua Hu, Kangcheng Hou, Kenji Kawaguchi, Alex Peiyuan Liao, Jingling Li, Chengtao Li, Shengjie Luo, David Alvarez-Melis, Tomohiro Sonobe, Yonglong Tian, Ruosong Wang, Mozhi Zhang, and Han Zhao. Let our success continue.

Importantly, we shall appreciate working with the senior co-authors, Jure Leskovec, Ruslan Salakhutdinov, and Suvrit Sra. Their valuable advice and support are indispensable for the thesis works, and the conversations with them have exposed me to very different perspectives and expertise.

Last but not least, during the PhD journey, I have had insightful research conversations with other friends and senior researchers. Although these ideas and discussions eventually did not turn into formal publications, they have implicitly influenced my research. In particular, we thank Yuichi Yoshida for his discussions on deep learning theory, mathematically understanding intelligence, and graph theory. We thank Matt J. Staib for conversations on a broad range of topics such as kernels and optimization. We thank Takuya Konishi for his constant feedback and suggestions on my research. We thank Zi Wang for her discussion on formulating the next generation of AI. We thank Jiajun Wu for discussing building machines that can learn to reason. We thank all the members of Jegelka group for broadly exchanging research ideas. We also thank a part of the members of LOGSS for exchanging research ideas. We thank my mentors and colleagues at Two Sigma Investments for discussing quantitative trading. We thank my mentors and peers at Hudson River Trading for sharing ideas on high-frequency trading. We thank Yasuo Tabei for discussing the business applications of deep learning. We thank Makoto Yamada for the discussions on high-dimensional statistics and graph neural networks. We thank Ryoma Sato for the discussions on graph neural networks. We thank Gang Niu and Nan Lu for research discussions on noisy labels and learning with weak supervision. We thank David Lin for conversations on theoretical physics and large-scale experimental physics. We thank Tuan Anh Le for the discussions on neural program induction. We thank Kohei Hayashi for discussions on tensor methods. To keep the formal nature of the thesis, here, we only include research discussions with formal settings. Hence, we keep the list of friends and colleagues who had more casual discussions a secret. By a similar argument, co-authors of ongoing projects also do not appear here.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Artificial intelligence can be more powerful than human intelligence. This thesis is about modeling intelligence that can learn to represent and reason about the world.

Representation. The world consists of objects, from a quark to a planet. A core part of intelligence is being able to capture and then predict certain properties of these objects. While humans are good at inferring from visual hues, their innate power is perhaps more limited when they have to make sense of other complex and structured objects. For example, it is challenging to tell whether a drug molecule is lethal or effective without having seen it before. As another example, experienced human traders may well understand the macro, but they inevitably miss the high-frequency and statistical hues of the financial market.

The advances of modern deep learning have shown that an effective approach to the questions above is to learn a good representation [LeCun et al., 2015]. That is, we train a neural network $f : \mathcal{X} \to \mathbb{R}^d$ to map an object defined over $\mathcal{X}$ to the vector space $\mathbb{R}^d$, with which we can apply a linear transform to make the prediction[1]. Hence, we hope to build neural networks that can almost universally act on the objects in the world. In other words, we define $\mathcal{X}$ to be general enough to abstract many objects, and design neural networks f that can effectively act on and learn representations in $\mathcal{X}$. In fact, $\mathcal{X} = \mathbb{R}^{m \times n}$ was sufficient

[1]The linear transform is often a part of the neural network, learned end to end.

for some domains such as images. A more general choice of $\mathcal{X}$ is the space of *graphs*. We can define a graph $G = (V, E)$ by its graph structure (V, E), node features $\boldsymbol{h}_v \in \mathbb{R}^{m_1}$ for $v \in V$, and edge features $\boldsymbol{h}_{(u,v)} \in \mathbb{R}^{m_2}$ for $(u, v) \in E$. One can easily see that the space of graphs generalizes the image space $\mathbb{R}^{m \times n}$ by encoding the positional information in the node features. Indeed, many objects are either explicitly or implicitly represented by graphs, for instance, molecules, the global financial system, and 3D objects as meshes or point clouds. Hence, we consider the space of graphs and study representation from the lens of graph neural networks (GNNs), a class of neural networks acting on graphs [Gori et al., 2005, Scarselli et al., 2009].

Effective representation requires powerful and expressive models that can capture fine-grained information of the graphs. In the first part of the thesis, we address the core question regarding representation learning of graphs:

How can we build powerful graph neural networks?

We theoretically analyze the learned representation and expressive power of GNNs, and apply our theory to build maximally powerful GNNs. Specifically, we solve two problems. First, surprisingly, deeper GNNs often underperform shallow GNNs in node prediction tasks. We give a theoretical explanation for this puzzle and propose more powerful and flexible models to overcome such issues. Second, we ask how powerful GNNs are for distinguishing non-isomorphic graphs, i.e., solving the graph isomorphism problem. We introduce a framework to characterize the discriminative power of GNNs and design a simple yet powerful GNN architecture.

Reasoning. The second core part of intelligence is reasoning. Given the representations of the objects in the world, we are curious about how they relate, interact, and evolve. For example, one may be asked to reason about and predict how a set of physical objects would evolve in the next few seconds [Wu et al., 2017]. One may also be interested in learning to solve puzzles or mathematical equations after having been exposed to some examples [Santoro et al., 2018]. To reach an answer to questions like these, we need a corresponding reasoning process. We may understand a reasoning process as an algorithm $g : \mathcal{X} \to \mathbb{R}^d$ where $\mathcal{X}$ is the space of a collection or a graph of objects, for example, those used to solve a mathematical equation. Hence, learning to reason implies learning to implement a correct reasoning algorithm.

While many neural architectures are able to represent these complex, structured reasoning processes by the universal approximation properties [Hornik et al., 1989], the solutions they find via stochastic gradient descent often do not generalize well to unseen situations. The successful cases often possess specific network structure. For example, GNNs acting on a graph of the objects have shown to succeed in a broad range of reasoning problems, such as visual question answering, intuitive physics, and mathematical reasoning. The graph that the GNN acts on may be constructed based on prior knowledge or may simply be a complete graph. Theoretically, there is still limited understanding of what affects the generalization of neural networks, what reasoning they can learn, and what are the limitations. This is also crucial for modeling network architectures for new reasoning tasks. We ask the following fundamental question:

What can neural networks reason about?

To answer the question above, we must understand the *generalization* properties of neural networks, in particular structured networks like GNNs, for both the interpolation and extrapolation regimes. In the interpolation regime, we assume the training distribution $\mathcal{P}_{\text{train}}$ and test distribution $\mathcal{P}_{\text{test}}$ are the same. In the extrapolation regime, we may be asked to predict for situations outside the training domain $\mathcal{D}$, i.e., $\mathcal{P}_{\text{test}}$ is defined over $\mathcal{X} \setminus \mathcal{D}$. Suppose we train a neural network $f : \mathcal{X} \to \mathbb{R}^d$ with training examples $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n \subseteq \mathcal{D}$ by some training algorithm such as gradient descent. Let $g : \mathcal{X} \to \mathbb{R}^d$ be the underlying function (a correct reasoning process) and ℓ a loss function. Our goal is to understand under what conditions a neural network generalizes (interpolate or extrapolate) well in a task, i.e., the generalization or extrapolation error $\mathbb{E}_{\boldsymbol{x} \sim \mathcal{P}_{\text{test}}}[\ell(f(\boldsymbol{x}), g(\boldsymbol{x}))]$ is small.

Recent advances in deep learning show that highly over-parameterized models may generalize even better, suggesting the complexity of model space alone is not the determining factor of generalization [Zhang et al., 2017]. In our analysis, we take into account all the following factors: the training algorithm (e.g., gradient descent), the inductive biases of network structure, the task structure, and the training data distribution.

Approach	Assumptions	Practical implication	Regime
Complexity based [Scarselli et al., 2018] [Garg et al., 2020]	VC dimension, covering number via norms	Norms are unknown before training	Interpolation
Graph NTK [Du et al., 2019b] (Chapter 5)	+ Training algorithm	Provable learning of simple functions	Interpolation
Algorithmic alignment [Xu et al., 2020] [Xu et al., 2021b] (Chapter 6 and 8)	+ Network structure & task structure	Structured functions like algorithms	Interpolation & extrapolation

Table 1.1: **Approaches of generalization analysis of GNNs.** The first row shows existing approaches, and the second and third rows are our approaches. From top to bottom, more factors are assumed and taken into account, and in return we get more practical implications and refined analysis. In Chapter 5, we assume the training algorithm and show provable learning of simple functions on graphs. In Chapter 6 and 8, we additionally assume the network and task structure to analyze learning more complex and structured functions like reasoning algorithms. An important advantage of our approaches compared to complexity based approaches is that we can better predict in which tasks a network can perform well before actually training the neural network.

We study generalization for both interpolation and extrapolation, and show implications for reasoning. The table above provides an overview of the existing approaches and our approaches for generalization analysis of GNNs and demonstrates the differences in assumptions and practical implications. As a warm-up, we first study the *learning dynamics* of over-parameterized GNNs trained by gradient descent, and show equivalence to a graph neural tangent kernel. This relation gives generalization error bounds for GNNs and provable learning of simple functions on graphs. Second, we generalize our analysis to general network architectures, and show how the interplay of the network structure and the *algorithmic structure* of the reasoning process affects the sample efficiency. Our theory explains the success and limitations of GNNs, precisely characterizes what reasoning a neural network may sample-efficiently learn, and provides guidance for designing new architectures. Finally, we study how neural networks trained by gradient descent *extrapolate*, i.e., what they learn outside the support of the training distribution. We identify conditions under which feedforward neural networks and GNNs extrapolate well. Our analysis suggests implications for building models that generalize and extrapolate well for reasoning.

Above is our introduction to the main results of the thesis. In the last part of the thesis, to complete the picture of the theoretical understanding of GNNs, we present additional results on the optimization of GNNs. First, we analyze the optimization dynamics to show global convergence and implicit acceleration results. Second, we practically improve the training of GNNs by developing a normalization method which significantly accelerates the training as well as improves generalization.

1.2 Outline

This thesis consists of four parts. In Part I, we build models for representation learning. In Part II and Part III, we focus on reasoning. Each part also respectively studies one aspect of the theoretical landscape of learning: Part I studies expressive power; Part II studies generalization; Part III analyzes extrapolation; Part IV is about optimization.

Part I asks how to build powerful models for representation learning of graphs. We start by giving background in Chapter 2 on the problem formulation and graph neural networks. In Chapter 3, which is based on [Xu et al., 2018], we solve the puzzle of why deeper GNNs often underperform shallow GNNs, and propose a solution – Jumping Knowledge Networks (JK-Nets). Then in Chapter 4, based on [Xu et al., 2019], we introduce a theoretical framework to characterize the expressive power of GNNs for distinguishing non-isomorphic graphs, i.e., the graph isomorphism problem, and develop a maximally powerful GNN – the Graph Isomorphism Network (GIN).

Part II asks how to build models that generalize well for reasoning and what are the theoretical limits. We start with a warm-up in Chapter 5, based on [Du et al., 2019b], where we draw an equivalence of the learning dynamics of over-parameterized GNNs to that of graph neural tangent kernel (GNTK). This relation gives us generalization bound and provable learning of simple functions on graphs. We then answer the questions regarding reasoning in Chapter 6, based on [Xu et al., 2020], for general neural network architectures including GNNs. We introduce a theoretical framework, *algorithmic alignment*, that measures how well the network architecture aligns with the underlying reasoning process. We derive sample complexity bound that decreases with better alignment. Our framework implies GNNs can sample-efficiently learn dynamic programming, an algorithmic paradigm which solves a broad range of reasoning problems.

Part III asks how to design models that extrapolate well, i.e., generalize to unseen domains outside the training distribution. Extrapolation is especially desirable for reasoning. In Chapter 7, based on [Xu et al., 2021b], we analyze how feedforward neural networks trained by gradient descent extrapolate, and identify conditions under which they extrapolate well. Building upon the results of feedforward neural networks, in Chapter 8, we study how networks with more complex structures like GNNs extrapolate, and introduce the linear algorithmic alignment framework for improving extrapolation. Our analysis takes into account all the following factors: the training algorithm, the network architecture, the task structure, and the training distribution (e.g., data geometry and graph structure).

Finally, Part IV complements the previous parts and asks how to improve the optimization of GNNs. In Chapter 9, which is based on [Xu et al., 2021a], we establish that gradient descent training of linearized GNNs converges to a global minimum with a linear rate. Our results hold for GNNs with any finite depth, with or without skip connections. In addition, we study how factors like skip connections and depth influence the speed of convergence. In Chapter 10, based on [Cai et al., 2021], we practically improve the training of GNNs. We adapt, evaluate, and theoretically understand how normalization methods may help with training. As an example, we explain the effectiveness of InstanceNorm based on a preconditioning argument, and explain why BatchNorm is less effective by the variance of batch statistics in graphs. Our understanding further leads to GraphNorm, a method which significantly accelerates the training and improves the generalization.

1.3 Related Research

We first provide the complete references of the works covered in this thesis, and then other related works by the author not covered in this thesis.

- Part I - Chapter 3 [Xu et al., 2018]

  Representation Learning on Graphs with Jumping Knowledge Networks.

Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, Stefanie Jegelka. *In Proceedings of the 35th International Conference on Machine Learning, 2018.*

- Part I - Chapter 4 [Xu et al., 2019]

  How Powerful are Graph Neural Networks?

  Keyulu Xu, Weihua Hu, Jure Leskovec, Stefanie Jegelka. *In Proceedings of the 7th International Conference on Learning Representations, 2019.*

- Part II - Chapter 5 [Du et al., 2019b]

  Graph Neural Tangent Kernel: Fusing Graph Neural Networks with Graph Kernels.

  Simon S. Du, Kangcheng Hou, Barnabas Poczos, Ruslan Salakhutdinov, Ruosong Wang, Keyulu Xu. *Advances in Neural Information Processing Systems, 2019.*

- Part II - Chapter 6 [Xu et al., 2020]

  What Can Neural Networks Reason About?

  Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, Ken-ichi Kawarabayashi, Stefanie Jegelka. *In Proceedings of the 8th International Conference on Learning Representations, 2020.*

- Part III - Chapter 7 and 8 [Xu et al., 2021b]

  How Neural Networks Extrapolate: From Feedforward to Graph Neural Networks.

  Keyulu Xu, Mozhi Zhang, Jingling Li, Simon S. Du, Ken-ichi Kawarabayashi, Stefanie Jegelka. *In Proceedings of the 9th International Conference on Learning Representations, 2021.*

- Part IV - Chapter 9 [Xu et al., 2021a]

  Optimization of Graph Neural Networks: Implicit Acceleration by Skip Connections and More Depth.

Keyulu Xu, Mozhi Zhang, Stefanie Jegelka, Kenji Kawaguchi. *In Proceedings of the 38th International Conference on Machine Learning, 2021.*

- Part IV - Chapter 10 [Cai et al., 2021]

  GraphNorm: A Principled Approach to Accelerating Graph Neural Network Training.

  Tianle Cai, Shengjie Luo, Keyulu Xu, Di He, Tie-Yan Liu, Liwei Wang. *In Proceedings of the 38th International Conference on Machine Learning, 2021.*

Here we list other works by the author. While these papers are strongly related to the thesis, they were left out to maintain a clearer story.

- Work on natural language processing [Zhang et al., 2019b]:

  Are Girls Neko or Shōjo? Cross-Lingual Alignment of Non-Isomorphic Embeddings with Iterative Normalization.

  Mozhi Zhang, Keyulu Xu, Ken-ichi Kawarabayashi, Stefanie Jegelka, Jordan Boyd-Graber. *In Proceedings of the 57th Conference of the Association for Computational Linguistics, 2019,*

- Work on adversarial learning on graphs [Liao et al., 2021]:

  Information Obfuscation of Graph Neural Networks.

  Peiyuan Liao, Han Zhao, Keyulu Xu, Tommi Jaakkola, Geoffrey Gordon, Stefanie Jegelka, Russ Salakhutdinov. *In Proceedings of the 38th International Conference on Machine Learning, 2021.*

- Work on spectral graph theory [Harvey and Xu, 2016]:

  Generating Random Spanning Trees via Fast Matrix Multiplication.

  Nicholas J. A. Harvey and Keyulu Xu. *Latin American Theoretical Informatics Symposium, 2016.*

- Work on noisy labels and algorithmic alignment [Li et al., 2020b]:

Noisy Labels Can Induce Good Representations.

Jingling Li, Mozhi Zhang, Keyulu Xu, John P. Dickerson, Jimmy Ba.

- Work on generative models [Li et al., 2017]:

  Distributional Adversarial Networks.

  Chengtao Li, David Alvarez-Melis, Keyulu Xu, Stefanie Jegelka, Suvrit Sra. *International Conference on Learning Representations workshop, 2018.*

In connection to the current thesis, Zhang et al. [2019b] studies generalization across languages, a special type of extrapolation. Liao et al. [2021] states the inherent trade-off of task performance and adversarial defense for GNNs, complementing the generalization analysis in non-adversarial settings. Li et al. [2020b] is an extension and application of algorithmic alignment to the setting of training with noisy labels.

Chapter 2

Preliminaries

In this chapter, we introduce the basic problem setup and background on deep learning and graph neural networks[1].

Representation of graphs. Many objects in the world can be represented with graphs, explicitly or implicitly. A graph $G = (V, E)$ is defined by its graph structure (V, E), node features $X_v \in \mathbb{R}^{m_1}$ for $v \in V$, and edge features $X_{(u,v)} \in \mathbb{R}^{m_2}$ for $(u, v) \in E$. For example, a molecular graph encodes the atom types in node features and bond information in edge features; a global financial system encodes the institutional information in node features and inter-institutional information in edge features. One can generalize the image space $\mathbb{R}^{m \times n}$ or the sequence space $\mathbb{R}^{m \times l}$ to the graph space by encoding the positional information in the nodes or edges. It is also common to not have any node or edge features, in which case what matters is the graph structure. For such graphs, we can use a fixed vector (e.g., all-one vector) as node features.

Given the graph as input, there are mainly three types of prediction of interest: (1) *Node prediction*, where each node $v \in V$ has an associated label y_v and the goal is to learn a representation vector $h_v \in \mathbb{R}^d$ of v such that v's label can be predicted as $y_v = f^l(h_v)$ where $f^l : \mathbb{R}^d \to \mathbb{R}$ is a linear transform often learned end to end; (2) *Graph prediction*, where, given a set of graphs $\{G_1, ..., G_N\} \subseteq \mathcal{G}$ and their labels $\{y_1, ..., y_N\} \subseteq \mathcal{Y}$, we aim to learn a representation vector $h_G \in \mathbb{R}^d$ that helps predict the label of an entire graph, $y_G = f^l(h_G)$.

[1]We may use slightly different notations in each chapter for simplicity of exposition.

(3) *Edge prediction*, where a pair of nodes (u, v) has an associated label $y_{(u,v)}$ and the goal is to predict $y_{(u,v)}$ using either node representations h_u and h_v or additionally learn edge representations $h_{(u,v)}$ for prediction.

In general, we consider the supervised learning setting. We have a set of training examples $\{G_1, ..., G_N\}$. For each G_i we have label y_i for the entire graph, or labels for (a set of) the nodes on G_i, or labels for the edges. Let f be a graph neural network (GNN) that acts on a graph. The GNN f produces the 1) node representations h_v for $v \in V$, and 2) graph representation h_G for the entire graph[2]. Sometimes edge representations are also produced. We can train the GNN f by specifying a loss function on the labels (graph-level, node-level, or edge-level). For example, in the case of graph prediction with squared loss, we have the following loss:

$$\sum_{i=1}^{N} \left(f^l(f(G_i)) - y_i \right)^2, \tag{2.1}$$

where the linear transform $f^l : \mathbb{R}^d \to \mathbb{R}$ and $f : \mathcal{G} \to \mathbb{R}^d$ are trained together end to end. We train the GNN by minimizing the loss function with a training algorithm such as stochastic gradient descent. Next, we introduce background on GNNs.

Graph neural networks. GNN is a class of neural networks acting on graphs. The concept was initially proposed by [Gori et al., 2005, Scarselli et al., 2009]. GNNs use the graph structure, node features X_v, and edge features $X_{(u,v)}$ to learn representation vectors of the nodes h_v and the entire graph h_G.

Modern GNNs follow a recursive message passing (or neighbor aggregation) framework [Gilmer et al., 2017, Xu et al., 2018, 2019], where we iteratively update the representation of a node $h_v^{(k)}$ in iteration k by aggregating representations of its neighbors $h_u^{(k)}$, where $u \in \mathcal{N}(v)$ are adjacent to v. The initial node representations $h_v^{(0)}$ are set to the input node features $h_v^{(0)} = X_v$. After k iterations of aggregation or message passing, a node's

[2]We sometimes use bold symbols to indicate vectors. The notation depends on the specific chapter.

representation captures the information within its k-hop neighborhood.

$$a_v^{(k)} = \text{AGGREGATE}^{(k)} \left(\left\{ h_u^{(k-1)} : u \in \mathcal{N}(v) \right\} \right), \tag{2.2}$$

$$h_v^{(k)} = \text{COMBINE}^{(k)} \left(h_v^{(k-1)}, a_v^{(k)} \right), \tag{2.3}$$

where $h_v^{(k)}$ is the feature vector of node v in the k-th iteration/layer, and $\mathcal{N}(v)$ is a set of nodes adjacent to v. The choice of AGGREGATE$^{(k)}(\cdot)$ and COMBINE$^{(k)}(\cdot)$ in GNNs is crucial, and many architectures for AGGREGATE have been proposed [Duvenaud et al., 2015, Defferrard et al., 2016, Kearnes et al., 2016, Kipf and Welling, 2017]. To incorporate edge features, we can simply apply the following similar framework:

$$h_v^{(k)} = \text{AGGREGATE}^{(k)} \left(\left\{ \left(h_u^{(k-1)}, h_v^{(k-1)}, X_{(u,v)} \right) : u \in \mathcal{N}(v) \right\} \right). \tag{2.4}$$

For node prediction or edge prediction, the node representation $h_v^{(K)}$ of the final iteration is used for prediction. We can also use node representations from previous iterations. We will see the benefits of doing so in Chapter 3.

For graph prediction, we aggregate node representations for all nodes on the graph to obtain the entire graph's representation h_G:

$$h_G = \text{READOUT}\Big(\Big\{ h_v^{(K)} \Bigm| v \in G \Big\} \Big). \tag{2.5}$$

READOUT can be a simple permutation invariant function such as summation or a more sophisticated function. In the main thesis, we will understand how to build the AGGREGATE and READOUT for better expressive power, generalization, and extrapolation. For now, let us see a few example architectures from the previous works.

In Graph Convolutional Networks (GCN) [Kipf and Welling, 2017], the element-wise *mean* pooling is used for aggregating messages. Let W^k denote a learnable weight matrix and ReLU the non-linearity activation function ReLU$(x) = \max(0, x)$. The update steps of GCN are as follows:

$$h_v^{(k)} = \text{ReLU} \left(W^k \cdot \text{MEAN} \left\{ h_u^{(k-1)}, \ \forall u \in \mathcal{N}(v) \cup \{v\} \right\} \right). \tag{2.6}$$

The following update with a slightly different normalization is also used for GCNs:

$$h_v^{(k)} = \text{ReLU}\Big(W^k \cdot \sum_{u \in \widetilde{\mathcal{N}}(v)} (\deg(v)\deg(u))^{-1/2}\, h_u^{(k-1)}\Big), \tag{2.7}$$

where deg(v) stands for the degree of node v in G, and $\widetilde{\mathcal{N}}(v)$ denotes $\mathcal{N}(v) \cup v$.

A second example is GraphSAGE [Hamilton et al., 2017], whose AGGREGATE has been formulated as

$$a_v^{(k)} = \text{MAX}\left(\left\{\text{ReLU}\left(W^k \cdot h_u^{(k-1)}\right),\ \forall u \in \mathcal{N}(v)\right\}\right), \tag{2.8}$$

where MAX represents an element-wise max-pooling. Other versions of GraphSAGE are also possible, e.g., using MEAN or RNN in place of MAX. The COMBINE step is a concatenation followed by a linear mapping $W \cdot \left[h_v^{(k-1)}, a_v^{(k)}\right]$.

Other examples are Graph Attention Network (GAT) [Velickovic et al., 2018] and Transformer [Vaswani et al., 2017], whose update step uses the attention mechanism [Bahdanau et al., 2014], i.e., weighted average with learned weights. There are multiple ways to parameterize the attention weights, e.g., by dot product or a feedforward neural network. We refer to the respective papers for the specific attention operation used. It is worth mentioning that Transformers are proposed for natural language processing where the input is often a sequence, and it has been successfully applied to many other domains such as images [Dosovitskiy et al., 2020]. We can view the structure of Transformer as a message passing GNN acting on a complete graph, where each node is a word in a sequence and the node feature encodes the positional information.

Besides the message passing view, we can also view GNNs as structured neural networks with weight shared modules that parameterize the messages. This view is helpful in understanding GNNs for learning reasoning problems.

Reasoning with graph neural networks. Reasoning spans a variety of tasks, for instance, visual and text-based question answering [Johnson et al., 2017a, Weston et al., 2015, Hu et al., 2017, Fleuret et al., 2011, Antol et al., 2015], intuitive physics, i.e., predicting the time evolution of physical objects [Battaglia et al., 2016, Watters et al., 2017, Fragkiadaki et al., 2016, Wu et al., 2017], mathematical reasoning [Saxton et al., 2019, Chang et al., 2019] and visual IQ tests [Santoro et al., 2018, Zhang et al., 2019a].

It is often more sample efficient to separate reasoning from perception or representation [Mao et al., 2019, Yi et al., 2018a]. Hence, following previous works, we assume access to the representation of objects in a world or universe S, i.e., a configuration/set of objects to reason about. Each object $s \in S$ is represented by a representation vector X_s. The representation could be descriptions prepared by humans or features learned by representation learning steps such as image segmentation [Santoro et al., 2017]. Information about the specific question can also be included in the object representations. Given a set of universes $\{S_1, ..., S_M\}$ and answer labels $\{y_1, ..., y_M\} \subseteq \mathcal{Y}$, we aim to learn a function g (a reasoning process) that can answer questions about unseen universes, $y = g\left(S\right)$. Mathematically, a universe S can be a set, a sequence, or a graph, depending on the information available between the objects.

There are many reasoning models. Here we introduce GNNs for reasoning and defer the introduction of other models to the main thesis. Suppose the input S is a *set* of object representations. GNNs can act on the set by considering objects as nodes and assuming all objects pairs are connected, *i.e.*, a complete graph:

$$h_s^{(k)} = \sum\nolimits_{t \in S} \text{MLP}_1^{(k)} \left(h_s^{(k-1)}, h_t^{(k-1)} \right), \quad h_S = \text{MLP}_2 \Big(\sum\nolimits_{s \in S} h_s^{(K)} \Big), \tag{2.9}$$

where h_S is the answer/output and K is the number of GNN layers. Each object's representation is initialized as $h_s^{(0)} = X_s$. MLP stands for multilayer perceptron, a.k.a. feedforward neural networks. We can also easily incorporate edge features into the MLP module of (2.9). The slight differences of the most popular GNN architectures for representation and reasoning perhaps stem from the difference of the underlying function, e.g., statistical patterns in graphs or a reasoning process.

Part I

Representation: Expressive Power

Chapter 3

Puzzle of the Underperformance of Deeper GNNs

This part of the thesis is about modeling intelligence for representation. A good representation requires powerful and expressive models that can capture fine-grained information of an object. In the following two chapters, we address two fundamental questions regarding representation learning of graphs. First, surprisingly, deeper GNNs often underperform shallow GNNs in node prediction tasks. We give an explanation for this puzzle and propose a solution. Second, we ask how powerful GNNs are for distinguishing non-isomorphic graphs, i.e., the graph isomorphism problem. We introduce a framework to analyze the discriminative power of GNNs and design a simple yet powerful GNN architecture.

3.1 Introduction

Graphs are a ubiquitous structure. Real-world graphs such as social networks, financial networks, and biological networks represent important statistical information through its structures, for example, the communities a person is in, the functional role of a molecule, and the sensitivity of the assets of an enterprise to external shocks.

Representation learning of nodes in graphs aims to extract high-level features from a node as well as its neighborhood, and has proved extremely useful for many applications, such as node classification, clustering, and link prediction [Perozzi et al., 2014, Monti et al., 2017, Grover and Leskovec, 2016, Tang et al., 2015]. Graph neural networks (GNNs) are an effective framework for representation learning of nodes and graphs [Scarselli et al., 2009]. These models learn to iteratively aggregate the hidden features of every node in the graph with its adjacent nodes' as its new hidden features, where an iteration is parametrized by a layer of the neural network. Theoretically, an aggregation process of k iterations makes use of the subgraph structures within k hops.

Such promising models sometimes lead to surprises. For example, it has been observed that the best performance with one of the existing GNN models, Graph Convolutional Networks (GCN), is achieved with a 2-layer model. Deeper versions of the model that, in principle, have access to more information, perform worse [Kipf and Welling, 2017]. A similar degradation of learning for computer vision problems is resolved by residual connections [He et al., 2016] that greatly aid the training of deep models. But, even with residual connections, GCNs with more layers do not perform as well as the 2-layer GCN on many datasets, especially for node prediction tasks.

Motivated by this puzzle, in this chapter, we address two questions. First, we study properties and resulting limitations of the *learned node representations* of message passing GNNs. Our analysis explains the puzzle by relating the learned representations to the subgraph structure (e.g., expanders or trees) and suggests how the optimal depth depends on the structure. Second, based on this analysis, we propose a more powerful architecture that, as opposed to existing models, enables *adaptive, structure-aware* representations and makes deeper GNNs perform better. Such representations are particularly interesting for representation learning on large complex graphs with diverse subgraph structures.

Model analysis. To understand the learned node representations of message passing GNNs, we analyze the *effective range* of nodes that any given node's representation draws from. We summarize this sensitivity analysis by what we name the *influence distribution* of a node. This effective range implicitly encodes prior assumptions on what are the "nearest neighbors" that a node should draw information from. In particular, we will see that this influence is heavily affected by the graph structure, raising the question whether "one size fits all", in particular in graphs whose subgraphs have varying properties (such as more

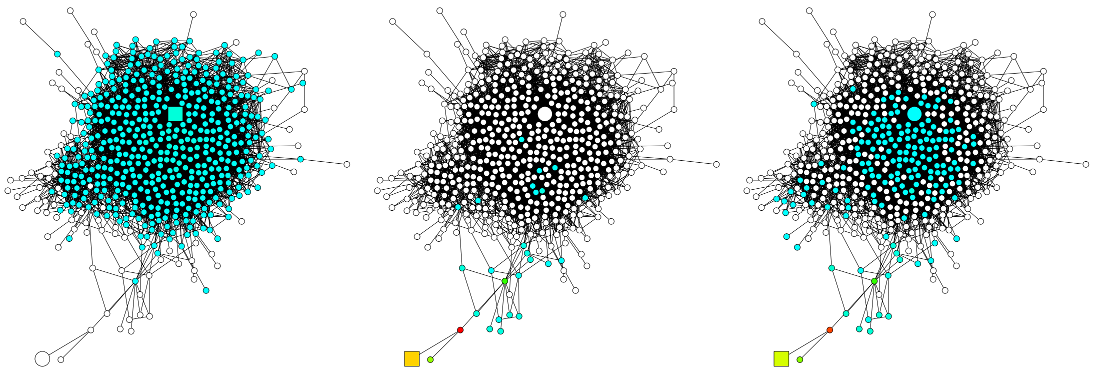

(a) 4 steps at expander (b) 4 steps at tree (c) 5 steps at tree

Figure 3-1: **Expansion of a random walk and influence distribution** starting at square nodes in subgraphs with different structures. Different subgraph structures result in very different effective neighborhood sizes.

tree-like or more expander-like).

In particular, our more formal analysis connects influence distributions with the spread of a random walk at a given node, a well-understood phenomenon as a function of the graph structure and eigenvalues [Lovász, 1993]. For instance, in some cases and applications, a 2-step random walk influence that focuses on local neighborhoods can be more informative than higher-order features where some of the information may be "washed out" via averaging.

Changing locality. To illustrate the effect and importance of graph structure, recall that many real-world graphs possess locally strongly varying structure. In biological and citation networks, the majority of the nodes have few connections, whereas some nodes (hubs) are connected to many other nodes. Social and web networks usually consist of an expander-like core part and an almost-tree (bounded treewidth) part, which represent well-connected entities and the small communities respectively [Leskovec et al., 2009, Maehara et al., 2014, Tsonis et al., 2006].

Besides node features, this subgraph structure has great impact on the result of neighborhood aggregation. The speed of expansion or, equivalently, growth of the influence radius, is characterized by the random walk's mixing time, which changes dramatically on subgraphs with different structures [Lovász, 1993]. Thus, the same number of iterations (layers) can lead to influence distributions of very different locality. As an example, consider the social network in Figure 3-1 from GooglePlus [Leskovec and Mcauley, 2012]. The figure illustrates the expansions of a random walk starting at the square node. The walk (a) from a node within the core rapidly includes almost the entire graph. In contrast, the walk (b) starting at a node in the tree part includes only a very small fraction of all nodes. After 5 steps, the same walk has reached the core and, suddenly, spreads quickly. Translated to graph representation models, these spreads become the influence distributions or, in other words, the averaged features yield the new feature of the walk's starting node. This shows that in the same graph, the same number of steps can lead to very different effects. Depending on the downstream task, wide-range or small-range feature combinations may be more desirable. A too rapid expansion may average too broadly and thereby lose information, while in other parts of the graph, a sufficient neighborhood may be needed for stabilizing predictions. When a majority of the nodes are in such expander subgraphs, a two-layer network may outperform deeper networks.

JK networks. The above observations raise the question whether it is possible to adaptively *adjust* (i.e., learn) the influence radii for each node and task, and hence make it possible for GNNs to go deeper. To achieve this, we explore an architecture that learns to selectively exploit information from neighborhoods of differing locality. This architecture selectively combines different aggregations at the last layer, i.e., the representations "jump" to the last layer. Hence, we name the resulting networks *Jumping Knowledge Networks (JK-Nets)*. We will see that empirically, when adaptation is an option, the networks indeed learn representations of different orders for different graph substructures. Moreover, we show that applying our framework to various state-of-the-art GNN base models consistently improves their performance.

3.1.1 Background on Graph Theory

We define two graph structures: expanders and bounded treewidth graphs.

Expanders. Intuitively, an expander is a possibly sparse graph that has strong connectivity properties, which can be quantified via vertex, edge or spectral expansion. A graph is an expander if it has high expansion parameters, e.g., high edge expansion.

Definition 3.1 (Edge expansion)**.** The edge expansion (or Cheeger constant) $h(G)$ of a graph G with n vertices is defined as

$$h(G) = \min_{0<|S|\leq\frac{2}{n}} \frac{|\partial S|}{|S|},$$

where the edge boundary $\partial S = \{(u, v) \in E(G) : u \in S, v \in V(G) \setminus S\}$

The higher the expansion parameter, the larger the smallest graph cut, which implies such expander graphs are highly connected. When we start a random walk on an expander, due to the highly connected structure, the random walk expands very rapidly and will cover almost the entire graph in a few iterations.

Some examples of expander graphs are random regular graphs, where each node is connected to a fixed number of neighbors randomly. In many real-world graphs, such as social networks and citation networks, there usually exist expander parts.

Bounded treewidth graphs. In contrast to expanders, the properties of bounded treewidth graphs are similar to trees. Trees induce small cuts. Hence, a random walk starting in a tree-like structure expands slowly. We can formally define tree-like graphs by the treewidth and tree decomposition [Arnborg et al., 1987]. The smaller the treewidth, the more tree-like the graph is. The treewidth of a graph can be computed via tree decomposition. In our analysis, we do not need to perform tree decomposition to compute the treewidth of a graph. We will instead be using the fact that real-world graphs contain bounded treewidth parts and random walk expansion on such subgraphs are slow.

3.2 Understanding How Deeper GNNs May Underperform

We present a theoretical framework "influence distributions" for analyzing the properties of GNN architectures. Our analysis framework relates a GNN architecture and its learned node representations to a random walk distribution. The behavior of a random walk distribution is highly related to the graph structure where it starts from, e.g., expanders or bounded treewidth graphs. Therefore, we effectively relate the effect of a GNN aggregation procedure to the graph structures it is acting on. From there, we conclude that we need to build more powerful GNN models are *structure-aware*, in the sense that the optimal number of random walk steps is adaptively applied to different subgraph structures.

3.2.1 Influence Distribution and Random Walks

We start by exploring some important properties of the message passing schemes of GNNs. Related to ideas of sensitivity analysis and influence functions in statistics [Koh and Liang, 2017] that measure the influence of a training point on parameters, we study the range of nodes whose features affect a given node's representation. This range gives insight into how large a neighborhood a node is drawing information from.

We measure the sensitivity of node x to node y, or the influence of y on x, by measuring how much a change in the input feature of y affects the representation of x in the last layer. For any node x, the *influence distribution* captures the relative influences of all other nodes.

Definition 3.2. (Influence score and distribution). For a simple graph $G = (V, E)$, let $h_x^{(0)}$ be the input feature and $h_x^{(k)}$ be the learned hidden representation of node $x \in V$ at the k-th (last) layer of the model. The *influence score* $I(x, y)$ of node x by any node $y \in V$ is the sum of the absolute values of the entries of the Jacobian matrix $\left[\frac{\partial h_x^{(k)}}{\partial h_y^{(0)}}\right]$. We define the *influence distribution* I_x of $x \in V$ by normalizing the influence scores: $I_x(y) = I(x, y) / \sum_z I(x, z)$, or

$$I_x(y) = e^T \left[\frac{\partial h_x^{(k)}}{\partial h_y^{(0)}}\right] e \Big/ \left(\sum_{z \in V} e^T \left[\frac{\partial h_x^{(k)}}{\partial h_z^{(0)}}\right] e\right),$$

where e is the all-ones vector.

Later, we will see connections of influence distributions with random walks. For completeness, we also define random walk distributions.

Definition 3.3. (Random walk distribution). Consider a random walk on $\widetilde{G}$, the graph G with self-loops, starting at a node v_0; if at the t-th step we are at a node v_t, we move to any neighbor of v_t (including v_t) with equal probability. The *t-step random walk distribution* P_t of v_0 is defined as

$$\mathbb{P}_t\left(i\right) = \mathbb{P}\left(v_t = i\right). \tag{3.1}$$

Analogous definitions apply for random walks with non-uniform transition probabilities.

An important property of the random walk distribution is that it becomes more spread out as t increases and converges to the limit distribution if the graph is non-bipartite. The rate of convergence depends on the structure of the subgraph and can be bounded by the spectral gap (or the conductance) of the random walk's transition matrix [Lovász, 1993].

Model Analysis The influence distribution for different aggregation models and nodes can give insights into the information captured by the respective representations. The following results show that the influence distributions of common aggregation schemes are closely connected to random walk distributions. This observation hints at specific implications – strengths and weaknesses – that we will discuss.

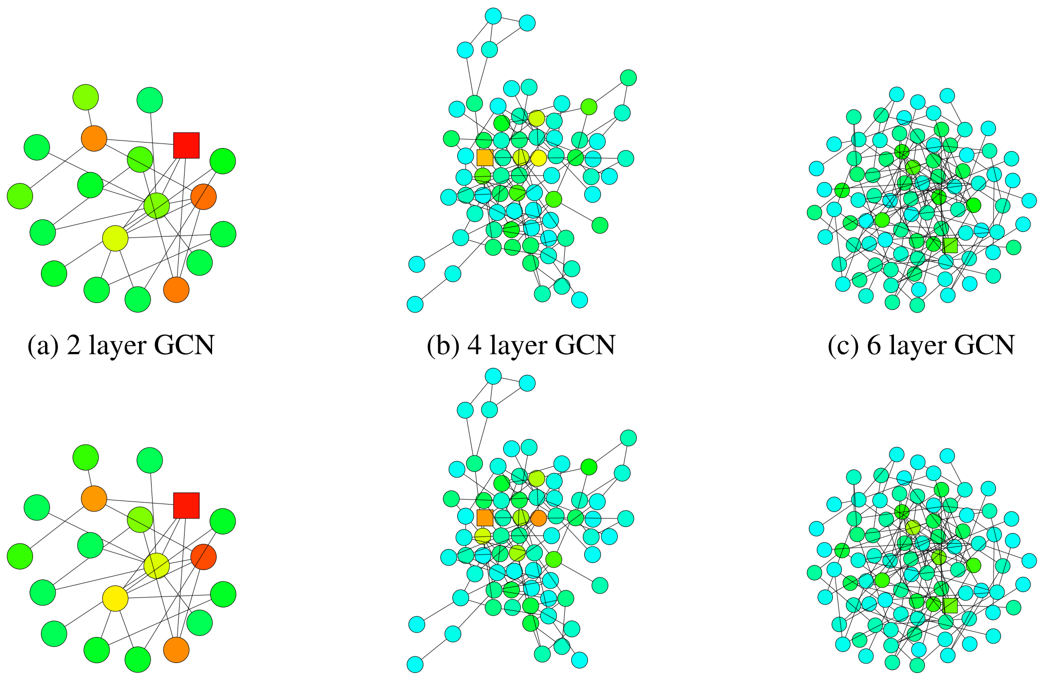
With a randomization assumption of the ReLU activations similar to that in [Kawaguchi, 2016, Choromanska et al., 2015], we can draw connections between GCNs and random walks:

Theorem 3.1. *(Equivalence to random walk distribution). Given a k-layer GCN, assume that all paths in the computation graph of the model are activated with the same probability of success ρ. Then the influence distribution I_x for any node $x \in V$ is equivalent, in expectation, to the k-step random walk distribution on $\widetilde{G}$ starting at node x.*

Proofs of all theorems may be found in Appendix.

Similarly, we can show that neighborhood aggregation schemes with directional biases (e.g., attention models) resemble biased random walk distributions.

Empirically, we observe that, despite somewhat simplifying assumptions, our theory is close to what happens in practice. We visualize the heat maps of the influence distributions for a node (labeled square) for trained GCNs, and compare with the random walk distributions starting at the same node. Figure 3-2 shows example results. Darker colors correspond



(a) 2 layer GCN (b) 4 layer GCN (c) 6 layer GCN

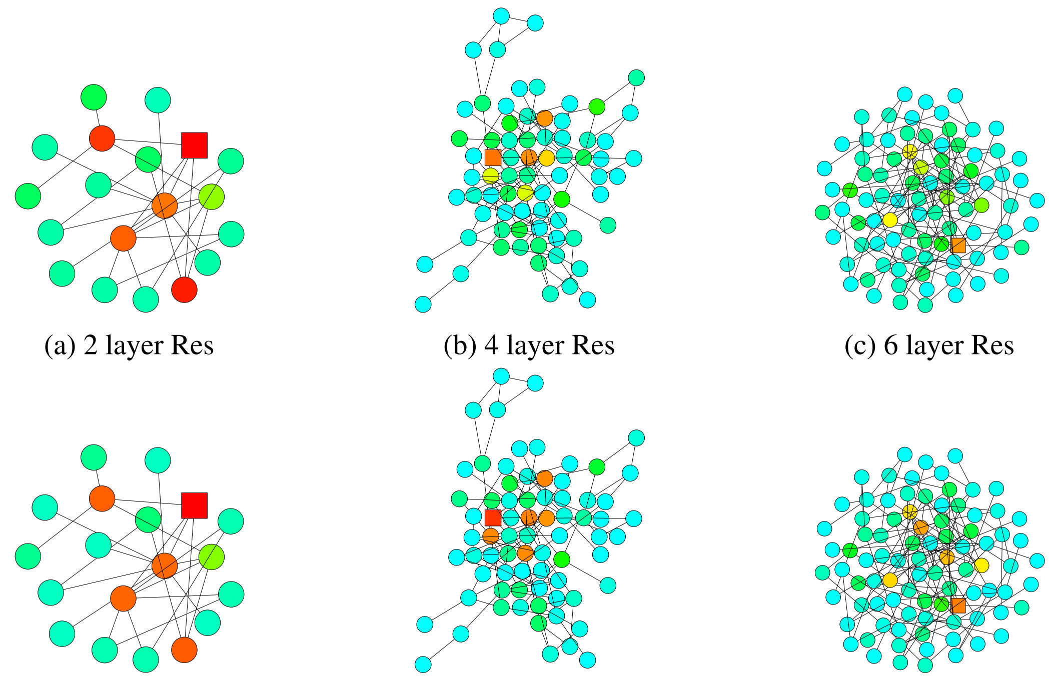
(d) 2 step r.w. (e) 4 step r.w. (f) 6 step r.w.

Figure 3-2: **Equivalence of influence distribution and random walk.** Top panel shows influence distributions of GCNs and bottom panel shows random walk distributions starting at the square node.

to higher influence probabilities. To show the effect of skip connections, Figure 3-3 visualizes the analogous heat maps for one example—GCN with residual connections. Indeed, we observe that the influence distributions of networks with residual connections approximately correspond to lazy random walks: each step has a higher probability of staying at the current node. Local information is retained with similar probabilities for all nodes in each iteration; this cannot adapt to diverse needs of specific upper-layer nodes. Further visualizations may be found in the appendix.

3.2.2 Fast Collapse on Expanders

To better understand the implication of Theorem 3.1 and the limitations of the corresponding neighborhood aggregation algorithms, we revisit the scenario of learning on a social network shown in Figure 3-1. Random walks starting inside an expander converge rapidly in $O(\log |V|)$ steps to an almost-uniform distribution [Hoory et al., 2006]. After $O(\log |V|)$


(a) 2 layer Res (b) 4 layer Res (c) 6 layer Res

(d) 2 step lazy r.w. (e) 4 step lazy r.w. (f) 6 step lazy r.w.

Figure 3-3: **Equivalence of influence distribution and lazy random walk.** Top panel shows influence distributions of GCNs with residual connections and bottom panel shows random walk distributions starting at the square node with lazy factor 0.4.

iterations of neighborhood aggregation, by Theorem 3.1 the representation of every node is influenced almost equally by any other node in the expander. Thus, the node representations will be representative of the global graph and carry limited information about individual nodes. In contrast, random walks starting at the bounded tree-width (almost-tree) part converge slowly, i.e., the features retain more local information. Models that impose a fixed random walk distribution inherit these discrepancies in the speed of expansion and influence neighborhoods, which may not lead to the best representations for all nodes.

Answers to the Puzzles Given our analysis, we restate an explanation for the puzzle.

First, as we have already seen in the analysis of fast collapse on expanders, more layers will lead to rapid expansion of influence distribution on expanders and thus possible degradation in performance depending on the downstream task and noise ratio in the more distant neighbors. Suppose a majority of the nodes are in expander subgraphs. Though more layers may lead to better performance in bounded treewidth parts, when a fixed

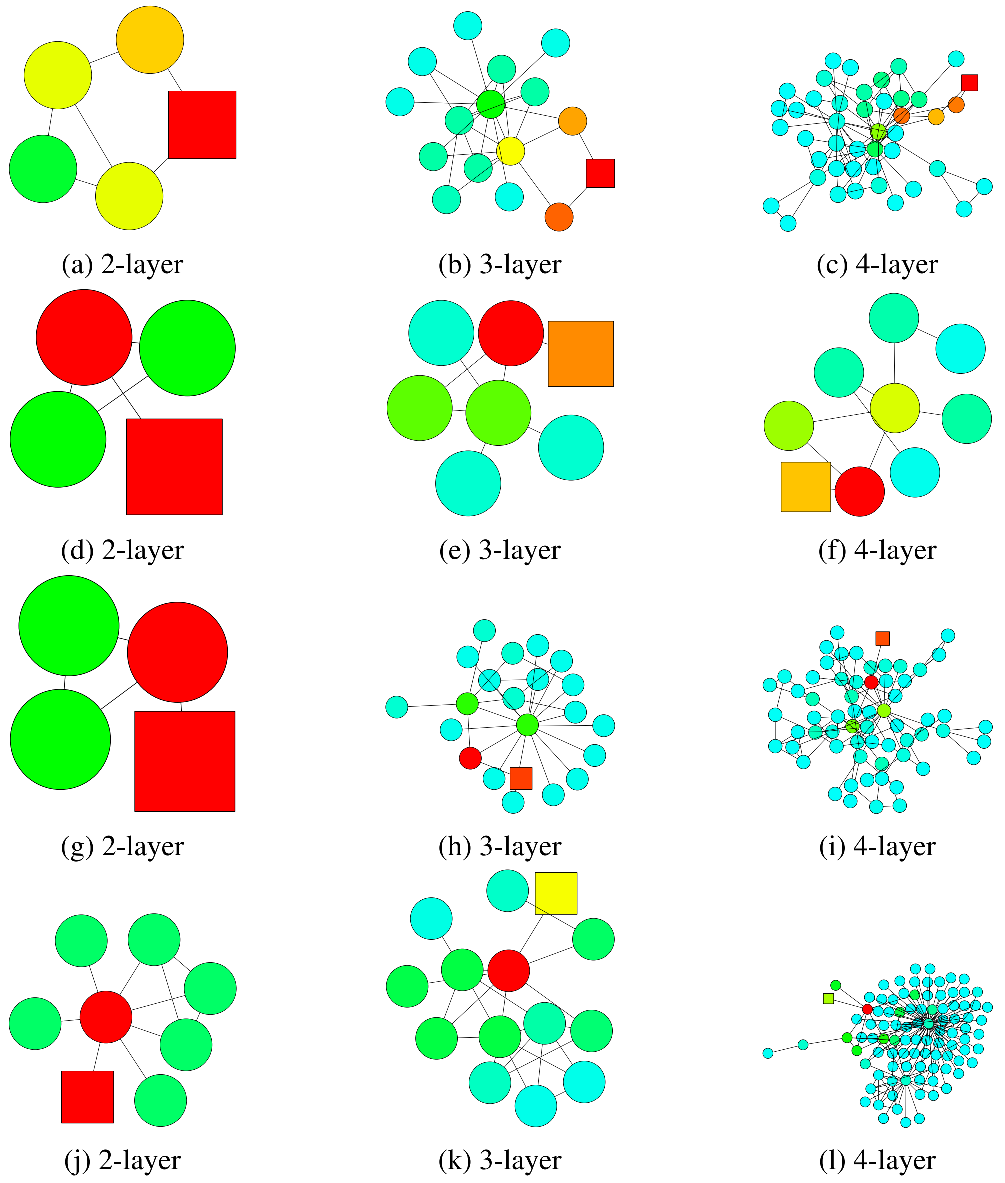


Figure 3-4: Subgraph structures where two-layer GCNs make a mistake, whereas three and four-layer GCNs make the correct prediction.

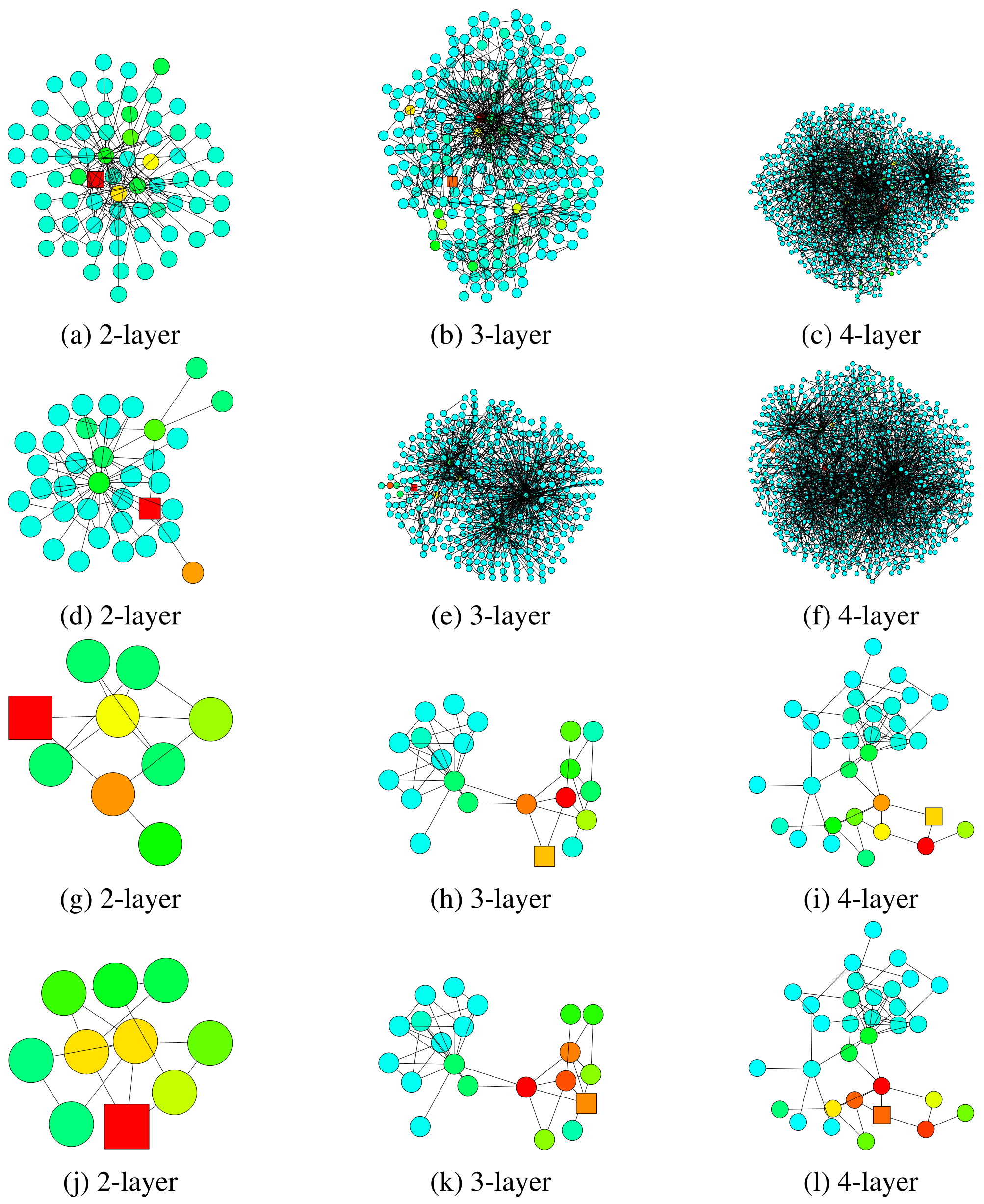


Figure 3-5: Subgraph structures where three and four-layer GCNs make a mistake, whereas two-layer GCNs make the correct prediction.

number of layers ia applied and no structure-aware adaptivity is available, the averaged best performance may be achieved with only two GNN layers.

Second, why does residual connection not resolve the problem? As we have seen in Figure 3-3, skip connections like residual connections play a role of "laziness" in the random walk expansion. The laziness, however, cannot fundamentally resolve, but at most alleviate, the problem of discrepancy in expansion speed in different graph structures. Therefore, a new architecture to address the problem is desirable. In contrast, in computer vision, the images (formulated as regular grid graphs) are far from expanders and the structures are the same almost everywhere. Hence, the same problem may not occur there.

We give more empirical evidence via visualization of the misclassfication results in node classification tasks. We demonstrate the typical subgraph structures where two-layer GCN models tend to make a mistake, whereas models with 3 or 4 layers are able to make the correct prediction, and vice versa. These visualization further complements Figure 3-1 and Theorem 3.1. In Figure 3-4, a two-layer GCN tends to make incorrect predictions if the local subgraph structure is tree-like (bounded treewidth). Thus, it would be desirable to look beyond the direct neighbors and draw information from nodes that are 3 or 4 hops away to learn a better representation. On the other hand, in Figure 3-5, three or four-layer GCNs may draw much information from less relevant neighbors and thus cannot learn the right representations. In expander subgraphs, models with 3 or 4 layers are essentially taking into account every node. Such global representations might not be ideal for the prediction for the node. In another scenario, despite possessing the locally bounded treewidth structure, because of the "bridge-like" structures, looking at distant nodes might imply drawing information from a completely different community, which would act like noises and influence the prediction results.

3.3 Jumping Knowledge Network (JK-Net)

In this section, we develop a more powerful architecture Jumping Knowledge Networks (JK-Nets), which can be applied on top of any base GNNs and enable adaptively learning the node representations and appropriate influence radii with respect to the subgraph structures.

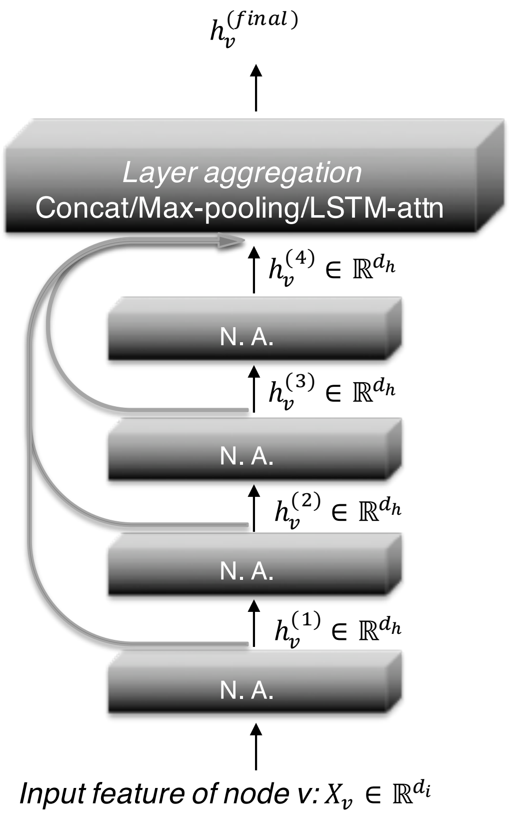

$h_v^{(final)}$
Layer aggregation
Concat/Max-pooling/LSTM-attn
$h_v^{(4)} \in \mathbb{R}^{d_h}$
N. A.
$h_v^{(3)} \in \mathbb{R}^{d_h}$
N. A.
$h_v^{(2)} \in \mathbb{R}^{d_h}$
N. A.
$h_v^{(1)} \in \mathbb{R}^{d_h}$
N. A.
Input feature of node v: $X_v \in \mathbb{R}^{d_i}$

Figure 3-6: **Jumping Knowledge Network (JK-Net)**. N.A. stands for neighborhood aggregation. The intermediate layer representations are aggregated at the final layer via e.g., concatenation, max-pooling, and LSTM-attention.

In particular, JK-Net enables *deep* GNNs.

Our theory above raise the question whether the fixed but structure-dependent influence radius size induced by common aggregation schemes really achieves the best representations for all nodes and tasks. Large radii may lead to too much averaging, while small radii may lead to instabilities or insufficient information aggregation. Hence, we propose two simple yet powerful architectural changes – jump connections and a subsequent selective but adaptive aggregation mechanism.

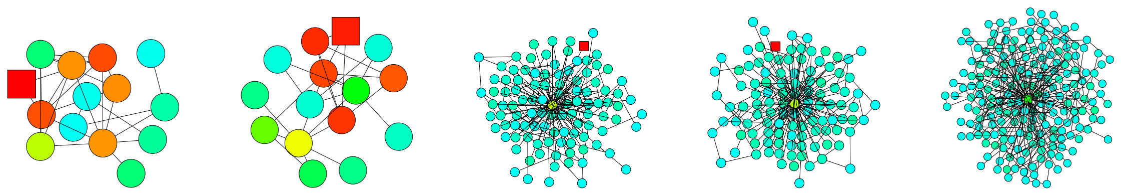
Figure 3-6 illustrates the main idea: as in common neighborhood aggregation networks, each layer increases the size of the influence distribution by aggregating neighborhoods from the previous layer. At the last layer, for each node, we carefully select from all of those itermediate representations (which “jump” to the last layer), potentially combining a few. If this is done independently for each node, then the model can adapt the effective neighborhood size for each node as needed, resulting in exactly the desired adaptivity.

Our model permits general layer-aggregation mechanisms. We explore three approaches; others are possible too. Let $h_v^{(1)}, ..., h_v^{(k)}$ be the jumping representations of node v (from k layers) that are to be aggregated.

Concatenation. A concatenation $\left[h_v^{(1)}, ..., h_v^{(k)}\right]$ is the most straightforward way to combine the layers, after which we may perform a linear transformation. If the transformation weights are shared across graph nodes, this approach is not node-adaptive. Instead, it optimizes the weights to combine the subgraph features in a way that works best for the dataset overall. One may expect concatenation to be suitable for small graphs and graphs with regular structure that require less adaptivity; also because weight-sharing helps reduce overfitting.

Max-pooling. An element-wise $\max\left(h_v^{(1)}, ..., h_v^{(k)}\right)$ selects the most informative layer *for each feature coordinate*. For example, feature coordinates that represent more local properties can use the feature coordinates learned from the close neighbors and those representing global status would favor features from the higher-up layers. Max-pooling is adaptive and has the advantage that it does not introduce any additional parameters to learn.

LSTM-attention. An attention mechanism identifies the most useful neighborhood ranges for each node v by computing an attention score $s_v^{(l)}$ for each layer l $\left(\sum_l s_v^{(l)} = 1\right)$, which represents the importance of the feature learned on the l-th layer for node v. The aggregated representation for node v is a weighted average of the layer features $\sum_l s_v^{(l)} \cdot h_v^{(l)}$. For LSTM attention, we input $h_v^{(1)}, ..., h_v^{(k)}$ into a bi-directional LSTM [Hochreiter and Schmidhuber, 1997] and generate the forward-LSTM and backward-LSTM hidden features $f_v^{(l)}$ and $b_v^{(l)}$ for each layer l. A linear mapping of the concatenated features $[f_v^{(l)}||b_v^{(l)}]$ yields the scalar importance score $s_v^{(l)}$. A Softmax layer applied to $\{s_v^{(l)}\}_{l=1}^k$ yields the attention of node v on its neighborhood in different ranges. Finally we take the sum of $[f_v^{(l)}||b_v^{(l)}]$ weighted by $\texttt{SoftMax}(\{s_v^{(l)}\}_{l=1}^k)$ to get the final layer representation. Another possible implementation combines LSTM with max-pooling. LSTM-attention is node adaptive because the attention scores are different for each node. We shall see that the this approach shines on large



(a) tree-like (b) tree-like (c) affiliate (d) affiliate (e) hub

Figure 3-7: **JK-Net learns to adapt to different subgraph structures**

complex graphs, although it may overfit on small graphs (fewer training nodes) due to its relatively higher complexity.

3.3.1 JK-Net Learns to Adapt to Subgraph Structure

The key idea for the design of layer-aggregation functions is to determine the importance of a node's subgraph features at different ranges after looking at the learned features on all layers, rather than to optimize and fix the same weights for all nodes. Under the same assumption on the ReLU activation distribution as in Theorem 3.1, we show below that layer-wise max-pooling implicitly learns the influence locality adaptively for different nodes. The proof for layer-wise attention follows similarly.

Proposition 3.2. *Assume that paths of the same length in the computation graph are activated with the same probability. The influence score* $I(x, y)$ *for any* $x, y \in V$ *under a* k*-layer JK-Net with layer-wise max-pooling is equivalent in expectation to a mixture of* $0, .., k$*-step random walk distributions on* $\widetilde{G}$ *at* y *starting at* x*, the coefficients of which depend on the values of the layer features* $h_x^{(l)}$.

Contrasting this result with the influence distributions of other aggregation mechanisms, we see that JK-networks indeed differ in their node-wise adaptivity of neighborhood ranges.

Figure 3-7 illustrates how a 6-layer JK-Net with max-pooling aggregation learns to adapt to different subgraph structures on a citation network. Within a tree-like structure, the influence stays in the "small community" the node belongs to. In contrast, 6-layer models whose influence distributions follow random walks, e.g. GCNs, would reach out too far into irrelevant parts of the graph, and models with few layers may not be able to cover the entire "community", as illustrated in Figure 3-1, and Figures 3-4, 3-5. For a node affiliated to a "hub", which presumably plays the role of connecting different types of nodes, JK-Net learns to put most influence on the node itself and otherwise spreads out the influence. GCNs, however, would not capture the importance of the node's own features in such a structure because the probability at an affiliate node is small after a few random walk steps. For hubs, JK-Net spreads out the influence across the neighboring nodes in a reasonable range, which makes sense because the nodes connected to the hubs are presumably as informative as the hubs' own features. For comparison, Table A.1 in the appendix includes more visualizations of how models with random walk priors behave.

3.3.2 Dense JK-Nets

Looking at Figure 3-6, one may wonder whether the same inter-layer connections could be drawn between all layers. The resulting architecture is approximately a graph correspondent of DenseNets, which were introduced for computer vision problems [Huang et al., 2017], if the layer-wise concatenation aggregation is applied. This version, however, would require many more features to learn. Viewing the DenseNet setting (images) from a graph-theoretic perspective, images correspond to regular, in fact, near-planar graphs. Such graphs are far from being expanders, and do not pose the challenges of graphs with varying subgraph structures. Indeed, as we shall see, models with concatenation aggregation, e.g., JK-Concat, perform well on graphs with more regular structures such as images and well-structured communities. On the other hand, on graphs with more diverse structures, the node-wise adaptive methods like JK-LSTM perform better. As a more general framework, JK-Net admits general layer-wise aggregation models and enables better structure-aware representations on graphs with complex structures.

3.4 Experiments

We evaluate JK-Nets on four node classification benchmark datasets. (I) The task on citation networks (Citeseer, Cora) [Sen et al., 2008] is to classify academic papers into different subjects. The dataset contains bag-of-words features for each document (node) and citation links (edges) between documents. (II) On Reddit [Hamilton et al., 2017], the task is to predict the community to which different Reddit posts belong. Reddit is an online discussion forum where users comment in different topical communities. Two posts (nodes) are connected if some user commented on both posts. The dataset contains word vectors as node features. (III) For protein-protein interaction networks (PPI) [Hamilton et al., 2017], the task is to classify protein functions. PPI consists of 24 graphs, each corresponds to a human tissue. Each node has positional gene sets, motif gene sets and immunological signatures as features and gene ontology sets as labels. 20 graphs are used for training, 2 graphs are used for validation and the rest for testing. Statistics of the datasets are summarized in Table 3.1.

Settings. In the *transductive setting*, we are only allowed to access a subset of nodes in one graph as training data, and validate/test on others. Our experiments on Citeseer, Cora and Reddit are transductive. In the *inductive setting*, we use a number of full graphs as training data and use other completely unseen graphs as validation/testing data. Our experiments on PPI are inductive.

We compare against three baselines: Graph Convolutional Networks (GCN) [Kipf and Welling, 2017], GraphSAGE [Hamilton et al., 2017] and Graph Attention Networks (GAT) [Velickovic et al., 2018].

Dataset	Nodes	Edges	Classes	Features
Citeseer	3,327	4,732	6	3,703
Cora	2,708	5,429	7	1,433
Reddit	232,965	avg deg 492	50	300
PPI	56,944	818,716	121	50

Table 3.1: Dataset statistics

Citeseer & Cora

For experiments on Citeseer and Cora, we choose GCN as the base model since on our data split, it is outperforming GAT. We construct JK-Nets by choosing MaxPooling (JK-MaxPool), Concatenation (JK-Concat), or LSTM-attention (JK-LSTM) as final aggregation layer. When taking the final aggregation, besides normal graph convolutional layers, we also take the first linear-transformed representation into account. The final prediction is done via

Model	Citeseer	Model	Cora
GCN (2)	77.3 (1.3)	GCN (2)	88.2 (0.7)
GAT (2)	76.2 (0.8)	GAT (3)	87.7 (0.3)
JK-MaxPool (1)	77.7 (0.5)	JK-Maxpool (6)	**89.6** (0.5)
JK-Concat (1)	**78.3** (0.8)	JK-Concat (6)	89.1 (1.1)
JK-LSTM (2)	74.7 (0.9)	JK-LSTM (1)	85.8 (1.0)

Table 3.2: Results of GCN-based JK-Nets on Citeseer and Cora. The baselines are GCN and GAT. The number in parentheses next to the model name indicates the best-performing number of layers among 1 to 6. Accuracy and standard deviation are computed from 3 random data splits.

a fully connected layer on top of the final aggregated representation. We split nodes in each graph into 60%, 20% and 20% for training, validation and testing. We vary the number of layers from 1 to 6 for each model and choose the best performing model with respect to the validation set. Throughout the experiments, we use the Adam optimizer [Kingma and Ba, 2015] with learning rate 0.005. We fix the dropout rate to be 0.5, the dimension of hidden features to be within $\{16, 32\}$, and add an $L2$ regularization of 0.0005 on model parameters. The results are shown in Table 3.2.

Results. We observe in Table 3.2 that JK-Nets outperform both GCN and GAT baselines in terms of prediction accuracy. Though JK-Nets perform well in general, there is no consistent winner and performance varies slightly across datasets.

Taking a closer look at results on Cora, both GCN and GAT achieve their best accuracies with only 2 or 3 layers, suggesting that local information is a stronger signal for classification than global ones. However, the fact that JK-Nets achieve the best performance with 6 layers indicates that global together with local information will help boost performance. This is where models like JK-Nets can be particularly beneficial. LSTM-attention may not be suitable for such small graphs because of its relatively high complexity.

Reddit

The Reddit data is too large to be handled well by current implementations of GCN or GAT. Hence, we use the more scalable GraphSAGE as the base model for JK-Net. It has skip connections and different modes of node aggregation. We experiment with Mean and

JK / Node	GraphSAGE	Maxpool	Concat	LSTM
Mean	0.950	0.953	0.955	0.950
MaxPool	0.948	0.924	**0.965**	0.877

Table 3.3: Results of GraphSAGE-based JK-Nets on Reddit. The baseline is GraphSAGE. Model performance is measured in Micro-F1 score. Each column shows the results of a JK-Net variant. For all models, the number of layers is fixed to 2.

JK / Node	SAGE	MaxPool	Concat	LSTM
Mean (10 epochs)	0.644	0.658	0.667	**0.721**
Mean (30 epochs)	0.690	0.713	0.694	**0.818**
MaxPool (10 epochs)	0.668	0.671	0.687	0.621*

Table 3.4: Results of GraphSAGE-based JK-Net on the PPI data. The baseline is GraphSAGE (SAGE). Each column, excluding SAGE, represents a JK-Net with different layer aggregation. All models use 3 layers, except for those with "*", whose number of layers is set to 2 due to GPU memory constraints. 0.6 is the corresponding 2-layer GraphSAGE performance.

MaxPool node aggregators, which take mean and max-pooling of a *linear transformation* of representations of the sampled neighbors. Combining each of GraphSAGE modes with MaxPooling, Concatenation or LSTM-attention as the last aggregation layer gives 6 JK-Net variants. We follow exactly the same setting of GraphSAGE as in the original paper [Hamilton et al., 2017], where the model consists of 2 hidden layers, each with 128 hidden units and is trained with Adam with learning rate of 0.01 and no weight decay. Results are shown in Table 3.3.

Results. With MaxPool as node aggregator and Concat as layer aggregator, JK-Net achieves the best Micro-F1 score among GarphSAGE and JK-Net variants. Note that the original GraphSAGE already performs fairly well with a Micro-F1 of 0.95. JK-Net reduces the error by 30%. The communities in the Reddit dataset were explicitly chosen from the well-behaved middle-sized communities to avoid the noisy cores and tree-like small communities [Hamilton et al., 2017]. As a result, this graph is more regular than the original Reddit data, and hence not exhibit the problems of varying subgraph structures. In such a case, the added flexibility of the node-specific neighborhood choices may not be as

Model	PPI
MLP	0.422
GAT	0.968 (0.002)
JK-Concat (2)	0.959 (0.003)
JK-LSTM (3)	0.969 (0.006)
JK-Dense-Concat (2)*	0.956 (0.004)
JK-Dense-LSTM (2)*	**0.976** (0.007)

Table 3.5: Micro-F1 scores of GAT-based JK-Nets on the PPI data. The baselines are GAT and MLP (Multilayer Perceptron). While the number of layers for JK-Concat and JK-LSTM are chosen from $\{2, 3\}$, the ones for JK-Dense-Concat and JK-Dense-LSTM are directly set to 2 due to GPU memory constraints.

relevant, and the stabilizing properties of concatenation instead come into play.

Protein-to-protein Interaction (PPI)

We demonstrate the power of adaptive JK-Nets, e.g., JK-LSTM, with experiments on the PPI data, where the subgraphs have more diverse and complex structures than those in the Reddit community detection dataset. We use both GraphSAGE and GAT as base models for JK-Net. The implementation of GraphSAGE and GAT are quite different: GraphSAGE is sample-based, where neighbors of a node are sampled to be a fixed number, while GAT considers all neighbors. Such differences cause large gaps in terms of both scalability and performances. Given that GraphSAGE scales to much larger graphs, it appears particularly valuable to evaluate how much JK-Net can improve upon GraphSAGE.

For GraphSAGE we follow the setup as in the Reddit experiments, except that we use 3 layers when possible, and compare the performance after 10 and 30 epochs of training. The results are shown in Table 3.4. For GAT and its JK-Net variants we stack two hidden layers with 4 attention heads computing 256 features (for a total of 1024 features), and a final prediction layer with 6 attention heads computing 121 features each. They are further averaged and input into sigmoid activations. We employ skip connections across intermediate attentional layers. These models are trained with Batch-size 2 and Adam optimizer with learning rate of 0.005. The results are shown in Table 3.5.

Results. JK-Nets with the LSTM-attention aggregators outperform the non-adaptive models GraphSAGE, GAT and JK-Nets with concatenation aggregators. In particular, JK-LSTM outperforms GraphSAGE by 0.128 in terms of micro-F1 score after 30 epochs of training. Structure-aware node adaptive models are especially beneficial on such complex graphs with diverse structures.

3.5 Discussion

We studied the first fundamental question of representation learning of graphs. Motivated by observations that reveal great differences in neighborhood information ranges for graph node embeddings in different graph structures, we propose a more flexible and powerful architecture – Jumping Knowledge Networks (JK-Nets) for GNNs that can adapt neigborhood ranges to nodes individually by exploiting the subgraph structures. This JK-network can improve representations in particular for graphs that have subgraphs of diverse local structure, and may hence not be well captured by fixed numbers of neighborhood aggregations. We also present an analysis framework for the influence distributions of GNNs, which relate the influence distributions of learned node representations to random walks and their neighborhood subgraph structures. Our model analysis provides insight into when a model would fail and how our model can help in these cases.

Interesting directions for future work include exploring other layer aggregators and studying the effect of the combination of various layer-wise and node-wise aggregators on different types of graph structures.

Since the release of this work, there have been many follow-up papers of the work that this chapter is based on. Many of these papers study how to make deeper GNNs perform better by extending the skip connections presented in this chapter to larger models, more refined connections, and better training techniques [Li et al., 2019, Chen et al., 2020a, Dehmamy et al., 2019, Klicpera et al., 2018, Li et al., 2020a]. Other approaches consider feature normalization and dropout techniques [Rong et al., 2020, Zhao and Akoglu, 2020]. In Chapter 10, we will see normalization techniques that improve the training of GNNs also implicitly help improve the issue presented in this chapter. More theoretical follow-up papers give a fine-grained analysis of the phenomenon in this chapter for ReLU activation [Oono and Suzuki, 2020a], and justify the optimization and generalization benefits of the multiscale skip connections as in JK-Nets [Oono and Suzuki, 2020b]. In Chapter 9, we will analyze the optimization dynamics of GNNs and give further justification for JK-Nets in terms of convergence rates to global minima.

Chapter 4

How Powerful are Graph Neural Networks

4.1 Introduction

This chapter addresses the second fundamental problem of representation of graphs. In the last few years, many GNN variants with different aggregation and graph-level readout have been proposed [Scarselli et al., 2009, Defferrard et al., 2016, Duvenaud et al., 2015, Hamilton et al., 2017, Kearnes et al., 2016, Kipf and Welling, 2017, Velickovic et al., 2018, Santoro et al., 2017, Xu et al., 2018, Ying et al., 2018, Zhang et al., 2018a]. Empirically, these GNNs have achieved state-of-the-art performance in many tasks such as node classification, link prediction, and graph classification. However, the design of new GNNs is mostly based on empirical intuition, heuristics, and experimental trial-and-error. There is little theoretical understanding of the properties and limitations of GNNs, and formal analysis of GNNs' representational capacity is limited.

We present a theoretical framework for analyzing the representational power of GNNs. We formally characterize how expressive different GNN variants are in learning to represent and distinguish between different graph structures. Our framework is inspired by the close connection between GNNs and the Weisfeiler-Lehman (WL) graph isomorphism test [Weisfeiler and Lehman, 1968], a powerful test known to distinguish a broad class of graphs [Babai and Kucera, 1979]. Similar to GNNs, the WL test iteratively updates a given node's feature vector by aggregating feature vectors of its network neighbors. What makes the WL test so powerful is its injective aggregation update that maps different node neighborhoods to different feature vectors. Our key insight is that a GNN can have as large discriminative power as the WL test if the GNN's aggregation scheme is highly expressive and can model injective functions.

To mathematically formalize the above insight, our framework first represents the set of feature vectors of a given node's neighbors as a *multiset*, i.e., a set with possibly repeating elements. Then, the neighbor aggregation in GNNs can be thought of as an *aggregation function over the multiset*. Hence, to have strong representational power, a GNN must be able to aggregate different multisets into different representations. We rigorously study several variants of multiset functions and theoretically characterize their discriminative power, i.e., how well different aggregation functions can distinguish different multisets. The more discriminative the multiset function is, the more powerful the representational power of the underlying GNN.

Our main results are summarized as follows:

1) We show that GNNs are *at most* as powerful as the WL test in distinguishing graph structures.

2) We establish conditions on the neighbor aggregation and graph readout functions under which the resulting GNN is *as powerful as* the WL test.

3) We identify graph structures that cannot be distinguished by popular GNN variants, such as GCN [Kipf and Welling, 2017] and GraphSAGE [Hamilton et al., 2017], and we precisely characterize the kinds of graph structures such GNN-based models can capture.

4) We develop a simple neural architecture, *Graph Isomorphism Network (GIN)*, and show that its discriminative/representational power is equal to the power of the WL test.

We validate our theory via experiments on graph classification datasets, where the expressive power of GNNs is crucial to capture graph structures. In particular, we compare the performance of GNNs with various aggregation functions. Our results confirm that the most powerful GNN by our theory, i.e., Graph Isomorphism Network (GIN), also empirically has high representational power as it almost perfectly fits the training data, whereas the less powerful GNN variants often severely underfit the training data. In addition, the representationally more powerful GNNs outperform the others by test set accuracy and achieve state-of-the-art performance on many graph classification benchmarks.

Graph isomorphism test. The graph isomorphism problem asks whether two graphs are topologically identical. This is a challenging problem: no polynomial-time algorithm is known for it yet [Garey, 1979, Garey and Johnson, 2002, Babai, 2016]. Apart from some corner cases [Cai et al., 1992], the Weisfeiler-Lehman (WL) test of graph isomorphism [Weisfeiler and Lehman, 1968] is an effective and computationally efficient test that distinguishes a broad class of graphs [Babai and Kucera, 1979]. Its 1-dimensional form, "naïve vertex refinement", is analogous to neighbor aggregation in GNNs. The WL test iteratively (1) aggregates the labels of nodes and their neighborhoods, and (2) hashes the aggregated labels into *unique* new labels. The algorithm decides that two graphs are non-isomorphic if at some iteration the labels of the nodes between the two graphs differ.

Based on the WL test, Shervashidze et al. [2011] proposed the WL subtree kernel that measures the similarity between graphs. The kernel uses the counts of node labels at different iterations of the WL test as the feature vector of a graph. Intuitively, a node's label at the k-th iteration of WL test represents a subtree structure of height k rooted at the node (Figure 4-1). Thus, the graph features considered by the WL subtree kernel are essentially counts of different rooted subtrees in the graph.

4.2 Theoretical Framework of Expressive Power

We start with an overview of our framework for analyzing the expressive power of GNNs. Figure 4-1 illustrates our idea. A GNN recursively updates each node's feature vector to capture the network structure and features of other nodes around it, *i.e.*, its rooted subtree structures (Figure 4-1). Throughout the chapter, we assume node input features are from

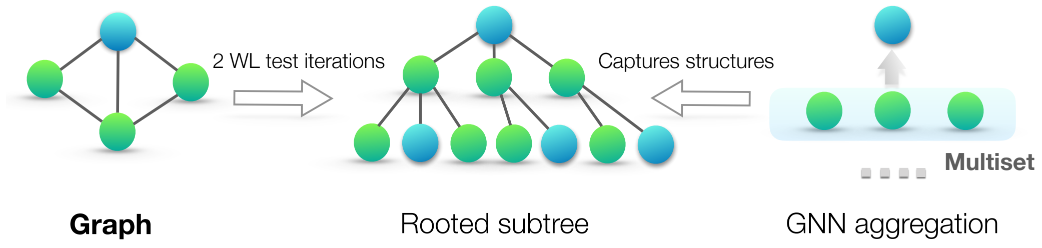


Figure 4-1: **Overview of our theoretical framework.** Middle panel: rooted subtree structures (at the blue node) that the WL test uses to distinguish different graphs. Right panel: if a GNN's aggregation function captures the *full multiset* of node neighbors, the GNN can capture the rooted subtrees in a recursive manner and be as powerful as the WL test.

a countable universe. For finite graphs, node feature vectors at deeper layers of any fixed model are also from a countable universe. For notational simplicity, we can assign each feature vector a unique label in $\{a, b, c \dots\}$. Then, feature vectors of a set of neighboring nodes form a *multiset* (Figure 4-1): the same element can appear multiple times since different nodes can have identical feature vectors.

Definition 4.1 (Multiset)**.** A multiset is a generalized concept of a set that allows multiple instances for its elements. More formally, a multiset is a 2-tuple $X = (S, m)$ where S is the *underlying set* of X that is formed from its *distinct elements*, and $m : S \to \mathbb{N}_{\geq 1}$ gives the *multiplicity* of the elements.

To study the representational power of a GNN, we analyze when a GNN maps two nodes to the same location in the embedding space. Intuitively, a maximally powerful GNN maps two nodes to the same location *only if* they have identical subtree structures with identical features on the corresponding nodes. Since subtree structures are defined recursively via node neighborhoods (Figure 4-1), we can reduce our analysis to the question whether a GNN maps two neighborhoods (*i.e.*, two multisets) to the same embedding or representation. A maximally powerful GNN would *never* map two different neighborhoods, *i.e.*, multisets of feature vectors, to the same representation. This means its aggregation scheme must be *injective*. Thus, we abstract a GNN's aggregation scheme as a class of functions over multisets that their neural networks can represent, and analyze whether they are able to represent injective multiset functions.

Next, we use this reasoning to develop a maximally powerful GNN. In Section 4.4, we study popular GNN variants and see that their aggregation schemes are inherently not injective and thus less powerful, but that they can capture other interesting properties of graphs.

4.3 Building Powerful Graph Neural Networks

First, we characterize the maximum representational capacity of a general class of GNN-based models. Ideally, a maximally powerful GNN could distinguish different graph structures by mapping them to different representations in the embedding space.

This ability to map any two different graphs to different embeddings, however, implies solving the challenging graph isomorphism problem. That is, we want isomorphic graphs to be mapped to the same representation and non-isomorphic ones to different representations. In our analysis, we characterize the representational capacity of GNNs via a slightly weaker criterion: a powerful heuristic called *Weisfeiler-Lehman (WL) graph isomorphism test*, that is known to work well in general, with a few exceptions, e.g., regular graphs [Cai et al., 1992, Douglas, 2011, Evdokimov and Ponomarenko, 1999].

Lemma 4.1. *(Upper bound of expressive power). Let G_1 and G_2 be any two non-isomorphic graphs. If a graph neural network $\mathcal{A} : \mathcal{G} \rightarrow \mathbb{R}^d$ maps G_1 and G_2 to different embeddings, the Weisfeiler-Lehman graph isomorphism test also decides G_1 and G_2 are not isomorphic.*

Proofs of all Lemmas and Theorems can be found in the Appendix. Hence, any aggregation-based GNN is at most as powerful as the WL test in distinguishing different graphs. A natural follow-up question is whether there exist GNNs that are, in principle, as powerful as the WL test? Our answer, in the following Theorem, is yes: if the neighbor aggregation and graph-level readout functions are injective, then the resulting GNN is as powerful as the WL test.

Theorem 4.2. *(Conditions for achieving maximal power). Let $\mathcal{A} : \mathcal{G} \rightarrow \mathbb{R}^d$ be a GNN. With a sufficient number of GNN layers, $\mathcal{A}$ maps any graphs G_1 and G_2 that the Weisfeiler-*

Lehman test of isomorphism decides as non-isomorphic, to different embeddings if the following conditions hold:

a) $\mathcal{A}$ *aggregates and updates node features iteratively with*

$$h_v^{(k)} = \phi\left(h_v^{(k-1)}, f\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)\right),$$

where the functions f*, which operates on multisets, and* ϕ *are injective.*

b) $\mathcal{A}$*'s graph-level readout, which operates on the multiset of node features* $\left\{h_v^{(k)}\right\}$*, is injective.*

We prove Theorem 4.2 in the appendix. For countable sets, injectiveness well characterizes whether a function preserves the distinctness of inputs. Uncountable sets, where node features are continuous, need some further considerations. In addition, it would be interesting to characterize how close together the learned features lie in a function's image. We leave these questions for future work, and focus on the case where input node features are from a countable set (that can be a subset of an uncountable set such as $\mathbb{R}^n$).

Lemma 4.3. *(Countablility of hidden features). Assume the input feature space* $\mathcal{X}$ *is countable. Let* $g^{(k)}$ *be the function parameterized by a GNN's* k*-th layer for* $k = 1, ..., L$*, where* $g^{(1)}$ *is defined on multisets* $X \subset \mathcal{X}$ *of bounded size. The range of* $g^{(k)}$, i.e., *the space of node hidden features* $h_v^{(k)}$*, is also countable for all* $k = 1, ..., L$*.*

Here, it is also worth discussing an important benefit of GNNs beyond distinguishing different graphs, that is, capturing similarity of graph structures. Note that node feature vectors in the WL test are essentially one-hot encodings and thus cannot capture the similarity between subtrees. In contrast, a GNN satisfying the criteria in Theorem 3 generalizes the WL test by *learning to embed* the subtrees to low-dimensional space. This enables GNNs to not only discriminate different structures, but also to learn to map similar graph structures to similar embeddings and capture dependencies between graph structures. Capturing structural similarity of the node labels is shown to be helpful for generalization particularly when the co-occurrence of subtrees is sparse across different graphs or there are noisy edges and node features [Yanardag and Vishwanathan, 2015].

4.3.1 Graph Isomorphism Network (GIN)

Having developed conditions for a maximally powerful GNN, we next develop a simple architecture, *Graph Isomorphism Network (GIN)*, that provably satisfies the conditions in Theorem 4.2. This model generalizes the WL test and hence achieves maximum discriminative power among GNNs.

To model injective multiset functions for the neighbor aggregation, we develop a theory of "deep multisets", *i.e.*, parameterizing universal multiset functions with neural networks. Our next lemma states that sum aggregators can represent injective, in fact, *universal* functions over multisets.

Lemma 4.4. *Assume $\mathcal{X}$ is countable. There exists a function $f : \mathcal{X} \to \mathbb{R}^n$ so that $h(X) = \sum_{x \in X} f(x)$ is unique for each multiset $X \subset \mathcal{X}$ of bounded size. Moreover, any multiset function g can be decomposed as $g(X) = \phi\left(\sum_{x \in X} f(x)\right)$ for some function ϕ.*

We prove Lemma 4.4 in the appendix. The proof extends the setting in [Zaheer et al., 2017] from sets to multisets. An important distinction between deep multisets and sets is that certain popular injective set functions, such as the mean aggregator, are not injective multiset functions. With the mechanism for modeling universal multiset functions in Lemma 4.4 as a building block, we can conceive aggregation schemes that can represent universal functions over a node and the multiset of its neighbors, and thus will satisfy the injectiveness condition (a) in Theorem 4.2. Our next corollary provides a simple and concrete formulation among many such aggregation schemes.

Corollary 4.5. *(Universal aggregation). Assume $\mathcal{X}$ is countable. There exists a function $f : \mathcal{X} \to \mathbb{R}^n$ so that for infinitely many choices of ϵ, including all irrational numbers, $h(c, X) = (1 + \epsilon) \cdot f(c) + \sum_{x \in X} f(x)$ is unique for each pair (c, X), where $c \in \mathcal{X}$ and $X \subset \mathcal{X}$ is a multiset of bounded size. Moreover, any function g over such pairs can be decomposed as $g(c, X) = \varphi\left((1 + \epsilon) \cdot f(c) + \sum_{x \in X} f(x)\right)$ for some function φ.*

We can use multi-layer perceptrons (MLPs) to model and learn f and φ in Corollary 4.5, thanks to the universal approximation theorem [Hornik et al., 1989, Hornik, 1991]. In practice, we model $f^{(k+1)} \circ \varphi^{(k)}$ with one MLP, because MLPs can represent the composition of functions. In the first iteration, we do not need MLPs before summation if input features are one-hot encodings as their summation alone is injective. We can make ϵ a learnable parameter or a fixed scalar. Then, GIN updates node representations as

$$h_v^{(k)} = \mathrm{MLP}^{(k)}\left(\left(1+\epsilon^{(k)}\right)\cdot h_v^{(k-1)} + \textstyle\sum_{u\in\mathcal{N}(v)} h_u^{(k-1)}\right). \tag{4.1}$$

Generally, there may exist many other powerful GNNs. GIN is one such example among many maximally powerful GNNs, while being simple.

4.3.2 Graph-level Readout of GIN

Node embeddings learned by GIN can be directly used for tasks like node classification and link prediction. For graph classification tasks we use the following "readout" function that, given embeddings of individual nodes, produces the embedding of the entire graph.

An important aspect of the graph-level readout is that node representations, corresponding to subtree structures, get more refined and global as the number of iterations increases. A sufficient number of iterations is key to achieving good discriminative power. Yet, features from earlier iterations may sometimes generalize better. To consider all structural information, we use information from all depths/iterations of the model. We achieve this by Jumping Knowledge Networks (JK-Nets) [Xu et al., 2018] introduced in the previous chapter, where we replace the graph-level readout using the final node representations with graph representations concatenated across *all iterations/layers* of GIN:

$$h_G = \mathrm{CONCAT}\Big(\mathrm{READOUT}\Big(\left\{h_v^{(k)}|v\in G\right\}\Big)\ \Big|\ k=0,1,\ldots,K\Big). \tag{4.2}$$

By Theorem 4.2 and Corollary 4.5, if GIN replaces READOUT in 4.2 with summing all node features from the same iterations (we do not need an extra MLP before summation for the same reason as in 4.1), it provably generalizes the WL test and the WL subtree kernel.

4.4 Less Powerful but Still Interesting GNNs

Next, we study GNNs that do not satisfy the conditions in Theorem 4.2, including GCN [Kipf and Welling, 2017] and GraphSAGE [Hamilton et al., 2017]. We conduct ablation studies on two aspects of the aggregator in 4.1: (1) 1-layer perceptrons instead of MLPs and (2) mean or max-pooling instead of the sum. We will see that these GNN variants get confused by surprisingly simple graphs and are less powerful than the WL test. Nonetheless, models with mean aggregators like GCN perform well for *node classification* tasks. To better understand this, we precisely characterize what different GNN variants can and cannot capture about a graph and discuss the implications for learning with graphs.

4.4.1 One-layer Perceptrons are not Sufficient

The function f in Lemma 4.4 helps map distinct multisets to unique embeddings. It can be parameterized by an MLP by the universal approximation theorem [Hornik, 1991]. Nonetheless, many existing GNNs instead use a 1-layer perceptron $\sigma \circ W$ [Duvenaud et al., 2015, Kipf and Welling, 2017, Zhang et al., 2018a], a linear mapping followed by a non-linear activation function such as a ReLU. Such 1-layer mappings are examples of Generalized Linear Models [Nelder and Wedderburn, 1972]. Therefore, we are interested in understanding whether 1-layer perceptrons are enough for graph learning. Lemma 4.6 suggests that there are indeed network neighborhoods (multisets) that models with 1-layer perceptrons can never distinguish.

Lemma 4.6. *There exist finite multisets $X_1 \neq X_2$ so that for any linear mapping W, $\sum_{x \in X_1} \mathrm{ReLU}\left(Wx\right) = \sum_{x \in X_2} \mathrm{ReLU}\left(Wx\right)$.*

The main idea of the proof for Lemma 4.6 is that 1-layer perceptrons can behave much like linear mappings, so the GNN layers degenerate into simply summing over neighborhood features. Our proof builds on the fact that the bias term is lacking in the linear mapping. With the bias term and sufficiently large output dimensionality, 1-layer perceptrons might be able to distinguish different multisets. Nonetheless, unlike models using MLPs, the 1-layer perceptron (even with the bias term) is *not a universal approximator* of multiset functions.

Consequently, even if GNNs with 1-layer perceptrons can embed different graphs to different locations to some degree, such embeddings may not adequately capture structural similarity, and can be difficult for simple classifiers, e.g., linear classifiers, to fit. In Section F.3, we will empirically see that GNNs with 1-layer perceptrons, when applied to graph classification, sometimes severely underfit training data and often perform worse than GNNs with MLPs in terms of test accuracy.

4.4.2 Structures that Confuse Less Powerful GNNs

What happens if we replace the sum in $h\left(X\right) = \sum_{x \in X} f(x)$ with mean or max-pooling as in GCN and GraphSAGE? Mean and max-pooling aggregators are still well-defined multiset functions because they are permutation invariant. But, they are *not* injective. Figure 4-2 ranks the three aggregators by their representational power, and Figure 4-3 illustrates pairs of structures that the mean and max-pooling aggregators fail to distinguish. Here, node colors denote different node features, and we assume the GNNs aggregate neighbors first before combining them with the central node labeled as v and v'.

In Figure 4-3a, every node has the same feature a and $f(a)$ is the same across all nodes (for any function f). When performing neighborhood aggregation, the mean or maximum over $f(a)$ remains $f(a)$ and, by induction, we always obtain the same node representation everywhere. Thus, in this case mean and max-pooling aggregators fail to capture any structural information. In contrast, the sum aggregator distinguishes the structures because $2 \cdot f(a)$ and $3 \cdot f(a)$ give different values. The same argument can be applied to any unlabeled graph. If node degrees instead of a constant value is used as node input features, in principle, mean can recover sum, but max-pooling cannot.

Fig. 4-3a suggests that mean and max have trouble distinguishing graphs with nodes that have repeating features. Let h_{color} (r for red, g for green) denote node features transformed by f. Fig. 4-3b shows that maximum over the neighborhood of the blue nodes v and v' yields $\max\left(h_{\text{g}}, h_{\text{r}}\right)$ and $\max\left(h_{\text{g}}, h_{\text{r}}, h_{\text{r}}\right)$, which collapse to the same representation (even though the corresponding graph structures are different). Thus, max-pooling fails to distinguish them. In contrast, the sum aggregator still works because $\frac{1}{2}\left(h_{\text{g}} + h_{\text{r}}\right)$ and $\frac{1}{3}\left(h_{\text{g}} + h_{\text{r}} + h_{\text{r}}\right)$

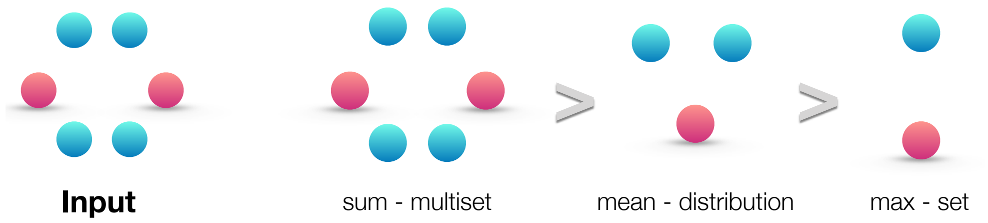
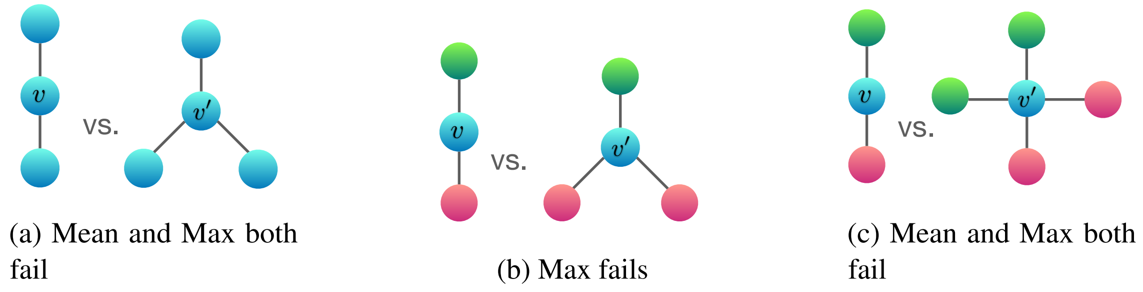


Figure 4-2: **Ranking by expressive power for sum, mean and max aggregators over a multiset**. Left panel shows the input multiset, i.e., the network neighborhood to be aggregated. The next three panels illustrate the aspects of the multiset a given aggregator is able to capture: sum captures the full multiset, mean captures the proportion/distribution of elements of a given type, and the max aggregator ignores multiplicities (reduces the multiset to a simple set).



(a) Mean and Max both fail

(b) Max fails

(c) Mean and Max both fail

Figure 4-3: **Examples of graph structures that mean and max aggregators fail to distinguish.** Between the two graphs, nodes v and v' get the same embedding even though their corresponding graph structures differ. Figure 4-2 gives reasoning about how different aggregators "compress" different multisets and thus fail to distinguish them.

are in general not equivalent. Similarly, in Fig. 4-3c, both mean and max fail as $\frac{1}{2}\left(h_{\mathrm{g}}+h_{\mathrm{r}}\right)=\frac{1}{4}\left(h_{\mathrm{g}}+h_{\mathrm{g}}+h_{\mathrm{r}}+h_{\mathrm{r}}\right)$.

To characterize the class of multisets that the mean aggregator can distinguish, consider the example $X_1=(S,m)$ and $X_2=(S,k\cdot m)$, where X_1 and X_2 have the same set of distinct elements, but X_2 contains k copies of each element of X_1. Any mean aggregator maps X_1 and X_2 to the same embedding, because it simply takes averages over individual element features. Thus, the mean captures the *distribution* (proportions) of elements in a multiset, but not the *exact* multiset.

Corollary 4.7. *Assume $\mathcal{X}$ is countable. There exists a function $f:\mathcal{X}\to\mathbb{R}^n$ so that for $h(X)=\frac{1}{|X|}\sum_{x\in X}f(x)$, $h(X_1)=h(X_2)$ if and only if multisets X_1 and X_2 have the same distribution. That is, assuming $|X_2|\geq|X_1|$, we have $X_1=(S,m)$ and $X_2=(S,k\cdot m)$ for some $k\in\mathbb{N}_{\geq 1}$.*

The mean aggregator may perform well if, for the task, the statistical and distributional information in the graph is more important than the exact structure. Moreover, when node features are diverse and rarely repeat, the mean aggregator is as powerful as the sum aggregator. This may explain why, despite the limitations identified in Section 4.4.2, GNNs with mean aggregators are effective for node classification tasks, such as classifying article subjects and community detection, where node features are rich and the distribution of the neighborhood features provides a strong signal for the task.

The examples in Figure 4-3 illustrate that max-pooling considers multiple nodes with the same feature as *only one* node (i.e., treats a multiset as a set). Max-pooling captures neither the exact structure nor the distribution. However, it may be suitable for tasks where it is important to identify representative elements or the "skeleton", rather than to distinguish the exact structure or distribution. Qi et al. [2017] empirically show that the max-pooling aggregator learns to identify the skeleton of a 3D point cloud and that it is robust to noise and outliers. Later in Chapter 8, we shall also see that the inductive biases of max may be useful for learning some graph algorithms. For completeness, the next corollary shows that the max-pooling aggregator captures the underlying set of a multiset.

Corollary 4.8. *Assume $\mathcal{X}$ is countable. Then there exists a function $f : \mathcal{X} \to \mathbb{R}^\infty$ so that for $h(X) = \max_{x \in X} f(x)$, $h(X_1) = h(X_2)$ if and only if X_1 and X_2 have the same underlying set.*

Remarks on other aggregators There are other non-standard neighbor aggregation schemes that we do not cover, e.g., weighted average via attention [Velickovic et al., 2018] and LSTM pooling [Hamilton et al., 2017]. We emphasize that our theoretical framework is general enough to characterize the representaional power of any aggregation-based GNNs. In the future, it would be interesting to apply our framework to analyze and understand other aggregation schemes.

4.5 Experiments

We evaluate and compare the training and test performance of GIN and less powerful GNN variants. Training set performance allows us to compare different GNN models based on their representational power and test set performance quantifies generalization ability.

Datasets. We use 9 graph classification benchmarks: 4 bioinformatics datasets (MUTAG, PTC, NCI1, PROTEINS) and 5 social network datasets (COLLAB, IMDB-BINARY, IMDB-MULTI, REDDIT-BINARY and REDDIT-MULTI5K) [Yanardag and Vishwanathan, 2015]. Importantly, our goal here is not to allow the models to rely on the input node features but mainly learn from the network structure. Thus, in the bioinformatic graphs, the nodes have categorical input features but in the social networks, they have no features. For social networks we create node features as follows: for the REDDIT datasets, we set all node feature vectors to be the same (thus, features here are uninformative); for the other social graphs, we use one-hot encodings of node degrees. Dataset statistics are summarized in Table 5.1, and more details of the data can be found in Appendix B.2.

Models and configurations. We evaluate GINs ((4.1) and (4.2)) and the less powerful GNN variants. Under the GIN framework, we consider two variants: (1) a GIN that learns ϵ

in 4.1 by gradient descent, which we call GIN-ϵ, and (2) a simpler (slightly less powerful)[1] GIN, where ϵ in (4.1) is fixed to 0, which we call GIN-0. As we will see, GIN-0 shows strong empirical performance: not only does GIN-0 fit training data equally well as GIN-ϵ, it also demonstrates good generalization, slightly but consistently outperforming GIN-ϵ in terms of test accuracy. For the less powerful GNN variants, we consider architectures that replace the sum in the GIN-0 aggregation with mean or max-pooling[2], or replace MLPs with 1-layer perceptrons, a linear mapping followed by ReLU. In Figure 4-4 and Table 4.1, a model is named by the aggregator/perceptron it uses. Here mean–1-layer and max–1-layer correspond to GCN and GraphSAGE, respectively, up to minor architecture modifications. We apply the same graph-level readout (READOUT in 4.2) for GINs and all the GNN variants, specifically, sum readout on bioinformatics datasets and mean readout on social datasets due to better test performance.

Following [Yanardag and Vishwanathan, 2015, Niepert et al., 2016], we perform 10-fold cross-validation with LIB-SVM [Chang and Lin, 2011]. We report the average and standard deviation of validation accuracies across the 10 folds within the cross-validation. For all configurations, 5 GNN layers (including the input layer) are applied, and all MLPs have 2 layers. Batch normalization [Ioffe and Szegedy, 2015] is applied on every hidden layer. We use the Adam optimizer [Kingma and Ba, 2015] with initial learning rate 0.01 and decay the learning rate by 0.5 every 50 epochs. The hyper-parameters we tune for each dataset are: (1) the number of hidden units $\in \{16, 32\}$ for bioinformatics graphs and 64 for social graphs; (2) the batch size $\in \{32, 128\}$; (3) the dropout ratio $\in \{0, 0.5\}$ after the dense layer [Srivastava et al., 2014]; (4) the number of epochs, *i.e.*, a single epoch with the best cross-validation accuracy averaged over the 10 folds was selected. Note that due to the small dataset sizes, an alternative setting, where hyper-parameter selection is done using a validation set, is extremely unstable, *e.g.*, for MUTAG, the validation set only contains 18 data points. We also report the training accuracy of different GNNs, where all the hyper-parameters were fixed across the datasets: 5 GNN layers (including the input layer), hidden units of size 64, minibatch of size 128, and 0.5 dropout ratio. For comparison, the training accuracy

[1] There exist certain (somewhat contrived) graphs that GIN-ϵ can distinguish but GIN-0 cannot.

[2] For REDDIT-BINARY, REDDIT–MULTI5K, and COLLAB, we did not run experiments for max-pooling due to GPU memory constraints.

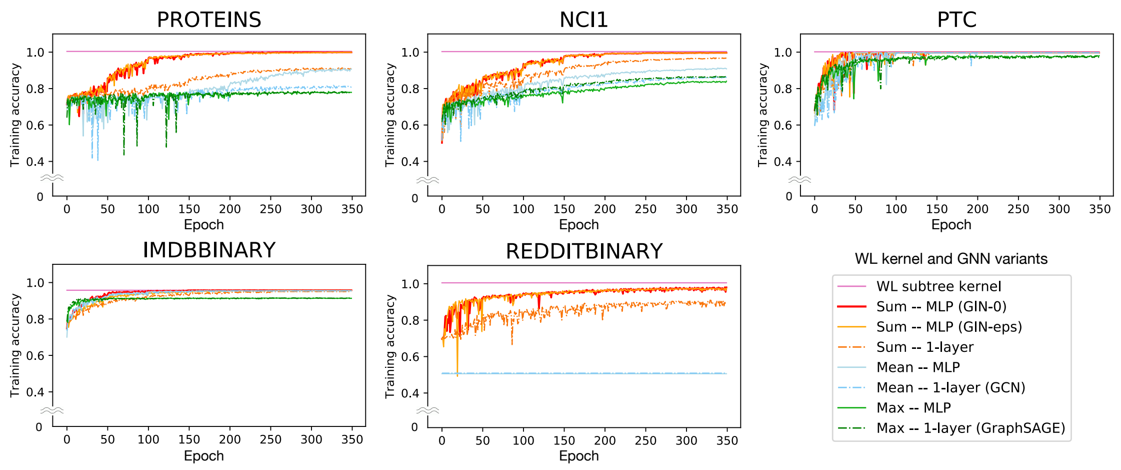


Figure 4-4: **Training set performance** of GIN, less powerful GNN variants, and the WL subtree kernel.

of the WL subtree kernel is reported, where we set the number of iterations to 4, which is comparable to the 5 GNN layers.

Baselines. We compare the GNNs above with a number of state-of-the-art baselines for graph classification: (1) the WL subtree kernel [Shervashidze et al., 2011], where C-SVM [Chang and Lin, 2011] was used as a classifier; the hyper-parameters we tune are C of the SVM and the number of WL iterations $\in \{1, 2, \dots, 6\}$; (2) state-of-the-art deep learning architectures, i.e., Diffusion-convolutional neural networks (DCNN) [Atwood and Towsley, 2016], PATCHY-SAN [Niepert et al., 2016] and Deep Graph CNN (DGCNN) [Zhang et al., 2018a]; (3) Anonymous Walk Embeddings (AWL) [Ivanov and Burnaev, 2018]. For the deep learning methods and AWL, we report the accuracies reported in the original papers.

4.5.1 Results

Training set performance. We validate our theoretical analysis of the representational power of GNNs by comparing their training accuracies. Models with higher representational power should have higher training set accuracy. Figure 4-4 shows training curves of GINs and less powerful GNN variants with the same hyper-parameter settings. First, both the theoretically most powerful GNN, *i.e.* GIN-ϵ and GIN-0, are able to almost perfectly fit all the training sets. In our experiments, explicit learning of ϵ in GIN-ϵ yields no gain in fitting

	Datasets	IMDB-B	IMDB-M	RDT-B	RDT-M5K	COLLAB	MUTAG	PROTEINS	PTC	NCI1
Datasets	# graphs	1000	1500	2000	5000	5000	188	1113	344	4110
	# classes	2	3	2	5	3	2	2	2	2
	Avg # nodes	19.8	13.0	429.6	508.5	74.5	17.9	39.1	25.5	29.8
Baselines	WL subtree	73.8 ± 3.9	50.9 ± 3.8	81.0 ± 3.1	52.5 ± 2.1	78.9 ± 1.9	90.4 ± 5.7	75.0 ± 3.1	59.9 ± 4.3	**86.0 ± 1.8** *
	DCNN	49.1	33.5	–	–	52.1	67.0	61.3	56.6	62.6
	PATCHYSAN	71.0 ± 2.2	45.2 ± 2.8	86.3 ± 1.6	49.1 ± 0.7	72.6 ± 2.2	**92.6 ± 4.2** *	75.9 ± 2.8	60.0 ± 4.8	78.6 ± 1.9
	DGCNN	70.0	47.8	–	–	73.7	85.8	75.5	58.6	74.4
	AWL	74.5 ± 5.9	51.5 ± 3.6	87.9 ± 2.5	54.7 ± 2.9	73.9 ± 1.9	87.9 ± 9.8	–	–	–
GNN variants	SUM–MLP (**GIN-0**)	**75.1 ± 5.1**	**52.3 ± 2.8**	**92.4 ± 2.5**	**57.5 ± 1.5**	**80.2 ± 1.9**	**89.4 ± 5.6**	**76.2 ± 2.8**	**64.6 ± 7.0**	**82.7 ± 1.7**
	SUM–MLP (**GIN-ϵ**)	**74.3 ± 5.1**	**52.1 ± 3.6**	**92.2 ± 2.3**	**57.0 ± 1.7**	**80.1 ± 1.9**	**89.0 ± 6.0**	**75.9 ± 3.8**	63.7 ± 8.2	**82.7 ± 1.6**
	SUM–1-LAYER	74.1 ± 5.0	**52.2 ± 2.4**	90.0 ± 2.7	55.1 ± 1.6	**80.6 ± 1.9**	**90.0 ± 8.8**	**76.2 ± 2.6**	63.1 ± 5.7	82.0 ± 1.5
	MEAN–MLP	73.7 ± 3.7	**52.3 ± 3.1**	50.0 ± 0.0	20.0 ± 0.0	79.2 ± 2.3	83.5 ± 6.3	75.5 ± 3.4	**66.6 ± 6.9**	80.9 ± 1.8
	MEAN–1-LAYER (GCN)	74.0 ± 3.4	51.9 ± 3.8	50.0 ± 0.0	20.0 ± 0.0	79.0 ± 1.8	85.6 ± 5.8	76.0 ± 3.2	64.2 ± 4.3	80.2 ± 2.0
	MAX–MLP	73.2 ± 5.8	51.1 ± 3.6	–	–	–	84.0 ± 6.1	76.0 ± 3.2	64.6 ± 10.2	77.8 ± 1.3
	MAX–1-LAYER (GraphSAGE)	72.3 ± 5.3	50.9 ± 2.2	–	–	–	85.1 ± 7.6	75.9 ± 3.2	63.9 ± 7.7	77.7 ± 1.5

Table 4.1: **Test set classification accuracies (%).** The best-performing GNNs are highlighted with boldface. On datasets where GINs' accuracy is not strictly the highest among GNN variants, we see that GINs are still comparable to the best GNN because a paired t-test at significance level 10% does not distinguish GINs from the best; thus, GINs are also highlighted with boldface. If a baseline performs significantly better than all GNNs, we highlight it with boldface and asterisk.

training data compared to fixing ϵ to 0 as in GIN-0. In comparison, the GNN variants using mean/max pooling or 1-layer perceptrons severely underfit on many datasets. In particular, the training accuracy patterns align with our ranking by the models' representational power: GNN variants with MLPs tend to have higher training accuracies than those with 1-layer perceptrons, and GNNs with sum aggregators tend to fit the training sets better than those with mean and max-pooling aggregators.

On our datasets, training accuracies of the GNNs never exceed those of the WL subtree kernel. This is expected because GNNs generally have lower discriminative power than the WL test. For example, on IMDBBINARY, none of the models can perfectly fit the training set, and the GNNs achieve at most the same training accuracy as the WL kernel. This pattern aligns with our result that the WL test provides an upper bound for the representational capacity of the aggregation-based GNNs. However, the WL kernel is not able to learn how to combine node features, which might be quite informative for a given prediction task as we will see next.

Test set performance. Next, we compare test accuracies. Although our theoretical results do not directly speak about the generalization ability of GNNs, it is reasonable to expect that GNNs with strong expressive power can accurately capture graph structures of interest and thus generalize well. Table 4.1 compares test accuracies of GINs (Sum–MLP), other GNN variants, as well as the state-of-the-art baselines.

First, GINs, especially GIN-0, outperform (or achieve comparable performance as) the less powerful GNN variants on all the 9 datasets, achieving state-of-the-art performance. GINs shine on the social network datasets, which contain a relatively large number of training graphs. For the Reddit datasets, all nodes share the same scalar as node feature. Here, GINs and sum-aggregation GNNs accurately capture the graph structure and significantly outperform other models. Mean-aggregation GNNs, however, fail to capture any structures of the unlabeled graphs (as predicted in Section 4.4.2) and do not perform better than random guessing. Even if node degrees are provided as input features, mean-based GNNs perform much worse than sum-based GNNs (the accuracy of the GNN with mean–MLP aggregation is $71.2\pm4.6\%$ on REDDIT-BINARY and $41.3\pm2.1\%$ on REDDIT-MULTI5K). Comparing GINs (GIN-0 and GIN-ϵ), we observe that GIN-0 slightly but consistently outperforms GIN-ϵ. Since both models fit training data equally well, the better generalization of GIN-0 may be explained by its simplicity compared to GIN-ϵ.

4.6 Conclusion

In this chapter, we developed theoretical foundations for reasoning about the expressive power of GNNs, and proved tight bounds on the representational capacity of popular GNN variants. We also designed a provably maximally powerful GNN under the message passing framework. An interesting direction for future work is to go beyond message passing in order to pursue possibly more powerful architectures for learning with graphs.

Since the release of this work, there have been many follow-up papers of the work that chapter is based on. We may approximately classify these into a few approaches for designing GNNs with more expressive power. First, our analysis assumed the node features do not contain any special number that identifies a node, because such identifiers do not easily extend to unseen graphs and thus isomorphic graphs may be mapped to different representations. A few works explore adding auxiliary node identifiers and empirically show that random features are useful in practice [Sato et al., 2019, 2020, Vignac et al., 2020]. In general, this approach can result in very powerful GNNs but it also requires a careful treatment of the node identifiers for good generalization performance. The second approach considers the specific application domains, such as logic, chemistry, and social networks, and exploit additional structure and information in these domains for more expressive GNNs [Barceló et al., 2020, Zhang et al., 2019c, You et al., 2019, Klicpera et al., 2020]. The third approach uses higher-order GNNs with computational structures that are more expressive but also significantly more expensive than the message passing scheme [Keriven and Peyré, 2019, Maron et al., 2019, Murphy et al., 2019, Tahmasebi and Jegelka, 2020, Chen et al., 2020b]. While these models are theoretically more expressive, it has been observed that their generalization performance is often not as good as simple message passing GNNs, and they are computationally intractable for large graphs. We believe this relates to the inductive biases of the architectures for generalization, which we shall discuss in Chapter 6 and Chapter 8. Finally, other follow-up works look at the expressive power of GNNs from the perspective of function approximation, counting substructures, and communication complexity [Chen et al., 2019, 2021, 2020b, Dehmamy et al., 2019, Garg et al., 2020, Loukas, 2020a,b].

To complete the picture, it would also be interesting to understand and improve the generalization properties of GNNs as well as better understand their optimization dynamics. In the remaining chapters of the thesis, we will study all these theoretical aspects of GNNs: generalization, extrapolation, and optimization.

Part II

Reasoning: Generalization

Chapter 5

Dynamics and Generalization of Over-parameterized GNNs

The second part of the thesis is about modeling intelligence for reasoning. Learning to reason implies learning to implement a reasoning process, within and outside the training distribution. While many neural architectures are able to represent these complex, structured reasoning processes, the solutions they find via stochastic gradient descent often do not generalize well to unseen situations. Hence, to understand how to build models for reasoning, we must understand what affects the generalization of neural networks, for both interpolation (in-distribution) and extrapolation (out of distribution) regimes. In general, this depends on the training algorithm, network structure, task structure, and training data. In Chapter 5, we start by taking into account the training algorithm, gradient descent, and training data. In Chapter 6, we further take into account the network structure and task structure. Both chapters focus on in-distribution generalization. In Part III, we will study extrapolation.

5.1 Introduction

In this chapter, we study the generalization properties of GNNs. In particular, we analyze the learning dynamics of GNNs trained by gradient descent, which then gives generalization bounds and provable learning of simple functions on graphs. Here, we will focus on learning simple and general functions, and do not yet take into account the task structure of reasoning, which we will do in the next chapter.

In general, the function that a neural network implements depends on the training. However, due to the *non-convex* nature of the training procedure, it has been often difficult to analyze the learned GNNs directly. For example, one may ask whether GNNs can provably learn certain class of functions. This question seems hard to answer given our limited theoretical understanding of deep learning.

Recent advances of deep learning theory establishes a connection between the over-parameterized neural networks and a specific neural tangent kernel [Arora et al., 2019b,c, Du et al., 2019a, Jacot et al., 2018]. Inspired by this connection, we show that over-parameterized GNNs trained by gradient descent are equivalent to *Graph Neural Tangent Kernel (GNTK)*, where the word "tangent" corresponds to the training algorithm — gradient descent. Moreover, when the width goes to *infinity*, we show analytic formulas for computing such infinitely wide GNNs via GNTK, exactly and efficiently.

First, we present a general recipe which translates a GNN architecture to its corresponding GNTK. This recipe works for a wide range of GNNs, including graph isomorphism network (GIN) [Xu et al., 2019], graph convolutional network (GCN) [Kipf and Welling, 2017], and GNN with jumping knowledge [Xu et al., 2018]. Second, we conduct a theoretical analysis of GNTK, and hence over-parameterized GNNs. We show for a broad range of smooth functions over graphs, simple GNNs can learn them with polynomial number of samples. To our knowledge, this is the first sample complexity analysis in the GNN literature. As a by-product, we empirically find such infinitely wide GNNs generalize well and outperform the finite GNNs on graph classification tasks.

Notations

We introduce several notations. In this chapter, we refer to the neighbor aggregation or message passing step in GNNs as a BLOCK operation.

BLOCK Operation. We denote the number of fully-connected layers in each BLOCK operation, i.e., the number of hidden layers of an MLP, by R.

When $R = 1$, the BLOCK operation can be formulated as

$$\text{BLOCK}^{(\ell)}(u) = \sqrt{\frac{c_\sigma}{m}} \cdot \sigma\left(\boldsymbol{W}_\ell \cdot c_u \sum_{v \in \mathcal{N}(u) \cup \{u\}} \boldsymbol{h}_v^{(\ell-1)}\right). \tag{5.1}$$

Here, $\boldsymbol{W}_\ell$ are learnable weights, initialized as Gaussian random variables. σ is an activation function like ReLU. m is the output dimension of $\boldsymbol{W}_\ell$. We set the scaling factor c_σ to 2, following the initialization scheme in He et al. [2015]. c_u is a scaling factor for neighbor aggregation. Different GNNs often have different choices for c_u. In Graph Convolution Network (GCN) [Kipf and Welling, 2017], $c_u = \frac{1}{|\mathcal{N}(u)|+1}$, and in Graph Isomorphism Network (GIN) [Xu et al., 2019], $c_u = 1$, which correspond to averaging and summing over neighbor features, respectively.

When the number of fully-connected layers $R = 2$, the BLOCK operation can be written as

$$\text{BLOCK}^{(\ell)}(u) = \sqrt{\frac{c_\sigma}{m}} \sigma\left(\boldsymbol{W}_{\ell,2} \sqrt{\frac{c_\sigma}{m}} \cdot \sigma\left(\boldsymbol{W}_{\ell,1} \cdot c_u \sum_{v \in \mathcal{N}(u) \cup \{u\}} \boldsymbol{h}_v^{(\ell-1)}\right)\right), \tag{5.2}$$

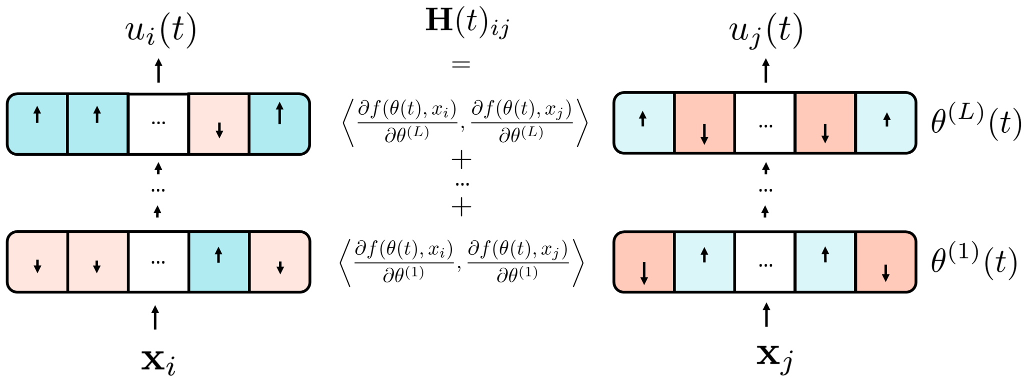
where $\boldsymbol{W}_{\ell,1}$ and $\boldsymbol{W}_{\ell,2}$ are learnable weights. BLOCK operations can be defined similarly for $R > 2$.

5.2 Graph Neural Tangent Kernel

In this section we present our general recipe which translates a GNN architecture to its corresponding GNTK. GNTK computes the kernel value, i.e., similarity, of a pair of graphs. We first provide some intuition on neural tangent kernels (NTKs). We refer readers to Jacot et al. [2018], Arora et al. [2019c] for more comprehensive descriptions.

5.2.1 Intuition of Formulas

Consider a general neural network $f(\theta, x) \in \mathbb{R}$ where $\theta \in \mathbb{R}^m$ is all the parameters in the network and x is the input. Given a training dataset $\{(x_i, y_i)_{i=1}^n\}$, consider training the



$u_i(t)$
$\mathbf{H}(t)_{ij}$
$=$
$\left\langle \frac{\partial f(\theta(t),x_i)}{\partial \theta^{(L)}}, \frac{\partial f(\theta(t),x_j)}{\partial \theta^{(L)}} \right\rangle$
$+$
$\cdots$
$+$
$\left\langle \frac{\partial f(\theta(t),x_i)}{\partial \theta^{(1)}}, \frac{\partial f(\theta(t),x_j)}{\partial \theta^{(1)}} \right\rangle$
$u_j(t)$
$\theta^{(L)}(t)$
$\theta^{(1)}(t)$
$\mathbf{x}_i$
$\mathbf{x}_j$

Figure 5-1: **Illustration of NTK theory.** Consider a general neural network with L layers $\theta^{(1)}, \ldots, \theta^{(L)}$, given input x_i and x_j, the neural network will output $f(\theta(t), x_i), f(\theta(t), x_j)$. When trained by gradient descent, evolution of $u(t)$ follows $\frac{du}{dt} = -\boldsymbol{H}(t)(u(t) - y)$. One can show that when the number of parameters in the neural network is large enough, and parameters of the neural network are initialized as Gaussian variables, $\boldsymbol{H}(t) \approx \boldsymbol{H}(0)$ and can be calculated analytically.

neural network by minimizing the squared loss over training data

$$\ell(\theta) = \frac{1}{2}\sum_{i=1}^{n}(f(\theta, x_i) - y_i)^2.$$

Suppose we minimize the squared loss $\ell(\theta)$ by gradient descent with infinitesimally small learning rate, i.e., $\frac{d\theta(t)}{dt} = -\nabla\ell(\theta(t))$. Let $u(t) = (f(\theta(t), x_i))_{i=1}^{n}$ be the network outputs. $u(t)$ follows the evolution

$$\frac{du}{dt} = -\boldsymbol{H}(t)(u(t) - y), \tag{5.3}$$

where

$$\boldsymbol{H}(t)_{ij} = \left\langle \frac{\partial f(\theta(t), x_i)}{\partial \theta}, \frac{\partial f(\theta(t), x_j)}{\partial \theta} \right\rangle \text{ for } (i, j) \in [n] \times [n]. \tag{5.4}$$

Recent advances in optimization of neural networks have shown, for sufficiently over-parameterized neural networks, the matrix $\boldsymbol{H}(t)$ keeps almost unchanged during the training process [Arora et al., 2019b,c, Du et al., 2019a, Jacot et al., 2018], in which case the training dynamics is identical to that of kernel regression. Moreover, under a random initialization of parameters, the random matrix $\boldsymbol{H}(0)$ converges in probability to a certain deterministic kernel matrix, which is called Neural Tangent Kernel (NTK) [Jacot et al., 2018] and corresponds to infinitely wide neural networks. See Figure 5-1 for an illustration.

Explicit formulas for NTKs of fully-connected neural networks have been given in Jacot et al. [2018]. Recently, explicit formulas for NTKs of convolutional neural networks are given in Arora et al. [2019c]. The goal of this section is to give an explicit formula for NTKs that correspond to GNNs defined previously. Let $f(\theta, G) \in \mathbb{R}$ be the output of the corresponding GNN under parameters θ and input graph G, for two given graphs G and G', to calculate the corresponding GNTK value, we need to calculate the expected value of

$$\left\langle \frac{\partial f(\theta, G)}{\partial \theta}, \frac{\partial f(\theta, G')}{\partial \theta} \right\rangle$$

in the limit that $m \to \infty$ and θ are all Gaussian random variables, which can be viewed as a Gaussian process. For each layer in the GNN, we use $\boldsymbol{\Sigma}$ to denote the covariance matrix of outputs of that layer, and $\dot{\boldsymbol{\Sigma}}$ to denote the covariance matrix corresponds to the derivative of that layer. Due to the multi-layer structure of GNNs, these covariance matrices can be naturally calculated via dynamic programming.

5.2.2 Exact Computation of Infinitely Wide GNNs

Given two graphs $G = (V, E), G' = (V', E')$ with $|V| = n$, $|V'| = n'$ and a GNN with L BLOCK operations and R fully-connected layers with ReLU activation in each BLOCK operation. We give the GNTK formula of pairwise kernel value $\Theta(G, G') \in \mathbb{R}$ induced by this GNN.

We first define the covariance matrix between input features of two input graphs G, G', which we use $\boldsymbol{\Sigma}^{(0)}(G, G') \in \mathbb{R}^{n \times n'}$ to denote. For two nodes $u \in V$ and $u' \in V'$, $\left[\boldsymbol{\Sigma}^{(0)}(G, G')\right]_{uu'}$ is defined to be $\boldsymbol{h}_u^\top \boldsymbol{h}_{u'}$, where $\boldsymbol{h}_u$ and $\boldsymbol{h}_{u'}$ are the input features of $u \in V$ and $u' \in V'$.

BLOCK Operation. A BLOCK operation in GNTK calculates a covariance matrix $\boldsymbol{\Sigma}_{(R)}^{(\ell)}(G, G') \in \mathbb{R}^{n \times n'}$ using $\boldsymbol{\Sigma}_{(R)}^{(\ell-1)}(G, G') \in \mathbb{R}^{n \times n'}$, and calculates intermediate kernel values $\boldsymbol{\Theta}_{(r)}^{(\ell)}(G, G') \in \mathbb{R}^{n \times n'}$, which will be later used to compute the final output.

More specifically, we first perform a neighborhood aggregation operation

$$\begin{aligned}
\left[\boldsymbol{\Sigma}_{(0)}^{(\ell)}(G, G')\right]_{uu'} =& c_u c_{u'} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \sum_{v' \in \mathcal{N}(u') \cup \{u'\}} \left[\boldsymbol{\Sigma}_{(R)}^{(\ell-1)}(G, G')\right]_{vv'}, \\
\left[\boldsymbol{\Theta}_{(0)}^{(\ell)}(G, G')\right]_{uu'} =& c_u c_{u'} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \sum_{v' \in \mathcal{N}(u') \cup \{u'\}} \left[\boldsymbol{\Theta}_{(R)}^{(\ell-1)}(G, G')\right]_{vv'}.
\end{aligned}$$

Here we define $\boldsymbol{\Sigma}_{(R)}^{(0)}(G, G')$ and $\boldsymbol{\Theta}_{(R)}^{(0)}(G, G')$ as $\boldsymbol{\Sigma}^{(0)}(G, G')$, for notational convenience. Next we perform R transformations that correspond to the R fully-connected layers with ReLU activation. Here $\sigma(z) = \max\{0, z\}$ is the ReLU activation function. We denote $\dot{\sigma}(z) = \mathbb{1}[z \geq 0]$ to be the derivative of the ReLU activation function.

For each $r \in [R]$, we define

- For $u \in V, u' \in V'$,

$$\left[\boldsymbol{A}_{(r)}^{(\ell)}(G, G')\right]_{uu'} = \begin{pmatrix} \left[\boldsymbol{\Sigma}_{(r-1)}^{(\ell)}(G, G)\right]_{u,u} & \left[\boldsymbol{\Sigma}_{(r-1)}^{(\ell)}(G, G')\right]_{uu'} \\ \left[\boldsymbol{\Sigma}_{(r-1)}^{(\ell)}(G', G)\right]_{uu'} & \left[\boldsymbol{\Sigma}_{(r-1)}^{(\ell)}(G', G')\right]_{u'u'} \end{pmatrix} \in \mathbb{R}^{2 \times 2}.$$

- For $u \in V, u' \in V'$,

$$\left[\boldsymbol{\Sigma}_{(r)}^{(\ell)}(G, G')\right]_{uu'} = c_\sigma \mathbb{E}_{(a,b) \sim \mathcal{N}\left(\mathbf{0}, \left[\boldsymbol{A}_{(r)}^{(\ell)}(G,G')\right]_{uu'}\right)} \left[\sigma(a)\sigma(b)\right], \tag{5.5}$$

$$\left[\dot{\boldsymbol{\Sigma}}_{(r)}^{(\ell)}(G, G')\right]_{uu'} = c_\sigma \mathbb{E}_{(a,b) \sim \mathcal{N}\left(\mathbf{0}, \left[\boldsymbol{A}_{(r)}^{(\ell)}(G,G')\right]_{uu'}\right)} \left[\dot{\sigma}(a)\dot{\sigma}(b)\right]. \tag{5.6}$$

- For $u \in V, u' \in V'$,

$$\left[\boldsymbol{\Theta}_{(r)}^{(\ell)}(G, G')\right]_{uu'} = \left[\boldsymbol{\Theta}_{(r-1)}^{(\ell)}(G, G')\right]_{uu'} \left[\dot{\boldsymbol{\Sigma}}_{(r)}^{(\ell)}(G, G')\right]_{uu'} + \left[\boldsymbol{\Sigma}_{(r)}^{(\ell)}(G, G')\right]_{uu'}.$$

Note in the above we have shown how to calculate $\boldsymbol{\Theta}_{(R)}^{(\ell)}(G, G')$ for each $\ell \in \{0, 1, \ldots, L\}$. These intermediate outputs will be used to calculate the final output of the corresponding GNTK.

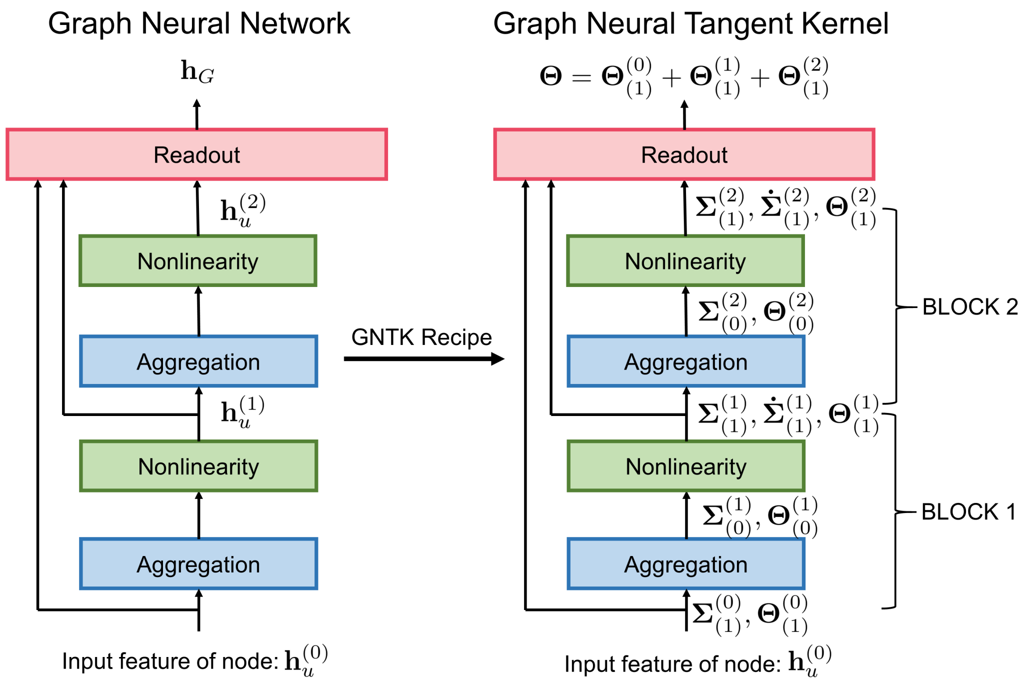

Graph Neural Network
$\mathbf{h}_G$
Readout
$\mathbf{h}_u^{(2)}$
Nonlinearity
Aggregation
$\mathbf{h}_u^{(1)}$
Nonlinearity
Aggregation
Input feature of node: $\mathbf{h}_u^{(0)}$
GNTK Recipe
Graph Neural Tangent Kernel
$\mathbf{\Theta} = \mathbf{\Theta}_{(1)}^{(0)} + \mathbf{\Theta}_{(1)}^{(1)} + \mathbf{\Theta}_{(1)}^{(2)}$
Readout
$\mathbf{\Sigma}_{(1)}^{(2)}, \dot{\mathbf{\Sigma}}_{(1)}^{(2)}, \mathbf{\Theta}_{(1)}^{(2)}$
Nonlinearity
$\mathbf{\Sigma}_{(0)}^{(2)}, \mathbf{\Theta}_{(0)}^{(2)}$
Aggregation
$\mathbf{\Sigma}_{(1)}^{(1)}, \dot{\mathbf{\Sigma}}_{(1)}^{(1)}, \mathbf{\Theta}_{(1)}^{(1)}$
Nonlinearity
$\mathbf{\Sigma}_{(0)}^{(1)}, \mathbf{\Theta}_{(0)}^{(1)}$
Aggregation
$\mathbf{\Sigma}_{(1)}^{(0)}, \mathbf{\Theta}_{(1)}^{(0)}$
Input feature of node: $\mathbf{h}_u^{(0)}$
BLOCK 2
BLOCK 1

Figure 5-2: **Illustration of our recipe that translates a GNN to a GNTK.** For a GNN with $L = 2$ BLOCK operations, $R = 1$ fully-connected layer in each BLOCK operation, and jumping knowledge, the corresponding GNTK is calculated as follow. For two graphs G and G', we first calculate $\left[\mathbf{\Theta}_{(1)}^{(0)}(G, G')\right]_{uu'} = \left[\mathbf{\Sigma}_{(1)}^{(0)}(G, G')\right]_{uu'} = \left[\mathbf{\Sigma}^{(0)}(G, G')\right]_{uu'} = \boldsymbol{h}_u^\top \boldsymbol{h}_{u'}$. We follow our GNTK formula to calculate $\mathbf{\Sigma}_{(0)}^{(\ell)}, \mathbf{\Theta}_{(0)}^{(\ell)}$ using $\mathbf{\Sigma}_{(R)}^{(\ell-1)}, \mathbf{\Theta}_{(R)}^{(\ell-1)}$ (Aggregation) and calculate $\mathbf{\Sigma}_{(r)}^{(\ell)}, \dot{\mathbf{\Sigma}}_{(r)}^{(\ell)}, \mathbf{\Theta}_{(r)}^{(\ell)}$ using $\mathbf{\Sigma}_{(r-1)}^{(\ell)}, \mathbf{\Theta}_{(r-1)}^{(\ell)}$ (Nonlinearity). The final output is $\Theta(G, G') = \sum_{u \in V, u' \in V'} \left[\sum_{\ell=0}^{L} \mathbf{\Theta}_{(R)}^{(\ell)}(G, G')\right]_{uu'}$.

READOUT Operation. Given these intermediate outputs, we can now calculate the final output of GNTK using the following formula.

$$\Theta(G, G') = \begin{cases} \sum_{u \in V, u' \in V'} \left[\mathbf{\Theta}_{(R)}^{(\ell)}(G, G')\right]_{uu'} & \text{without jumping knowledge} \\ \sum_{u \in V, u' \in V'} \left[\sum_{\ell=0}^{L} \mathbf{\Theta}_{(R)}^{(\ell)}(G, G')\right]_{uu'} & \text{with jumping knowledge} \end{cases}.$$

To better illustrate our general recipe, in Figure 5-2 we give a concrete example in which we translate a GNN with $L = 2$ BLOCK operations, $R = 1$ fully-connection layer in each BLOCK operation, and jumping knowledge, to its corresponding GNTK.

5.3 Generalization Bound

In this section, we analyze the generalization ability of GNTK which corresponds to over-parameterized GNNs trained by gradient descent. We consider the standard supervised learning setup. We are given n training data $\{(G_i, y_i)\}_{i=1}^n$ drawn i.i.d. from the underlying distribution $\mathcal{D}$, where G_i is the i-th input graph and y_i is its label. Consider a GNN with a single BLOCK operation, followed by the READOUT operation (without jumping knowledge). Here we set $c_u = \left(\left\|\sum_{v \in \mathcal{N}(u) \cup \{u\}} \boldsymbol{h}_v\right\|_2\right)^{-1}$. We use $\mathbf{\Theta} \in \mathbb{R}^{n \times n}$ to denote the kernel matrix, where $\mathbf{\Theta}_{ij} = \Theta(G_i, G_j)$. Here $\Theta(G, G')$ is the kernel function that corresponds to the simple GNN. Throughout the discussion, we assume that the kernel matrix $\mathbf{\Theta} \in \mathbb{R}^{n \times n}$ is invertible.

For a testing point G_{te}, the prediction of kernel regression using GNTK on this testing point is

$$f_{ker}(G_{te}) = \left[\Theta(G_{te}, G_1), \Theta(G_{te}, G_1), \ldots, \Theta(G_{te}, G_n)\right]^\top \mathbf{\Theta}^{-1} \boldsymbol{y}.$$

The following result is a standard result for kernel regression proved using Rademacher complexity. For a proof, see Bartlett and Mendelson [2002].

Theorem 5.1 (Bartlett and Mendelson [2002]). *Given n training data $\{(G_i, y_i)\}_{i=1}^n$ drawn i.i.d. from the underlying distribution $\mathcal{D}$. Consider any loss function $\ell : \mathbb{R} \times \mathbb{R} \to [0, 1]$ that is 1-Lipschitz in the first argument such that $\ell(y, y) = 0$. With probability at least $1 - \delta$, the* *population loss of the GNTK predictor can be upper bounded by*

$$L_{\mathcal{D}}\left(f_{ker}\right)=\mathbb{E}_{(G,y)\sim\mathcal{D}}\left[\ell(f_{ker}(G),y)\right]=O\left(\frac{\sqrt{\boldsymbol{y}^\top\boldsymbol{\Theta}^{-1}\boldsymbol{y}\cdot\operatorname{tr}\left(\boldsymbol{\Theta}\right)}}{n}+\sqrt{\frac{\log(1/\delta)}{n}}\right).$$

Note that this theorem presents a data-dependent generalization bound which is related to the kernel matrix $\boldsymbol{\Theta}\in\mathbb{R}^{n\times n}$ and the labels $\{y_i\}_{i=1}^n$. Using this theorem, if we can bound $\boldsymbol{y}^\top\boldsymbol{\Theta}^{-1}\boldsymbol{y}$ and $\operatorname{tr}\left(\boldsymbol{\Theta}\right)$, then we can obtain a concrete sample complexity bound. We instantiate this idea to study the class of graph labeling functions that can be efficiently learned by GNTKs.

The following two theorems guarantee that if labels are generated as described in (5.7), then the GNTK that corresponds to the simple GNN described above can learn this function with polynomial number of samples. We first give an upper bound on $\boldsymbol{y}^\top\boldsymbol{\Theta}^{-1}\boldsymbol{y}$.

Theorem 5.2. *For each $i\in[n]$, if the labels $\{y_i\}_{i=1}^n$ satisfy*

$$y_i=\alpha_1\sum_{u\in V}\left(\overline{\boldsymbol{h}}_u^\top\boldsymbol{\beta}_1\right)+\sum_{l=1}^{\infty}\alpha_{2l}\sum_{u\in V}\left(\overline{\boldsymbol{h}}_u^\top\boldsymbol{\beta}_{2l}\right)^{2l}, \tag{5.7}$$

where $\overline{\boldsymbol{h}}_u=c_u\sum_{v\in\mathcal{N}(u)\cup\{u\}}\boldsymbol{h}_v$, $\alpha_1,\alpha_2,\alpha_4,\ldots\in\mathbb{R}$, $\boldsymbol{\beta}_1,\boldsymbol{\beta}_2,\boldsymbol{\beta}_4,\ldots\in\mathbb{R}^d$, and $G_i=(V,E)$, then we have

$$\sqrt{\boldsymbol{y}^\top\boldsymbol{\Theta}^{-1}\boldsymbol{y}}\leq 2|\alpha_1|\cdot\|\boldsymbol{\beta}_1\|_2+\sum_{l=1}^{\infty}\sqrt{2\pi}(2l-1)|\alpha_{2l}|\cdot\|\boldsymbol{\beta}_{2l}\|_2^{2l}.$$

The following theorem gives an upper bound on $\operatorname{tr}\left(\boldsymbol{\Theta}\right)$.

Theorem 5.3. *If for all graphs $G_i=(V_i,E_i)$ in the training set, $|V_i|$ is upper bounded by $\overline{V}$, then $\operatorname{tr}(\boldsymbol{\Theta})\leq O(n\overline{V}^2)$. Here n is the number of training samples.*

Combining Theorem 5.2 and Theorem 5.3 with Theorem 5.1, we know if

$$2|\alpha_1|\cdot\|\boldsymbol{\beta}_1\|_2+\sum_{l=1}^{\infty}\sqrt{2\pi}(2l-1)|\alpha_{2l}|\cdot\|\boldsymbol{\beta}_{2l}\|_2^{2l}$$

is bounded, and $|V_i|$ is bounded for all graphs $G_i=(V_i,E_i)$ in the training set, then the GNTK that corresponds to the simple GNN described above can learn functions of forms in (5.7), with polynomial number of samples. To our knowledge, this is the first sample complexity analysis in the GK and GNN literature.

5.4 Experiments

In this section, we demonstrate that such infinitely-wide GNNs in fact perform well in practice, and can even outperform the finite GNNs. In particular, we validate the effectiveness of GNTKs with experiments on graph classification tasks. Following common practices of evaluating performance of graph classification models Yanardag and Vishwanathan [2015], we perform 10-fold cross validation and report the mean and standard deviation of validation accuracies. More details about the experiment setup can be found in the Appendix.

Datasets. The benchmark datasets include four bioinformatics datasets MUTAG, PTC, NCI1, PROTEINS and three social network datasets COLLAB, IMDB-BINARY, IMDB-MULTI. For each graph, we transform the categorical input features to one-hot encoding representations. For datasets where the graphs have no node features, i.e. only graph structure matters, we use degrees as input node features.

Results We compare GNTK with various state-of-the-art graph classification algorithms: (1) the WL subtree kernel [Shervashidze et al., 2011]; (2) state-of-the-art deep learning architectures, including Graph Convolutional Network (GCN) [Kipf and Welling, 2017], GraphSAGE [Hamilton et al., 2017], Graph Isomorphism Network(GIN) [Xu et al., 2019], PATCHY-SAN [Niepert et al., 2016] and Deep Graph CNN (DGCNN) [Zhang et al., 2018a]; (3) Graph kernels based on random walks, i.e., Anonymous Walk Embeddings [Ivanov and Burnaev, 2018] and RetGK [Zhang et al., 2018b]. For deep learning methods and random walk graph kernels, we report the accuracies reported in the original papers. The experiment setup is deferred to Appendix.

The graph classification results are shown in Table 5.1. The best results are highlighted as bold. Infinitely wide GNNs, i.e., GNTKs, are powerful and achieve state-of-the-art classification accuracy on most datasets. In four of them, we find GNTKs outperform all

	Method	COLLAB	IMDB-B	IMDB-M	PTC	NCI1	MUTAG	PROTEINS
GNN	GCN	79.0 ± 1.8	74.0 ± 3.4	51.9 ± 3.8	64.2 ± 4.3	80.2 ± 2.0	85.6 ± 5.8	76.0 ± 3.2
	GraphSAGE	–	72.3 ± 5.3	50.9 ± 2.2	63.9 ± 7.7	77.7 ± 1.5	85.1 ± 7.6	75.9 ± 3.2
	PatchySAN	72.6 ± 2.2	71.0 ± 2.2	45.2 ± 2.8	60.0 ± 4.8	78.6 ± 1.9	$\mathbf{92.6} \pm \mathbf{4.2}$	75.9 ± 2.8
	DGCNN	73.7	70.0	47.8	58.6	74.4	85.8	75.5
	GIN	80.2 ± 1.9	75.1 ± 5.1	52.3 ± 2.8	64.6 ± 7.0	82.7 ± 1.7	89.4 ± 5.6	$\mathbf{76.2} \pm \mathbf{2.8}$
GK	WL subtree	78.9 ± 1.9	73.8 ± 3.9	50.9 ± 3.8	59.9 ± 4.3	$\mathbf{86.0} \pm \mathbf{1.8}$	90.4 ± 5.7	75.0 ± 3.1
	AWL	73.9 ± 1.9	74.5 ± 5.9	51.5 ± 3.6	–	–	87.9 ± 9.8	–
	RetGK	81.0 ± 0.3	71.9 ± 1.0	47.7 ± 0.3	62.5 ± 1.6	84.5 ± 0.2	90.3 ± 1.1	75.8 ± 0.6
	GNTK	$\mathbf{83.6} \pm \mathbf{1.0}$	$\mathbf{76.9} \pm \mathbf{3.6}$	$\mathbf{52.8} \pm \mathbf{4.6}$	$\mathbf{67.9} \pm \mathbf{6.9}$	84.2 ± 1.5	90.0 ± 8.5	75.6 ± 4.2

Table 5.1: **Classification results (in %) for graph classification datasets.** GNN: graph neural network based methods. GK: graph kernel based methods. GNTK: fusion of GNN and GK.

baseline methods. In particular, GNTKs achieve 83.6% accuracy on COLLAB dataset and 67.9% accuracy on PTC dataset, compared to the best of baselines, 81.0% and 64.6% respectively. Notably, GNTKs give the best performance on all social network datasets. Moreover, In our experiments, we also observe that with the same architecture, GNTK is more computational efficient that its GNN counterpart. On IMDB-B dataset, running GIN with the default setup (official implementation of Xu et al. [2019]) takes 19 minutes on a TITAN X GPU and running GNTK only takes 2 minutes.

Conclusion. In this chapter, we kept our focus simple, and studied the generalization of over-parameterized GNNs for general and simple functions on graphs. In the next chapter, we will go back to our core question of modeling intelligence for reasoning. Our analysis will take into account the task structure of reasoning problems.

Chapter 6

What Can Neural Networks Reason About

This chapter asks what reasoning a neural network can learn. To this end, we analyze the generalization and sample complexity of neural networks (e.g., GNNs) in learning reasoning problems, by considering the interplay of the network structure and the task structure.

6.1 Introduction

Recently, there have been many advances in building neural networks that can learn to reason. Reasoning spans a variety of tasks, for instance, visual and text-based question answering [Johnson et al., 2017a, Weston et al., 2015, Hu et al., 2017, Fleuret et al., 2011, Antol et al., 2015], intuitive physics, i.e., predicting the time evolution of physical objects [Battaglia et al., 2016, Watters et al., 2017, Fragkiadaki et al., 2016, Chang et al., 2017], mathematical reasoning [Saxton et al., 2019, Chang et al., 2019] and visual IQ tests [Santoro et al., 2018, Zhang et al., 2019a].

Curiously, neural networks that perform well in reasoning tasks usually possess specific *structures* [Santoro et al., 2017]. Many successful models follow the Graph Neural Network (GNN) framework [Battaglia et al., 2018, 2016, Palm et al., 2018, Mrowca et al., 2018, Sanchez-Gonzalez et al., 2018, Janner et al., 2019]. These networks explicitly model pairwise relations and recursively update each object's representation by aggregating its relations with other objects. Other computational structures, e.g., neural symbolic programs [Yi et al., 2018a, Mao et al., 2019, Johnson et al., 2017b], are effective on specific tasks.

However, there is limited understanding of the relation between the generalization ability and network structure for reasoning. Answering the following fundamental question is crucial for understanding the empirical success and limitations of existing models, and for designing better models for new reasoning tasks.

What tasks can a neural network sample efficiently learn to reason about?

This chapter answers this fundamental question by developing a theoretical framework to characterize what tasks a neural network can reason about. We build on a simple observation that reasoning processes resemble algorithms. Hence, we study how well a reasoning algorithm *aligns* with the computation graph of the network. Intuitively, if they align well, the network only needs to learn *simple* algorithm steps to simulate the reasoning process, which leads to better sample efficiency. We formalize this intuition with a numeric measure of algorithmic alignment, and show initial support for our hypothesis that algorithmic alignment facilitates learning: Under simplifying assumptions, we show a sample complexity bound that decreases with better alignment.

Our framework explains the empirical success of popular reasoning models and suggests their limitations. As concrete examples, we study four categories of increasingly complex reasoning tasks: summary statistics, relational argmax (asking about properties of the result of comparing multiple relations), dynamic programming, and NP-hard problems (Figure 6-1). Using alignment, we characterize which architectures are expected to learn each task well: Networks inducing permutation invariance, such as Deep Sets [Zaheer et al., 2017], can learn summary statistics, and one-iteration GNNs can learn relational argmax. Many other more complex tasks, such as intuitive physics, visual question answering, and shortest paths – despite seeming different – can all be solved via a powerful algorithmic paradigm: dynamic programming (DP) [Bellman, 1966]. Multi-iteration GNNs algorithmically align with DP and hence are expected to sample-efficiently learn these tasks. Indeed, they do. Our results offer an explanation for the popularity of GNNs in the relational reasoning literature, and also suggest limitations for tasks with even more complex structure. As an

Summary statistics What is the maximum value difference among treasures? | *Relational argmax* What are the colors of the furthest pair of objects? | *Dynamic programming* What is the cost to defeat monster X by following the optimal path? | *NP-hard problem* Subset sum: Is there a subset that sums to 0?

Figure 6-1: **Overview of reasoning tasks with increasingly complex structure.** Each *task category* shows an example task on which we perform experiments. Algorithmic alignment suggests that (a) Deep Sets and GNNs, but not MLP, can *sample efficiently* learn summary statistics, (b) GNNs, but not Deep Sets, can learn relational argmax, (c) GNNs can learn dynamic programming, an algorithmic paradigm that we show to unify many reasoning tasks, (d) GNNs cannot learn subset sum (NP-hard), but NES, a network we design based on exhaustive search, can generalize. Our theory agrees with empirical results (Figure 6-3).

example of such a task, we consider subset sum, an NP-hard problem where GNNs indeed fail. Overall, empirical results (Figure 6-3) agree with our theoretical analysis based on algorithmic alignment (Figure 6-1). These findings also suggest how to take into account task structure when designing new architectures.

The perspective that structure in networks helps is not new. For example, in a well-known position paper, Battaglia et al. [2018] argue that GNNs are suitable for relational reasoning because they have relational inductive biases, but without formalizations. Here, we take such ideas one step further, by introducing a *formal* definition (algorithmic alignment) for quantifying the relation between network and task structure, and by formally deriving implications for learning. These theoretical ideas are the basis for characterizing what reasoning tasks a network can learn well. Our algorithmic structural condition also differs from structural assumptions common in learning theory [Vapnik, 2013, Bartlett and Mendelson, 2002, Bartlett et al., 2017, Neyshabur et al., 2015, Golowich et al., 2018] and specifically aligns with reasoning.

In summary, we introduce algorithmic alignment to analyze learning for reasoning. Our initial theoretical results suggest that algorithmic alignment is desirable for generalization. On four categories of reasoning tasks with increasingly complex structure, we apply our framework to analyze which tasks some popular networks can learn well. GNNs algorithmically align with dynamic programming, which solves a broad range of reasoning tasks.

Finally, our framework implies guidelines for designing networks for new reasoning tasks. Experimental results confirm our theory.

6.1.1 Preliminaries

We begin by introducing notations and a few example architectures that we analyze for reasoning tasks. It is often more sample efficient to separate reasoning from perception or representation [Mao et al., 2019, Yi et al., 2018a]. Hence, following previous works, we assume access to the representation of objects in a world or universe S, i.e., a configuration/set of objects to reason about. Each object $s \in S$ is represented by a representation vector X_s. The representation could be descriptions prepared by humans or features learned by representation learning steps such as image segmentation [Santoro et al., 2017]. Information about the specific question can also be included in the object representations. Given a set of universes $\{S_1, ..., S_M\}$ and answer labels $\{y_1, ..., y_M\} \subseteq \mathcal{Y}$, we aim to learn a function g (a reasoning process) that can answer questions about unseen universes, $y = g\,(S)$. Mathematically, a universe S can be a set, a sequence, or a graph, depending on the information available between the objects.

Multi-layer perceptron. For a single-object universe, applying an MLP, a.k.a. feedforward neural network, on the object representation usually works well. But when there are multiple objects, simply applying an MLP to the *concatenated object representations* often does not generalize [Santoro et al., 2017].

Deep Sets. Suppose the collection of objects is a set, i.e., we do not have other order or dependence information. As the input to the reasoning function is an unordered set, the function should be permutation-invariant, i.e., the output is the same for all input orderings. To induce permutation invariance in a neural network, Zaheer et al. [2017] propose *Deep Sets*, of the form

$$y = \text{MLP}_2\Big(\sum_{s \in S} \text{MLP}_1\left(X_s\right)\Big). \tag{6.1}$$

Graph Neural Networks. If we have a graph of objects, constructed based on prior knowledge, we can apply GNNs to this graph. If we are given a set of objects and there is no present graph, GNNs can be adopted for reason about the set by considering objects as nodes, and assume all objects pairs are connected, *i.e.*, a complete graph:

$$h_s^{(k)} = \sum\nolimits_{t \in S} \text{MLP}_1^{(k)} \left(h_s^{(k-1)}, h_t^{(k-1)} \right), \quad h_S = \text{MLP}_2 \Big(\sum\nolimits_{s \in S} h_s^{(K)} \Big), \tag{6.2}$$

where h_S is the answer/output and K is the number of GNN layers. Each object's representation is initialized as $h_s^{(0)} = X_s$. Although other aggregation functions are proposed, we use sum in our experiments (we shall see in Chapter 8 that the aggregation does not matter much for interpolation, but matters for extrapolation). Similar to Deep Sets, GNNs are also permutation invariant. While Deep Sets focus on individual objects, GNNs can also focus on pairwise relations. The GNN framework includes many reasoning models. Relation Networks [Santoro et al., 2017] and Interaction Networks [Battaglia et al., 2016] resemble one-layer GNNs. Recurrent Relational Networks [Palm et al., 2018] apply LSTMs [Hochreiter and Schmidhuber, 1997] after aggregation.

Neural symbolic programs. This class of neural models usually target a specific task, such as visual question answering, and pre-encode a library of symbolic operations to be executed [Yi et al., 2018a, Mao et al., 2019, Johnson et al., 2017b]. The neural networks learn which symbolic operations to execute. With an appropriate library and training, neural symbolic programs often show impressive generalization and even extrapolation. However, they are also limited in the scope of tasks as the library needs to be carefully designed for the specific task.

6.2 Algorithmic Alignment: Inductive Biases and Sample Complexity

Next, we study how the network structure and task may interact, and implications for generalization. Empirically, different network structures have different degrees of success

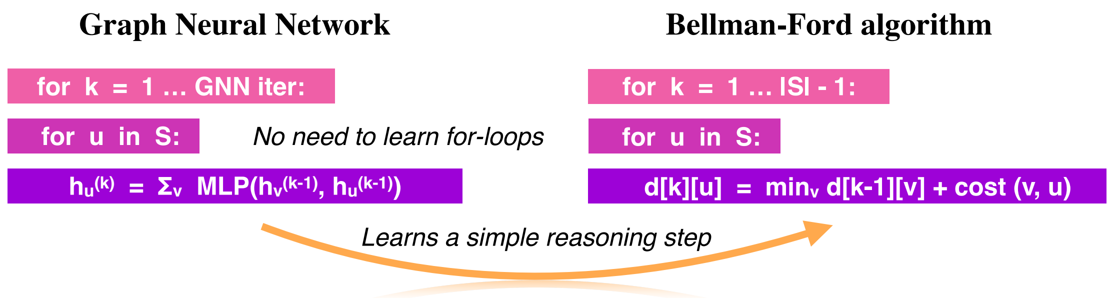

Graph Neural Network
for k = 1 ... GNN iter:
for u in S:
$h_u^{(k)} = \Sigma_v$ MLP($h_v^{(k-1)}$, $h_u^{(k-1)}$)
No need to learn for-loops
Bellman-Ford algorithm
for k = 1 ... ISI - 1:
for u in S:
d[k][u] = $\min_v$ d[k-1][v] + cost (v, u)
Learns a simple reasoning step

Figure 6-2: **Our framework suggests that better algorithmic alignment improves generalization.** As an example, our framework explains why GNN generalizes when learning to answer shortest paths. A correct reasoning process for the shortest paths task is the Bellman-Ford algorithm. The computation structure of a GNN (left) *aligns* well with Bellman-Ford (right): the GNN can simulate Bellman-Ford by merely learning a *simple reasoning step*, i.e., the relaxation step in the last line (a sum, and a min over neighboring nodes v) via its aggregation operation. In contrast, a giant MLP or Deep Set must learn the structure of the *entire for-loop*. Thus, the GNN is expected to generalize better when learning shortest paths, as is confirmed in experiments (Section 6.3.3).

in learning reasoning tasks, e.g., GNNs can learn relations well, but Deep Sets often fail (Figure 6-3). However, all these networks are universal approximators (Propositions 6.1 and 6.2). Thus, their differences in test accuracy must come from generalization.

We observe that the answer to many reasoning tasks may be computed via a reasoning algorithm; we shall further illustrate the algorithms for some reasoning tasks. Many neural networks can *represent* algorithms [Pérez et al., 2019]. For example, Deep Sets can universally represent permutation-invariant set functions [Zaheer et al., 2017, Wagstaff et al., 2019]. This also holds for GNNs and MLPs, as we show in Propositions 6.1 and 6.2:

Proposition 6.1. *Let $f : \mathbb{R}^{d\times N} \to \mathbb{R}$ be any continuous function over sets S of bounded cardinality $|S| \leq N$. If f is permutation-invariant to the elements in S, and the elements are in a compact set in $\mathbb{R}^d$, then f can be approximated arbitrarily closely by a GNN (of any depth).*

Proposition 6.2. *For any GNN $\mathcal{N}$, there is an MLP that can represent all functions $\mathcal{N}$ can represent.*

But, empirically, not all network structures work well when *learning* these algorithms, i.e., they generalize differently. Intuitively, a network may generalize better if it can represent a function "more easily". We formalize this idea by *algorithmic alignment*, formally defined in Definition 6.2. Indeed, not only the reasoning process has an algorithmic structure: the neural network's architecture induces a computational structure on the function it computes. This corresponds to an algorithm that prescribes how the network combines computations from modules. Figure 6-2 illustrates this idea for a GNN, where the modules are its MLPs applied to pairs of objects. In the shortest paths problem, the GNN matches the structure of the Bellman-Ford algorithm: to simulate the Bellman-Ford with a GNN, the GNN's MLP modules only need to learn a *simple* update equation (Figure 6-2). In contrast, if we want to represent the Bellman-Ford algorithm with a single MLP, it needs to simulate an *entire for-loop*, which is much more complex than one update step. Therefore, we expect the GNN to have better sample complexity than MLP when learning to solve shortest path problems.

This perspective suggests that a neural network which better aligns with a correct reasoning process (algorithmic solution) can more easily learn a reasoning task than a neural network that does not align well. If we look more broadly at reasoning, there may also exist solutions which only solve a task approximately, or whose structure is obtuse. In this work, we focus on reasoning tasks whose underlying reasoning process is exact and has clear algorithmic structure. We leave the study of approximation algorithms and unknown structures for future work.

6.2.1 Formalization of Algorithmic Alignment

We formalize the above intuition in a PAC learning framework [Valiant, 1984]. PAC learnability formalizes *simplicity* as sample complexity, i.e., the number of samples needed to ensure low test error with high probability. It refers to a learning algorithm $\mathcal{A}$ that, given training samples $\{x_i, y_i\}_{i=1}^M$, outputs a function $f = \mathcal{A}(\{x_i, y_i\}_{i=1}^M)$. The learning algorithm here is the neural network and its training method, e.g., gradient descent. A function is simple if it has low sample complexity.

Definition 6.1. (PAC learning and sample complexity). Fix an error parameter $\epsilon > 0$ and failure probability $\delta \in (0, 1)$. Suppose $\{x_i, y_i\}_{i=1}^M$ are i.i.d. samples from some distribution $\mathcal{D}$, and the data satisfies $y_i = g(x_i)$ for some underlying function g. Let $f = \mathcal{A}(\{x_i, y_i\}_{i=1}^M)$

be the function generated by a learning algorithm $\mathcal{A}$. Then g is (M, ϵ, δ)-*learnable* with $\mathcal{A}$ if

$$\mathbb{P}_{x \sim \mathcal{D}} \left[\| f(x) - g(x) \| \leq \epsilon \right] \geq 1 - \delta. \tag{6.3}$$

The *sample complexity* $\mathcal{C}_{\mathcal{A}}(g, \epsilon, \delta)$ is the minimum M so that g is (M, ϵ, δ)-learnable with $\mathcal{A}$.

With the PAC learning framework, we define a numeric measure of algorithmic alignment (Definition 6.2), and under simplifying assumptions, we show that the sample complexity decreases with better algorithmic alignment (Theorem 6.4).

Formally, a neural network aligns with an algorithm if it can simulate the algorithm via a limited number of modules, and each module is simple, i.e., has low sample complexity.

Definition 6.2. (Algorithmic alignment). Let g be a reasoning function and $\mathcal{N}$ a neural network with n modules $\mathcal{N}_i$. The module functions $f_1, ..., f_n$ generate g for $\mathcal{N}$ if, by replacing $\mathcal{N}_i$ with f_i, the network $\mathcal{N}$ simulates g. Then $\mathcal{N}$ (M, ϵ, δ)-algorithmically aligns with g if (1) $f_1, ..., f_n$ generate g and (2) there are learning algorithms $\mathcal{A}_i$ for the $\mathcal{N}_i$'s such that $n \cdot \max_i C_{\mathcal{A}_i}(f_i, \epsilon, \delta) \leq M$.

Good algorithmic alignment, i.e., small M, implies that all algorithm steps f_i to simulate the algorithm g are *easy to learn*. Therefore, the algorithm steps should not simulate complex programming constructs such as for-loops, whose sample complexity is large (Theorem 6.3).

Next, we show how to compute the algorithmic alignment value M. Algorithmic alignment resembles Kolmogorov complexity [Kolmogorov, 1998] for neural networks. Thus, it is generally non-trivial to obtain the optimal alignment between a neural network and an algorithm. However, one important difference to Kolmogorov complexity is that any algorithmic alignment that yields decent sample complexity is good enough (unless we want the tightest bound). We will see several examples where finding a good alignment is not hard. Then, we can compute the value of an alignment by summing the sample complexity of the algorithm steps with respect to the modules, e.g. MLPs. For ilustration, we show an example of how one may compute sample complexity of MLP modules.

A line of works show one can analyze the optimization and generalization behavior of overparameterized neural networks via neural tangent kernel (NTK) [Allen-Zhu et al., 2019a, Arora et al., 2019b,c, 2020, Du et al., 2019c,a, Jacot et al., 2018, Li and Liang, 2018]. In the previous chapter, we have shown that infinitely-wide GNNs trained with gradient descent can provably learn certain smooth functions [Yi et al., 2018a, Mao et al., 2019, Johnson et al., 2017b]. This chapter further takes into account the task structure.

Here, Theorem 6.3, proved in the Appendix, summarizes and extends Theorem 6.1 of Arora et al. [2019b] for over-parameterized MLP modules to vector-valued functions. Our framework can be used with other sample complexity bounds for other types of modules, too.

Theorem 6.3. ***(Sample complexity for overparameterized MLP modules).*** *Let $\mathcal{A}$ be an overparameterized and randomly initialized two-layer MLP trained with gradient descent for a sufficient number of iterations. Suppose $g : \mathbb{R}^d \to \mathbb{R}^m$ with components $g(x)^{(i)} = \sum_j \alpha_j^{(i)} \left(\beta_j^{(i)\top} x\right)^{p_j^{(i)}}$, where $\beta_j^{(i)} \in \mathbb{R}^d$, $\alpha \in \mathbb{R}$, and $p_j^{(i)} = 1$ or $p_j^{(i)} = 2l \, (l \in \mathbb{N}_+)$. The sample complexity $\mathcal{C}_{\mathcal{A}}(g, \epsilon, \delta)$ is*

$$\mathcal{C}_{\mathcal{A}}(g, \epsilon, \delta) = O\Big(\frac{\max_i \sum_{j=1}^K p_j^{(i)} |\alpha_j^{(i)}| \cdot \|\beta_j^{(i)}\|_2^{p_j^{(i)}} + \log{(m/\delta)}}{(\epsilon/m)^2}\Big). \tag{6.4}$$

Theorem 6.3 suggests that functions that are "simple" when expressed as a polynomial, e.g., via a Taylor expansion, are sample efficiently learnable by an MLP module. Thus, algorithm steps that perform computation over many objects may require many samples for an MLP module to learn, since the number K of polynomials or $\|\beta_j^{(i)}\|$ can increase in Eqn. (6.4). "For loop" is one example of such complex algorithm steps.

6.2.2 Better Algorithmic Alignment Implies Better Generalization

We show an initial result demonstrating that algorithmic alignment is desirable for generalization. Theorem 6.4 states that, in a simplifying setting where we sequentially train modules of a network with auxiliary labels, the sample complexity bound increases with algorithmic alignment value M.

While we do not have auxiliary labels in practice, we observe the same pattern for end-to-end learning in experiments. We leave sample complexity analysis for end-to-end-learning to future work. We prove Theorem 6.4 in Appendix D.1.4.

Theorem 6.4. ***(Algorithmic alignment improves sample complexity).*** *Fix ϵ and δ. Suppose $\{S_i, y_i\}_{i=1}^M \sim \mathcal{D}$, where $|S_i| < N$, and $y_i = g(S_i)$ for some g. Suppose $\mathcal{N}_1, ..., \mathcal{N}_n$ are network $\mathcal{N}$'s MLP modules in sequential order. Suppose $\mathcal{N}$ and g (M, ϵ, δ)-algorithmically align via functions $f_1, ..., f_n$. Under the following assumptions, g is $(M, O(\epsilon), O(\delta))$-learnable by $\mathcal{N}$.*

a) Algorithm stability. *Let $\mathcal{A}$ be the learning algorithm for the $\mathcal{N}_i$'s. Suppose $f = \mathcal{A}(\{x_i, y_i\}_{i=1}^M)$, and $\hat{f} = \mathcal{A}(\{\hat{x}_i, y_i\}_{i=1}^M)$. For any x, $\|f(x) - \hat{f}(x)\| \leq L_0 \cdot \max_i \|x_i - \hat{x}_i\|$, for some L_0.*

b) Sequential learning. *We train $\mathcal{N}_i$'s sequentially: $\mathcal{N}_1$ has input samples $\{\hat{x}_i^{(1)}, f_1(\hat{x}_i^{(1)})\}_{i=1}^N$, with $\hat{x}_i^{(1)}$ obtained from S_i. For $j > 1$, the input $\hat{x}_i^{(j)}$ for $\mathcal{N}_j$ are the outputs from the previous modules, but labels are generated by the correct functions $f_{j-1}, ..., f_1$ on $\hat{x}_i^{(1)}$.*

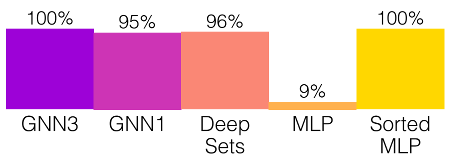
c) Lipschitzness. *The learned functions $\hat{f}_j$ satisfy $\|\hat{f}_j(x) - \hat{f}_j(\hat{x})\| \leq L_1 \|x - \hat{x}\|$, for some L_1.*

In our analysis, the Lipschitz constants and the universe size are constants going into $O(\epsilon)$ and $O(\delta)$. As an illustrative example, we use Theorem 6.4 and 6.3 to show that GNN has a polynomial improvement in sample complexity over MLP when learning simple relations. Indeed, GNN aligns better with summary statistics of pairwise relations than MLP does (Section 6.3.1).

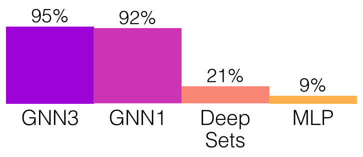
Corollary 6.5. *Suppose universe S has ℓ objects $X_1, ..., X_\ell$, and $g(S) = \sum_{i,j}(X_i - X_j)^2$. In the setting of Theorem 6.4, the sample complexity bound for MLP is $O(\ell^2)$ times larger than for GNN.*

6.3 Predicting What Neural Networks Can Reason About

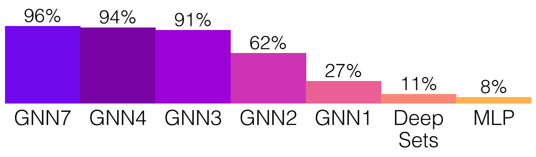
Next, we apply our framework to analyze several neural networks for reasoning introduced previously: MLP, Deep Sets, and GNNs. Using algorithmic alignment, we predict whether each model can generalize on four categories of increasingly complex reasoning tasks: summary statistics, relational argmax, dynamic programming, and an NP-hard problem



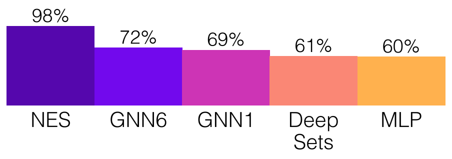
(a) Maximum value difference.



(b) Furthest pair.



(c) Monster trainer.



(d) Subset sum. Random yields 50%.

Figure 6-3: **Test accuracies on reasoning tasks with increasingly complex structure.** Figure 6-1 shows an overview of the tasks. GNNk is GNN with k iterations. (a) Summary statistics. All models except MLP generalize. (b) Relational argmax. Deep Sets fail. (c) Dynamic programming. Only GNNs with sufficient iterations generalize. (d) An NP-hard problem. Even GNNs fail, but NES generalizes.

(Figure 6-3). Our theoretical analysis is confirmed with experiments (Dataset and training details are in Appendix D.2). To empirically compare sample complexity of different models, we make sure all models perfectly fit training sets through extensive hyperparameter tuning. Therefore, the test accuracy reflects how well a model generalizes.

The examples in this section, together with our framework, suggest an explanation why GNNs are widely successful across reasoning tasks: Popular reasoning tasks such as visual question answering and intuitive physics can be solved by DP. GNNs align well with DP, and hence are expected to learn sample efficiently.

6.3.1 Summary Statistics

As discussed previously, we assume each object X has a state representation $X = [h_1, h_2, ..., h_k]$, where each $h_i \in \mathbb{R}^{d_i}$ is a feature vector. An MLP can learn simple polynomial functions of the state representation (Theorem 6.3). In this section, we show how Deep Sets use MLP as building blocks to learn summary statistics.

Questions about summary statistics are common in reasoning tasks. One example from CLEVR [Johnson et al., 2017a] is “How many objects are either small cylinders or red things?” Deep Sets (Eqn. 6.1) align well with algorithms that compute summary statistics over individual objects. Suppose we want to compute the sum of a feature over all objects. To simulate the reasoning algorithm, we can use the first MLP in Deep Sets to extract the desired feature and aggregate them using the pooling layer. Under this alignment, each MLP only needs to learn simple steps, which leads to good sample complexity. Similarly, Deep Sets can learn to compute *max or min* of a feature by using smooth approximations like the softmax $\max_{s \in S} X_s \approx \log(\sum_{s \in X_s} \exp(X_s))$. In contrast, if we train an MLP to perform sum or max, the MLP must learn a complex for-loop and therefore needs more samples. Therefore, our framework predicts that Deep Sets have better sample complexity than MLP when learning summary statistics.

Maximum value difference. We confirm our predictions by training models to compute the *maximum value difference* task. Each object in this task is a treasure $X = [h_1, h_2, h_3]$ with location h_1, value h_2, and color h_3. We train models to predict the difference in value between the most and the least valuable treasure, $y(S) = \max_{s \in S} h_2(X_s) - \min_{s \in S} h_2(X_s)$.

The test accuracy follows our prediction (Figure 6-3a). MLP does not generalize and only has 9% test accuracy, while Deep Sets has 96%. Interestingly, if we sort the treasures by value (Sorted MLP in Figure 6-3a), MLP achieves perfect test accuracy. This observation can be explained with our theory—when the treasures are sorted, the reasoning algorithm is reduced to a simple subtraction: $y(S) = h_2(X_{|S|}) - h_2(X_1)$, which has a low sample complexity for even MLPs (Theorem 6.3). GNNs also have high test accuracies. This is because summary statistics are a special case of relational argmax, which GNNs can learn as shown next.

6.3.2 Relational Argmax

Next, we study *relational argmax*: tasks where we need to compare pairwise relations and answer a question about that result. For example, a question from Sort-of-CLEVR [Santoro et al., 2017] asks "What is the shape of the object that is farthest from the gray object?", which requires comparing the distance between object pairs.

One-iteration GNN aligns well with relational argmax, as it sums over all pairs of objects, and thus can compare, e.g. via softmax, pairwise information without learning the "for loops". In contrast, Deep Sets require many samples to learn this, because most pairwise relations cannot be encoded as a sum of individual objects:

Claim 6.6. *Suppose $g(x, y) = 0$ if and only if $x = y$. There is no f such that $g(x, y) = f(x) + f(y)$.*

Therefore, if we train a Deep Set to compare pairwise relations, one of the MLP modules has to learn a complex "for loop", which leads to poor sample complexity. Our experiment confirms that GNNs generalize better than Deep Sets when learning relational argmax.

Furthest pair. As an example of relational argmax, we train models to identify the furthest pair among a set of objects. We use the same object settings as the maximum value difference task. We train models to find the colors of the two treasures with the largest distance. The answer is a pair of colors, encoded as an integer category:

$$y(S) = (h_3(X_{s_1}), h_3(X_{s_2})) \quad \text{s.t.} \ \{X_{s_1}, X_{s_2}\} = \arg\max_{s_1,s_2 \in S} \|h_1(X_{s_1}) - h_1(X_{s_2})\|_{\ell_1}$$

Distance as a pairwise function satisfies the condition in Claim 6.6. As predicted by our framework, Deep Sets has only 21% test accuracy, while GNNs have more than 90% accuracy.

6.3.3 Dynamic Programming

We observe that a broad class of relational reasoning tasks can be unified by the powerful algorithmic paradigm *dynamic programming (DP)* [Bellman, 1966]. DP recursively breaks down a problem into simpler sub-problems. It has the following general form:

$$\text{Answer}[k][i] = \text{DP-Update}(\{\text{Answer}[k-1][j]\}, j = 1...n), \tag{6.5}$$

where Answer$[k][i]$ is the solution to the sub-problem indexed by iteration k and state i, and DP-Update is an task-specific update function that computes Answer$[k][i]$ from Answer$[k-1][j]$'s.

GNNs algorithmically align with a class of DP algorithms. We can interpret GNN as a DP algorithm, where node representations $h_i^{(k)}$ are Answer$[k][i]$, and the GNN aggregation step is the DP-Update. Therefore, Theorem 6.4 suggests that a GNN with enough iterations can sample efficiently learn any DP algorithm with a simple DP-update function, e.g. sum/min/max.

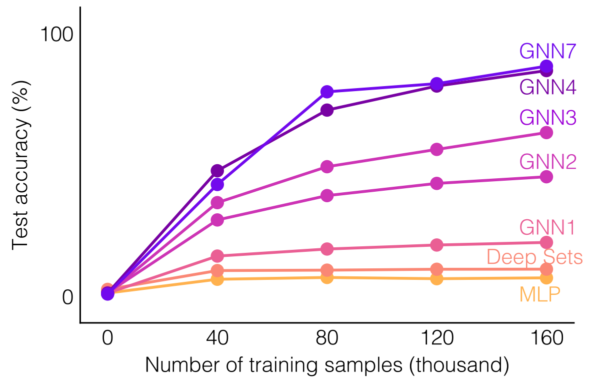


Figure 6-4: **Test accuracy vs. training set size** for models trained on sub-sampled training sets and evaluated on the same test set of monster trainer (DP task). Test accuracies increase faster when a neural network aligns well with an algorithmic solution of the task. For example, the test accuracy of GNN4 increases by 23% when the number of training samples increases from $40,000$ to $80,000$, which is much higher than that of Deep Sets (0.2%).

Shortest paths. As an example, we experiment with GNN on Shortest paths, a standard DP problem. Shortest paths can be solved by the Bellman-Ford algorithm [Bellman, 1958], which recursively updates the minimum distance between each object u and the source s:

$$\text{distance}[1][u] = \text{cost}(s, u), \quad \text{distance}[k][u] = \min_v \Big\{\text{distance}[k-1][v] + \text{cost}(v, u)\Big\}, \tag{6.6}$$

As discussed above, GNN aligns well with this DP algorithm. Therefore, our framework predicts that GNN has good sample complexity when learning to find shortest paths. To verify this, we test different models on a **monster trainer** game, which is a shortest path variant with unkown cost functions that need to be learned by the models. Appendix D.2.3 describes the task in details.

In Figure 6-3c, only GNNs with at least four iterations generalize well. The empirical result confirms our theory: a neural network can sample efficiently learn a task if it aligns with a correct algorithm. Interestingly, GNN does not need as many iterations as Bellman-Ford. While Bellman-Ford needs $N = 7$ iterations, GNNs with four iterations have almost identical test accuracy as GNNs with seven iterations (94% vs 95%). This can also be explained through algorithmic alignment, as GNN aligns with an optimized version of Bellman-Ford, which we explain in Appendix D.2.3.

Figure 6-4 shows how the test accuracies of different models vary with the number of sub-sampled training points. Indeed, the test accuracy increases more slowly for models that align worse with the task, which implies they need more training samples to achieve similar generalization performance. Again, this confirms our theory.

After verifying that GNNs can sample-efficiently learn DP, we show that two popular families of reasoning tasks, visual question answering and intuitive physics, can be formulated as DP. Therefore, our framework explains why GNNs are effective in these tasks.

Visual question answering. The Pretty-CLEVR dataset [Palm et al., 2018] is an extension of Sort-of-CLEVR [Santoro et al., 2017] and CLEVR [Johnson et al., 2017a]. GNNs work well on these datasets. Each question in Pretty-CLEVR has state representations and asks "Starting at object X, if each time we jump to the closest object, which object is K jumps away?". This problem can be solved by DP, which computes the answers for k jumps from the answers for $(k-1)$ jumps.

$$\text{closest}[1][i] = \arg\min_j d(i,j), \quad \text{closest}[k][i] = \text{closest}[k-1]\Big[\text{closest}[1][i]\Big] \text{ for } k > 1, \tag{6.7}$$

where closest$[k][i]$ is the answer for jumping k times from object i, and $d(i,j)$ is the distance between the i-th and the j-th object.

Intuitive physics. Battaglia et al. [2016] and Watters et al. [2017] train neural networks to predict object dynamics in rigid body scenes and n-body systems. Chang et al. [2017] and Janner et al. [2019] study other rigid body scenes. If the force acting on a physical object stays constant, we can compute the object's trajectory with simple functions (physics laws) based on its initial position and force. Physical interactions, however, make the force change, which means the function to compute the object's dynamics has to change too. Thus, a DP algorithm would *recursively* compute the next force changes in the system and update DP states (velocity, momentum, position etc of objects) according to the (learned) forces and physics laws [Thijssen, 2007].

$$\text{for } k = 1..K: \quad \text{time} = \min_{i,j} \text{Force-change-time}(\text{state}[k-1, i], \text{state}[k-1, j]), \tag{6.8}$$

$$\text{for } i = 1..N: \quad \text{state}[k][i] = \text{Update-by-forces}(\text{state}[k-1][j], \text{time}), \; j = 1..N, \tag{6.9}$$

Force-change-time computes the time at which the force between object i and j will change. Update-by-forces updates the state of each object at the next force change time. In rigid body systems, force changes only at collision. In datasets where no object collides more than once between time frames, one-iteration algorithm/GNN can work [Battaglia et al., 2016]. More iterations are needed if multiple collisions occur between two consecutive frames [Li and Liang, 2018]. In n-body systems, forces change continuously but smoothly. Thus, finite-iteration DP/GNN can be viewed as a form of Runge-Kutta method [DeVries and Hamill, 1995].

6.3.4 Designing Neural Networks with Algorithmic Alignment

While DP solves many reasoning tasks, it has limitations. For example, NP-hard problems cannot be solved by DP. It follows that GNN also cannot sample-efficiently learn these hard problems. Our framework, however, goes beyond GNNs. If we know the structure of a suitable underlying reasoning algorithm, we can design a network with a similar structure to learn it. If we have no prior knowledge about the structure, then neural architecture search over algorithmic structures will be needed.

Subset Sum. As an example, we design a new architecture that can learn to solve the subset sum problem: Given a set of numbers, does there exist a subset that sums to 0? Subset sum is NP-hard [Karp, 1972] and cannot be solved by DP. Therefore, our framework predicts that GNN cannot generalize on this task. One subset sum algorithm is exhaustive search, where we enumerate all $2^{|S|}$ possible subsets τ and check whether τ has zero-sum.

Following this algorithm, we design a similarly structured neural network which we call **Neural Exhaustive Search (NES)**. Given a universe, NES enumerates all subsets of objects and passes each subset through an LSTM followed by a MLP. The results are aggregated with a max-pooling layer and MLP:

$$\text{MLP}_2(\max_{\tau \subseteq S} \text{MLP}_1 \circ \text{LSTM}(X_1, ..., X_{|\tau|} : X_1, ..., X_{|\tau|} \in \tau)). \tag{6.10}$$

This architecture aligns well with subset-sum, since the first MLP and LSTM only need to learn a simple step, checking whether a subset has zero sum. Therefore, we expect NES to generalize well in this task. Indeed, NES has 98% test accuracy, while other models perform much worse (Figure 6-3d).

6.4 Discussion

This chapter studies the fundamental question of formally understanding how neural networks can learn to reason. In particular, we answer what tasks a neural network can learn to reason about well, by studying the generalization ability of learning the underlying reasoning processes for a task. To this end, we introduce an algorithmic alignment framework to formalize the interaction between the structure of a neural network and a reasoning process, and provide preliminary results on sample complexity. Our results explain the success and suggest the limits of current neural architectures: Graph neural networks generalize in many popular reasoning tasks because the underlying reasoning processes for those tasks resemble dynamic programming.

Our algorithmic alignment perspective may inspire neural network design and opens up theoretical avenues. An interesting direction for future work is to design, e.g. via algorithmic alignment, neural networks that can learn other reasoning paradigms beyond dynamic programming, and to explore the neural architecture search space of algorithmic structures. It is also interesting to apply our insights to deep reinforcement learning. From a broader standpoint, reasoning assumes a good representation of the concepts and objects in the world. To complete the picture, it would also be interesting to understand how to better disentangle and eventually integrate “representation” and “reasoning”.

Since the release of this work, there have been several follow-up works applying the principles of algorithmic alignment. Several works design neural architectures that better align with their respective underlying algorithms [Veličković et al., 2020, Georgiev and Lió, 2020, Puny et al., 2020, Thost and Chen, 2021, Cappart et al., 2021]. Several other works apply the idea to reinforcement learning [Deac et al., 2020, Li and Littman, 2020]. A paper by the author studies the noisy label setting, and shows that neural networks which well align with the signal, but not the noise, learn a good representation (e.g., representations before the last layer) even if the overall accuracy may be bad [Li et al., 2020b].

By this chapter, we have assumed the interpolation regime, where the training and test data are from the same distribution. Hence, nothing has been said yet about what the neural network implements out of the training distribution. In the next part, we will study generalization under the extrapolation regime.

Part III

Reasoning: Extrapolation

Chapter 7

How Neural Networks Extrapolate

This part of the thesis completes the picture of learning to reason with neural networks. We ask what a neural network implements outside the training distribution, and under what conditions it extrapolates well, i.e., generalizes to unseen domains.

7.1 Introduction

Humans extrapolate well in many tasks. For example, we can apply arithmetics to arbitrarily large numbers. One may wonder whether a neural network can do the same and generalize to examples arbitrarily far from the training data [Lake et al., 2017]. Curiously, previous works report mixed extrapolation results with neural networks. Early works demonstrate feedforward neural networks, a.k.a. multilayer perceptrons (MLPs), fail to extrapolate well when learning simple polynomial functions [Barnard and Wessels, 1992, Haley and Soloway, 1992]. However, recent works show Graph Neural Networks (GNNs) [Scarselli et al., 2009], a class of structured networks with MLP building blocks, can generalize to graphs much larger than training graphs in challenging algorithmic reasoning tasks, such as predicting the time evolution of physical systems [Battaglia et al., 2016], learning graph algorithms [Velickovic et al., 2020], and solving mathematical equations [Lample and Charton, 2020].

To explain this puzzle, we formally study how neural networks trained by gradient descent (GD) extrapolate, i.e., what they learn outside the support of training distribution.

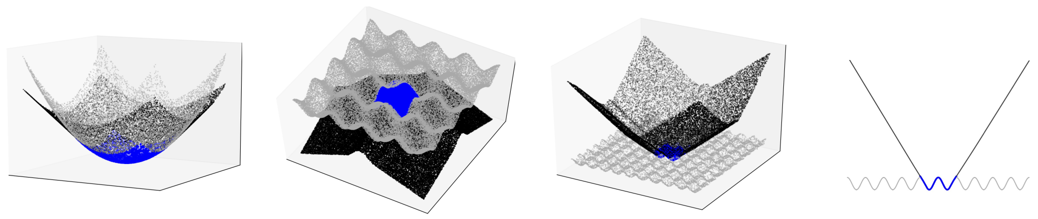
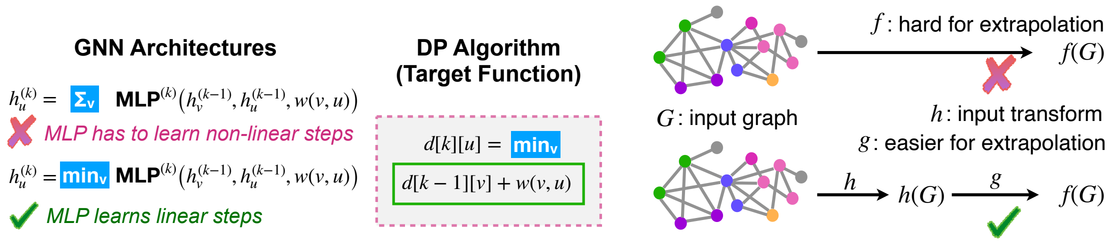


Figure 7-1: **How ReLU MLPs extrapolate.** We train MLPs to learn nonlinear functions (grey) and plot their predictions both within (blue) and outside (black) the training distribution. MLPs converge quickly to linear functions outside the training data range along directions from the origin (Theorem 7.1). Hence, MLPs do not extrapolate well in most nonlinear tasks. But, with appropriate training data, MLPs can provably extrapolate linear target functions (Theorem 7.3).

We say a neural network extrapolates well if it learns a task outside the training distribution. At first glance, it may seem that neural networks can behave arbitrarily outside the training distribution since they have high capacity [Zhang et al., 2017] and are universal approximators [Cybenko, 1989, Funahashi, 1989, Hornik et al., 1989, Kurková, 1992]. However, neural networks are constrained by gradient descent training [Hardt et al., 2016, Soudry et al., 2018]. In our analysis, we explicitly consider such implicit bias through the analogy of the training dynamics of over-parameterized neural networks and kernel regression via the *neural tangent kernel (NTK)* [Jacot et al., 2018].

Starting with feedforward networks, the simplest neural networks and building blocks of more complex architectures such as GNNs, we establish that the predictions of over-parameterized MLPs with ReLU activation trained by GD converge to linear functions along any direction from the origin. We prove a convergence rate for two-layer networks and empirically observe that convergence often occurs *close to* the training data (Figure 7-1), which suggests ReLU MLPs cannot extrapolate well for most nonlinear tasks. We emphasize that our results do not follow from the fact that ReLU networks have finitely many linear regions [Arora et al., 2018a, Hanin and Rolnick, 2019, Hein et al., 2019]. While having finitely many linear regions implies ReLU MLPs *eventually* become linear, it does not say whether MLPs will learn the correct target function close to the training distribution. In contrast, our results are non-asymptotic and quantify what kind of functions MLPs will learn close to the training distribution. Second, we identify a condition when MLPs extrapolate well: the task is linear and the geometry of the training distribution is sufficiently "diverse".


GNN Architectures
$h_u^{(k)} = \Sigma_v$ MLP$^{(k)}(h_v^{(k-1)}, h_u^{(k-1)}, w(v,u))$
MLP has to learn non-linear steps
$h_u^{(k)} = \min_v$ MLP$^{(k)}(h_v^{(k-1)}, h_u^{(k-1)}, w(v,u))$
MLP learns linear steps
DP Algorithm (Target Function)
$d[k][u] = \min_v$
$d[k-1][v] + w(v,u)$
f: hard for extrapolation
$f(G)$
G: input graph
h: input transform
g: easier for extrapolation
h
$h(G)$
g
$f(G)$

(a) Network architecture (b) Input representation.

Figure 7-2: **How GNNs extrapolate.** Since MLPs can extrapolate well when learning linear functions, we hypothesize that GNNs can extrapolate well in dynamic programming (DP) tasks if we encode appropriate non-linearities in the architecture (left) and input representation (right; through domain knowledge or representation learning). The encoded non-linearities may not be necessary for interpolation, as they can be approximated by MLP modules, but they help extrapolation. We support the hypothesis theoretically (Theorem 8.2) and empirically (Figure 8-1).

To our knowledge, our results are the first extrapolation results of this kind for feedforward neural networks.

We then relate our insights into feedforward neural networks to GNNs, to explain why GNNs extrapolate well in some algorithmic tasks. Prior works report successful extrapolation for tasks that can be solved by dynamic programming (DP) [Bellman, 1966], which has a computation structure aligned with GNNs [Xu et al., 2020]. DP updates can often be decomposed into nonlinear and linear steps. Hence, we hypothesize that GNNs trained by GD can extrapolate well in a DP task, if we encode appropriate non-linearities in the *architecture* and *input representation* (Figure 7-2). Importantly, encoding non-linearities may be unnecessary for GNNs to *interpolate*, because the MLP modules can easily learn many nonlinear functions inside the training distribution as we have seen in the previous chapter [Cybenko, 1989, Hornik et al., 1989, Xu et al., 2020], but it is crucial for GNNs to *extrapolate* correctly. We prove this hypothesis for a simplified case using *Graph NTK*, which we have developed in Chapter 5 [Du et al., 2019b]. Empirically, we validate the hypothesis on three DP tasks: max degree, shortest paths, and n-body problem. We show GNNs with appropriate architecture, input representation, and training distribution can predict well on graphs with unseen sizes, structures, edge weights, and node features. Our theory explains the empirical success in previous works and suggests their limitations:

successful extrapolation relies on encoding task-specific non-linearities, which requires domain knowledge or extensive model search. From a broader standpoint, our insights go beyond GNNs and apply broadly to other neural networks.

To summarize, we study how neural networks extrapolate. First, ReLU MLPs trained by GD converge to linear functions along directions from the origin with a rate of $O(1/t)$. Second, to explain why GNNs extrapolate well in some algorithmic tasks, we prove that ReLU MLPs can extrapolate well in linear tasks, leading to a hypothesis: a neural network can extrapolate well when appropriate non-linearities are encoded into the architecture and features. We prove this hypothesis for a simplified case and provide empirical support for more general settings.

Related work. Early works show example tasks where MLPs do not extrapolate well, e.g. learning simple polynomials [Barnard and Wessels, 1992, Haley and Soloway, 1992]. We instead show a *general* pattern of how ReLU MLPs extrapolate and identify conditions for MLPs to extrapolate well. More recent works study the implicit biases induced on MLPs by gradient descent, for both the NTK and mean field regimes [Bietti and Mairal, 2019, Chizat and Bach, 2018, Song et al., 2018]. Related to our results, some works show MLP predictions converge to "simple" piecewise linear functions, e.g., with few linear regions [Hanin and Rolnick, 2019, Maennel et al., 2018, Savarese et al., 2019, Williams et al., 2019]. Our work differs in that none of these works explicitly studies extrapolation, and some focus only on one-dimensional inputs. Recent works also show that in high-dimensional settings of the NTK regime, MLP is asymptotically at most a linear predictor in certain scaling limits [Ba et al., 2020, Ghorbani et al., 2019]. We study a different setting (extrapolation), and our analysis is non-asymptotic in nature and does not rely on random matrix theory.

Prior works explore GNN extrapolation by testing on larger graphs [Battaglia et al., 2018, Santoro et al., 2018, Saxton et al., 2019, Velickovic et al., 2020]. We are the first to theoretically study GNN extrapolation, and we complete the notion of extrapolation to include unseen features and structures.

7.1.1 Problem Setting

We begin by introducing our setting in this chapter. Let $\mathcal{X}$ be the domain of interest, e.g., vectors or graphs. The task is to learn an underlying function (e.g., reasoning process) $g : \mathcal{X} \to \mathbb{R}$ with a training set $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n \subset \mathcal{D}$, where $y_i = g(\boldsymbol{x}_i)$ and $\mathcal{D}$ is the support of training distribution. Previous works have extensively studied in-distribution generalization where the training and the test distributions are identical [Valiant, 1984, Vapnik, 2013]; i.e., $\mathcal{D} = \mathcal{X}$. In contrast, extrapolation addresses predictions on a domain $\mathcal{X}$ that is larger than the support of the training distribution $\mathcal{D}$. We will say a model *extrapolates well* if it has a small *extrapolation error*.

Definition 7.1. (Extrapolation error). Let $f : \mathcal{X} \to \mathbb{R}$ be a model trained on $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n \subset \mathcal{D}$ with underlying function $g : \mathcal{X} \to \mathbb{R}$. Let $\mathcal{P}$ be a distribution over $\mathcal{X} \setminus \mathcal{D}$ and let $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ be a loss function. We define the extrapolation error of f as $\mathbb{E}_{\boldsymbol{x} \sim \mathcal{P}}[\ell(f(\boldsymbol{x}), g(\boldsymbol{x}))]$.

We focus on neural networks trained by *gradient descent* (GD) or its variants with *squared loss*. We study two network architectures: feedforward and graph neural networks. We will refer to the following GNN architecture in our analysis in this chapter.

$$\boldsymbol{h}_u^{(k)} = \sum_{v \in \mathcal{N}(u)} \text{MLP}^{(k)}\Big(\boldsymbol{h}_u^{(k-1)}, \boldsymbol{h}_v^{(k-1)}, \boldsymbol{w}_{(v,u)}\Big), \quad \boldsymbol{h}_G = \text{MLP}^{(K+1)}\Big(\sum_{u \in G} \boldsymbol{h}_u^{(K)}\Big). \tag{7.1}$$

Here, $\boldsymbol{h}_u^{(k)}$ stands for the representations of node u at iteration k, $\boldsymbol{w}_{(u,v)}$ is the edge feature of (u, v), and $\mathcal{N}(u)$ is the set of nodes adjacent to u.

7.2 How Feedforward Neural Networks Extrapolate

Feedforward networks are the simplest neural networks and building blocks of more complex architectures such as GNNs, so we first study how they extrapolate when trained by GD. Throughout this part of the thesis, we assume ReLU activation. Section 7.2.3 contains preliminary results for other activations.

7.2.1 Linear Extrapolation Behavior of ReLU MLPs

By architecture, ReLU networks learn piecewise linear functions, but what do these regions precisely look like outside the support of the training data? Figure 7-1 illustrates examples of how ReLU MLPs extrapolate when trained by GD on various nonlinear functions. These examples suggest that outside the training support, the predictions quickly become linear along directions from the origin. We systematically verify this pattern by linear regression on MLPs' predictions: the coefficient of determination (R^2) is always greater than 0.99 (Appendix E.3.2). That is, ReLU MLPs “linearize" almost immediately outside the training data range.

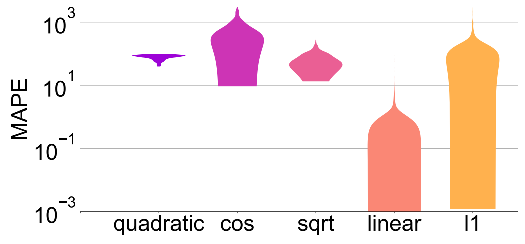
We formalize this observation using the implicit biases of neural networks trained by GD via the *neural tangent kernel (NTK)*: optimization trajectories of over-parameterized networks trained by GD are equivalent to those of kernel regression with a specific neural tangent kernel, under a set of assumptions called the “NTK regime” [Jacot et al., 2018]. We provide an informal definition here; for further details, we refer the readers to Jacot et al. [2018] and Appendix E.1.

Definition 7.2. (Informal) A neural network trained in the *NTK regime* is infinitely wide, randomly initialized with certain scaling, and trained by GD with infinitesimal steps.

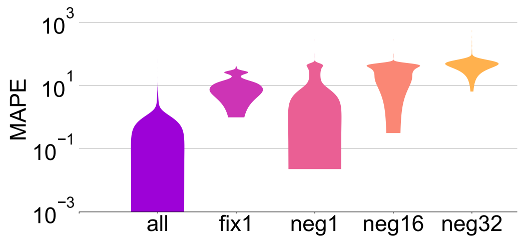
Prior works analyze optimization and in-distribution generalization of over-parameterized neural networks via NTK [Allen-Zhu et al., 2019a,b, Arora et al., 2019b,c, Cao and Gu, 2019, Du et al., 2019c,a, Li and Liang, 2018, Nitanda and Suzuki, 2021]. We instead analyze extrapolation.

Theorem 7.1 formalizes our observation from Figure 7-1: outside the training data range, along any direction $t\boldsymbol{v}$ from the origin, the prediction of a two-layer ReLU MLP quickly converges to a linear function with rate $O(\frac{1}{t})$. The linear coefficients $\boldsymbol{\beta}_v$ and the constant terms in the convergence rate depend on the training data and direction $\boldsymbol{v}$. The proof is in Appendix E.2.1.

Theorem 7.1. *(Linear extrapolation). Suppose we train a two-layer ReLU MLP $f : \mathbb{R}^d \to \mathbb{R}$ with squared loss in the NTK regime. For any direction $\boldsymbol{v} \in \mathbb{R}^d$, let $\boldsymbol{x}_0 = t\boldsymbol{v}$. As $t \to \infty$,*



(a) Different target functions



(b) Training distributions for linear target

Figure 7-3: **Extrapolation performance of ReLU MLPs.** We plot the distributions of MAPE (mean absolute percentage error) of MLPs trained with various hyperparameters (depth, width, learning rate, batch size). (a) Learning different target functions; (b) Different training distributions for learning linear target functions: "all" covers all directions, "fix1" has one dimension fixed to a constant, and "neg$d$" has d dimensions constrained to negative values. ReLU MLPs generally do not extrapolate well unless the target function is linear along each direction (Figure 7-3a), and extrapolate linear target functions if the training distribution covers sufficiently many directions (Figure 7-3b).

$f(\boldsymbol{x}_0 + h\boldsymbol{v}) - f(\boldsymbol{x}_0) \to \beta_{\boldsymbol{v}} \cdot h$ for any $h > 0$, where $\beta_{\boldsymbol{v}}$ is a constant linear coefficient. Moreover, given $\epsilon > 0$, for $t = O(\frac{1}{\epsilon})$, we have $|\frac{f(\boldsymbol{x}_0 + h\boldsymbol{v}) - f(\boldsymbol{x}_0)}{h} - \beta_{\boldsymbol{v}}| < \epsilon$.

ReLU networks have finitely many linear regions [Arora et al., 2018a, Hanin and Rolnick, 2019], hence their predictions *eventually* become linear. In contrast, Theorem 7.1 is a more fine-grained analysis of *how* MLPs extrapolate and provides a convergence rate. While Theorem 7.1 assumes two-layer networks in the NTK regime, experiments confirm that the linear extrapolation behavior happens across networks with different depths, widths, learning rates, and batch sizes (Appendix E.3.1 and E.3.2). Our proof technique potentially also extends to deeper networks.

Theorem 7.1 implies which target functions a ReLU MLP may be able to match outside the training data: only functions that are almost-linear along the directions away from the origin. Indeed, Figure 7-3a shows ReLU MLPs do not extrapolate target functions such as $\boldsymbol{x}^\top A \boldsymbol{x}$ (quadratic), $\sum_{i=1}^{d} \cos(2\pi \cdot \boldsymbol{x}^{(i)})$ (cos), and $\sum_{i=1}^{d} \sqrt{\boldsymbol{x}^{(i)}}$ (sqrt), where $\boldsymbol{x}^{(i)}$ is the i-th dimension of $\boldsymbol{x}$. With suitable hyperparameters, MLPs extrapolate the L1 norm correctly, which satisfies the directional linearity condition.

Figure 7-3a provides one more positive result: MLPs extrapolate linear target functions well, across many different hyperparameters. While learning linear functions may seem

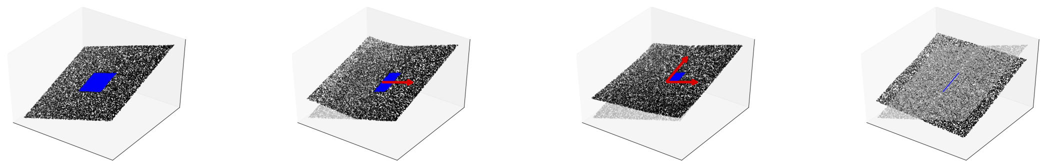


Figure 7-4: **Conditions for ReLU MLPs to extrapolate well.** We train MLPs to learn linear functions (grey) with different training distributions (blue) and plot out-of-distribution predictions (black). Following Theorem 7.3, MLPs extrapolate well when the training distribution (blue) has support in all directions (first panel), but not otherwise: in the two middle panels, some dimensions of the training data are constrained to be positive (red arrows); in the last panel, one dimension is a fixed constant.

very limited at first, in Section 8.1 this insight will help explain extrapolation properties of GNNs in non-linear practical tasks. Before that, we first theoretically analyze when MLPs extrapolate well.

7.2.2 When ReLU MLPs Provably Extrapolate Well

Figure 7-3a shows that MLPs can extrapolate well when the target function is linear. However, this is not always true. In this section, we show that successful extrapolation depends on the *geometry* of training data. Intuitively, the training distribution must be "diverse" enough for correct extrapolation.

We provide two conditions that relate the geometry of the training data to extrapolation. Lemma 7.2 states that over-parameterized MLPs can learn a linear target function with only $2d$ examples.

Lemma 7.2. *Let $g(\boldsymbol{x}) = \boldsymbol{\beta}^\top \boldsymbol{x}$ be the target function for $\boldsymbol{\beta} \in \mathbb{R}^d$. Suppose $\{\boldsymbol{x}_i\}_{i=1}^n$ contains an orthogonal basis $\{\hat{\boldsymbol{x}}_i\}_{i=1}^d$ and $\{-\hat{\boldsymbol{x}}_i\}_{i=1}^d$. If we train a two-layer ReLU MLP f on $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ with squared loss in the NTK regime, then $f(\boldsymbol{x}) = \boldsymbol{\beta}^\top \boldsymbol{x}$ for all $\boldsymbol{x} \in \mathbb{R}^d$.*

Lemma 7.2 is mainly of theoretical interest, as the $2d$ examples need to be carefully chosen. Theorem 7.3 builds on Lemma 7.2 and identifies a more practical condition for successful extrapolation: if the support of the training distribution covers all directions (e.g., a hypercube that covers the origin), the MLP converges to a linear target function with sufficient training data.

Theorem 7.3. *(Conditions for extrapolation). Let $g(\boldsymbol{x}) = \boldsymbol{\beta}^\top \boldsymbol{x}$ be the target function for $\boldsymbol{\beta} \in \mathbb{R}^d$. Suppose $\{\boldsymbol{x}_i\}_{i=1}^n$ is sampled from a distribution whose support $\mathcal{D}$ contains a connected subset $\mathcal{S}$, where for any non-zero $\boldsymbol{w} \in \mathbb{R}^d$, there exists $k > 0$ so that $k\boldsymbol{w} \in \mathcal{S}$. If we train a two-layer ReLU MLP $f : \mathbb{R}^d \to \mathbb{R}$ on $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ with squared loss in the NTK regime, $f(\boldsymbol{x}) \xrightarrow{p} \boldsymbol{\beta}^\top \boldsymbol{x}$ as $n \to \infty$.*

Experiments: geometry of training data affects extrapolation. The condition in Theorem 7.3 formalizes the intuition that the training distribution must be "diverse" for successful extrapolation, e.g., $\mathcal{D}$ includes all directions. Empirically, the extrapolation error is indeed small when the condition of Theorem 7.3 is satisfied ("all" in Figure 7-3b). In contrast, the extrapolation error is much larger when the training examples are restricted to only some directions (Figure 7-3b and Figure 7-4).

Relating to previous works, Theorem 7.3 suggests why spurious correlations may hurt extrapolation, complementing the causality arguments [Arjovsky et al., 2019, Peters et al., 2016, Rojas-Carulla et al., 2018]. When the training data has spurious correlations, some combinations of features are missing; e.g., camels might only appear in deserts in an image collection. Therefore, the condition for Theorem 7.3 no longer holds, and the model may extrapolate incorrectly. Theorem 7.3 is also analogous to an identifiability condition for linear models, but stricter. We can uniquely identify a linear function if the training data has full (feature) rank. MLPs are more expressive, so identifying the linear target function requires additional constraints.

To summarize, we analyze how ReLU MLPs extrapolate and provide two insights: (1) MLPs cannot extrapolate most nonlinear tasks due to their linear extrapolation (Theorem 7.1); and (2) MLPs extrapolate well when the target function is linear, if the training distribution is "diverse" (Theorem 7.3). In the next section, these results help us understand how more complex networks extrapolate.

7.2.3 MLPs with Other Activation Functions

Before moving on to GNNs, we complete the picture of MLPs with experiments on other activation functions: tanh $\sigma(x) = \tanh(x)$, cosine $\sigma(x) = \cos(x)$ [Lapedes and Farber,

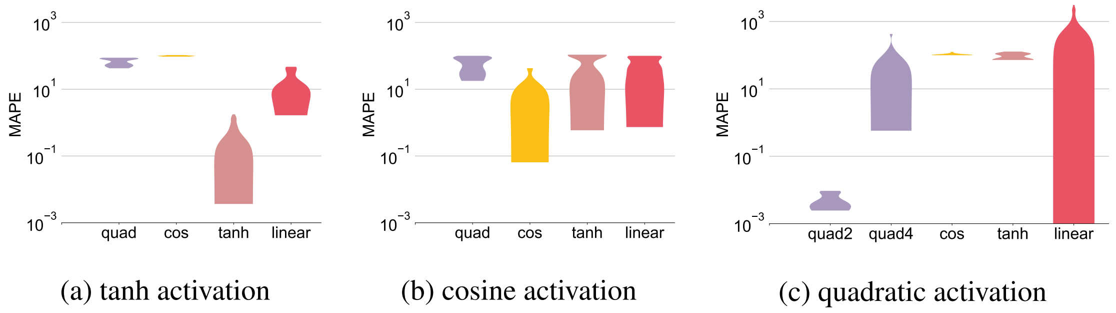


Figure 7-5: **Extrapolation performance of MLPs with other activation**. MLPs can extrapolate well when the activation is "similar" to the target function. When learning quadratic with quadratic activation, 2-layer networks (quad-2) extrapolate well, but 4-layer networks (quad-4) do not.

1987, McCaughan, 1997, Sopena and Alquezar, 1994], and quadratic $\sigma(x) = x^2$ [Du and Lee, 2018, Livni et al., 2014]. Details are in Appendix E.3.4. MLPs extrapolate well when the activation and target function are similar; e.g., tanh activation extrapolates well when learning tanh, but not other functions (Figure 7-5). Moreover, each activation function has different limitations. To extrapolate the tanh function with tanh activation, the training data range has to be sufficiently wide. When learning a quadratic function with quadratic activation, only two-layer networks extrapolate well as more layers lead to higher-order polynomials. Cosine activations are hard to optimize for high-dimensional data, so we only consider one/two dimensional cosine target functions. Related to our work, an improved initialization has shown to help with the optimization of periodic activation like sine and cosine, and they are shown to have good inductive biases for representing fine-grained details of graphics [Sitzmann et al., 2020].

In the next chapter, we will apply our insights into feedforward neural networks to more complex architectures such as GNNs, and to reasoning tasks.

Chapter 8

Extrapolation via Linear Algorithmic Alignment

8.1 How Graph Neural Networks Extrapolate

In the last chapter, we saw that extrapolation in nonlinear tasks is hard for MLPs. Despite this limitation, GNNs have been shown to extrapolate well in some nonlinear reasoning tasks, such as intuitive physics [Battaglia et al., 2016, Janner et al., 2019], graph algorithms [Battaglia et al., 2018, Velickovic et al., 2020], and symbolic mathematics [Lample and Charton, 2020]. To address this discrepancy, we build on our MLP results and study how GNNs trained by GD extrapolate.

We start with an example: training GNNs to solve the shortest path problem. For this task, prior works observe that a modified GNN architecture with min-aggregation can generalize to graphs larger than those in the training set [Battaglia et al., 2018, Velickovic et al., 2020]:

$$\boldsymbol{h}_u^{(k)} = \min_{v \in \mathcal{N}(u)} \text{MLP}^{(k)}\Big(\boldsymbol{h}_u^{(k-1)}, \boldsymbol{h}_v^{(k-1)}, \boldsymbol{w}_{(v,u)}\Big). \tag{8.1}$$

We first provide an intuitive explanation (Figure 7-2a). Shortest path can be solved by the

Bellman-Ford (BF) algorithm [Bellman, 1958] with the following update:

$$d[k][u] = \min_{v \in \mathcal{N}(u)} d[k-1][v] + \boldsymbol{w}(v,u), \tag{8.2}$$

where $\boldsymbol{w}(v,u)$ is the weight of edge (v,u), and $d[k][u]$ is the shortest distance to node u within k steps. The two equations can be easily aligned: GNNs simulate the BF algorithm if its MLP modules learn a linear function $d[k-1][v] + \boldsymbol{w}(v,u)$. Since MLPs can extrapolate linear tasks, this "alignment" may explain why min-aggregation GNNs can extrapolate well in this task.

For comparison, we can reason why we would not expect GNNs with the more commonly used and more expressive sum-aggregation to extrapolate well in this particular task. With sum-aggregation, the MLP modules need to learn a nonlinear function to simulate the BF algorithm, but Theorem 7.1 suggests that they will not extrapolate most nonlinear functions outside the training support.

8.2 Linear Algorithmic Alignment Helps Extrapolation

We can generalize the above intuition to other reasoning tasks. As we have seen in Chapter 6, many tasks where GNNs extrapolate well can be solved by dynamic programming (DP) [Bellman, 1966], an algorithmic paradigm with a recursive structure similar to GNNs' [Xu et al., 2020].

Definition 8.1. *Dynamic programming* (DP) is a recursive procedure with updates

$$\text{Answer}[k][s] = \text{DP-Update}(\{\text{Answer}[k-1][s']\}, s' = 1...n), \tag{8.3}$$

where Answer$[k][s]$ is the solution to a sub-problem indexed by iteration k and state s, and DP-Update is a task-specific update function that solves the sub-problem based on the previous iteration.

From a broader standpoint, we hypothesize that: if we encode appropriate non-linearities into the model architecture and input representations so that the MLP modules only need to learn nearly linear steps, then the resulting neural network can extrapolate well.

Hypothesis 8.1. (Linear algorithmic alignment). Let $f : \mathcal{X} \to \mathbb{R}$ be the underlying function and $\mathcal{N}$ a neural network with m MLP modules. Suppose there exist m *linear* functions $\{g_i\}_{i=1}^m$ so that by replacing $\mathcal{N}$'s MLP modules with g_i's, $\mathcal{N}$ simulates f. Given $\epsilon > 0$, there exists $\{(x_i, f(x_i))\}_{i=1}^n \subset \mathcal{D} \subsetneq \mathcal{X}$ so that $\mathcal{N}$ trained on $\{(x_i, f(x_i))\}_{i=1}^n$ by GD with squared loss learns $\hat{f}$ with $\|\hat{f} - f\| < \epsilon$.

Our hypothesis builds on the algorithmic alignment framework introduced in Chapter 6 [Xu et al., 2020], which states that a neural network *interpolates* well if the modules are "aligned" to easy-to-learn (possibly nonlinear) functions. Successful extrapolation is harder: the modules need to align with linear functions.

Applications of linear algorithmic alignment. In general, linear algorithmic alignment is not restricted to GNNs and applies broadly to neural networks. To satisfy the condition, we can encode appropriate nonlinear operations in the *architecture* or *input representation* (Figure 7-2). Learning DP algorithms with GNNs is one example of encoding non-linearity in the architecture [Battaglia et al., 2018]. Another example is to encode log-and-exp transforms in the architecture to help extrapolate multiplication in arithmetic tasks [Trask et al., 2018, Madsen and Johansen, 2020]. Neural symbolic programs take a step further and encode a library of symbolic operations to help extrapolation [Johnson et al., 2017b, Mao et al., 2019, Yi et al., 2018a].

For some tasks, it may be easier to change the input representation (Figure 7-2b). Sometimes, we can decompose the target function f as $f = g \circ h$ into a feature embedding h and a "simpler" target function g that our model can extrapolate well. We can obtain h via specialized features or feature transforms using *domain knowledge* [Lample and Charton, 2020, Webb et al., 2020], or via *representation learning* (e.g., BERT) with unlabeled out-of-distribution data in $\mathcal{X} \setminus \mathcal{D}$ [Chen et al., 2020a, Devlin et al., 2019, Hu et al., 2020b, Mikolov et al., 2013b, Peters et al., 2018]. This brings a new perspective of how representations help extrapolation in various application areas. For example, in natural language processing, pretrained representations [Mikolov et al., 2013a, Wu and Dredze, 2019] and feature transformation using domain knowledge [Yuan et al., 2020, Zhang et al., 2019b, 2020a] help models generalize across languages, a special type of extrapolation. In quantitative finance, identifying the right "factors" or features is crucial for deep learning models as the financial markets may frequently be in extrapolation regimes [Banz, 1981, Fama and French, 1993, Ross, 1976].

Linear algorithmic alignment explains successful extrapolation in the literature and suggests that extrapolation is harder in general: encoding appropriate non-linearity often requires domain expertise or model search. Next, we provide theoretical and empirical support for our hypothesis.

8.2.1 Theoretical and Empirical Support

We validate our hypothesis on three DP tasks: max degree, shortest path, and n-body problem, and prove the hypothesis for max degree. We highlight the role of graph structures in extrapolation.

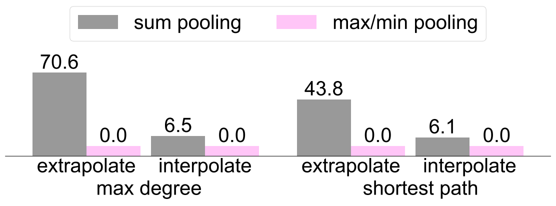
Theoretical analysis. We start with a simple yet fundamental task: learning the max degree of a graph, a special case of DP with one iteration. As a corollary of Theorem 7.1, the commonly used sum-based GNN cannot extrapolate well (proof in Appendix E.2.4).

Corollary 8.1. *GNNs with sum-aggregation and sum-readout do not extrapolate well in Max Degree.*

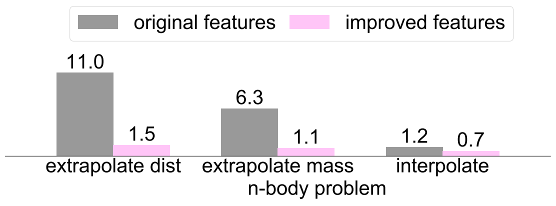
To achieve linear algorithmic alignment, we can encode the only non-linearity, the max function, in the readout. Theorem 8.2 confirms that a GNN with max-readout can extrapolate well in this task.

Theorem 8.2. *(Extrapolation with GNNs). Assume all nodes have the same feature. Let g and g' be the max/min degree function, respectively. Let $\{(G_i, g(G_i)\}_{i=1}^n$ be the training set. If $\{(g(G_i), g'(G_i), g(G_i) \cdot N_i^{\max}, g'(G_i) \cdot N_i^{\min})\}_{i=1}^n$ spans $\mathbb{R}^4$, where $N_i^{\max}$ and $N_i^{\min}$ are the number of nodes that have max/min degree on G_i, then one-layer max-readout GNNs trained on $\{(G_i, g(G_i))\}_{i=1}^n$ with squared loss in the NTK regime learn g.*

Theorem 8.2 does not follow immediately from Theorem 7.3, because MLP modules in GNNs only receive indirect supervision. We analyze the *Graph NTK* [Du et al., 2019b]

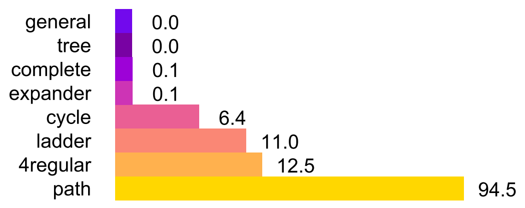


(a) Importance of architecture.

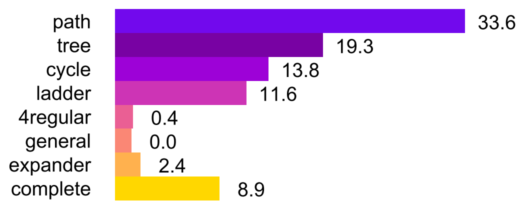


(b) Importance of representation.

Figure 8-1: **Extrapolation for algorithmic tasks.** Each column indicates the task and mean average percentage error (MAPE). Encoding appropriate non-linearity in the architecture or representation is less helpful for *interpolation*, but significantly improves *extrapolation*. Left: In max degree and shortest path, GNNs that appropriately encode max/min extrapolate well, but GNNs with sum-pooling do not. Right: With improved input representation, GNNs extrapolate better for the n-body problem.



(a) Max degree with max-pooling GNN.



(b) Shortest path with min-pooling GNN.

Figure 8-2: **Importance of the training graph structure.** Rows indicate the graph structure covered by the training set and the extrapolation error (MAPE). In max degree, GNNs with max readout extrapolate well if the max/min degrees of the training graphs are not restricted (Theorem 8.2). In shortest path, the extrapolation errors of min GNNs follow a U-shape in the sparsity of the training graphs. More results may be found in Appendix E.4.2.

to prove Theorem 8.2 in Appendix E.2.5. While Theorem 8.2 assumes identical node features, we empirically observe similar results for both identical and non-identical features (Figure E-9 in Appendix).

Interpretation of conditions. The condition in Theorem 8.2 is analogous to that in Theorem 7.3. Both theorems require diverse training data, measured by *graph structure* in Theorem 8.2 or *directions* in Theorem 7.3. In Theorem 8.2, the condition is violated if all training graphs have the same max or min node degrees, e.g., when training data are from one of the following families: path, C-regular graphs (regular graphs with degree C), cycle, and ladder.

Experiments: architectures that help extrapolation. We validate our theoretical analysis with two DP tasks: max degree and shortest path (details in Appendix E.3.5 and E.3.6). While previous works only test on graphs with different sizes [Battaglia et al., 2018, Velickovic et al., 2020], we also test on graphs with unseen structure, edge weights and node features. The results support our theory. For max degree, GNNs with max-readout are better than GNNs with sum-readout (Figure 8-1a), confirming Corollary 8.1 and Theorem 8.2. For shortest path, GNNs with min-readout and min-aggregation are better than GNNs with sum-readout (Figure 8-1a).

Experiments confirm the importance of training graphs structure (Figure 8-2). Interestingly, the two tasks favor different graph structure. For max degree, as Theorem 8.2 predicts, GNNs extrapolate well when trained on trees, complete graphs, expanders, and general graphs, and extrapolation errors are higher when trained on 4-regular, cycles, or ladder graphs. For shortest path, extrapolation errors follow a U-shaped curve as we change the *sparsity* of training graphs (Figure 8-2b and Figure E-11 in Appendix). Intuitively, models trained on sparse or dense graphs likely learn degenerative solutions.

Experiments: representations that help extrapolation. Finally, we show a good input representation helps extrapolation. We study the n-body problem [Battaglia et al., 2016, Watters et al., 2017] (Appendix E.3.7), that is, predicting the time evolution of n objects in a gravitational system. Following previous work, the input is a complete graph where the nodes are the objects [Battaglia et al., 2016]. The node feature for u is the concatenation of the object's mass m_u, position $\boldsymbol{x}_u^{(t)}$, and velocity $\boldsymbol{v}_u^{(t)}$ at time t. The edge features are set to zero. We train GNNs to predict the velocity of each object u at time $t+1$. The true velocity $f(G;u)$ for object u is approximately

$$f(G;u) \approx \boldsymbol{v}_u^t + \boldsymbol{a}_u^t \cdot dt, \quad \boldsymbol{a}_u^t = C \cdot \sum_{v \neq u} \frac{m_v}{\|\boldsymbol{x}_u^t - \boldsymbol{x}_v^t\|_2^3} \cdot \left(\boldsymbol{x}_v^t - \boldsymbol{x}_u^t\right), \tag{8.4}$$

where C is a constant. To learn f, the MLP modules need to learn a nonlinear function. Therefore, GNNs do not extrapolate well to unseen masses or distances ("original features" in Figure 8-1b). We instead use an improved representation $h(G)$ to encode non-linearity. At time t, we transform the edge features of (u,v) from zero to $\boldsymbol{w}_{(u,v)}^{(t)} = m_v \cdot \left(\boldsymbol{x}_v^{(t)} - \boldsymbol{x}_u^{(t)}\right)/\|\boldsymbol{x}_u^{(t)} - \boldsymbol{x}_v^{(t)}\|_2^3$. The new edge features do not add information, but the MLP modules now only need to learn linear functions, which helps extrapolation ("improved features" in Figure 8-1b).

8.3 Connections to Other Out-of-Distribution Settings

We discuss several related settings. Intuitively, from the viewpoint of our results above, methods in related settings may improve extrapolation by 1) learning useful non-linearities beyond the training data range and 2) mapping relevant test data to the training data range.

Domain adaptation studies generalization to a specific target domain [Ben-David et al., 2010, Blitzer et al., 2008, Mansour et al., 2009]. Typical strategies adjust the training process: for instance, use unlabeled samples from the target domain to align the target and source distributions [Ganin et al., 2016, Zhao et al., 2018]. Using target domain data during training may induce useful non-linearities and may mitigate extrapolation by matching the target and source distributions, though the correctness of the learned mapping depends on the label distribution [Zhao et al., 2019].

Self-supervised learning on a large amount of unlabeled data can learn useful non-linearities beyond the labeled training data range [Chen et al., 2020a, Devlin et al., 2019, Peters et al., 2018]. Hence, our results suggest an explanation why pre-trained representations such as BERT improve out-of-distribution robustness [Hendrycks et al., 2020]. In addition, self-supervised learning could map semantically similar data to similar representations, so some out-of-domain examples might fall inside the training distribution after the mapping.

Invariant models aim to learn features that respect specific invariances across multiple training distributions [Arjovsky et al., 2019, Rojas-Carulla et al., 2018, Zhou et al., 2021]. If the model indeed learns these invariances, which can happen in the linear case and when there are confounders or anti-causal variables [Ahuja et al., 2021, Rosenfeld et al., 2021], this may essentially increase the training data range, since variations in the invariant features may be ignored by the model.

Distributional robustness considers small adversarial perturbations of the data distribution, and ensures that the model performs well under these [Goh and Sim, 2010, Sagawa et al., 2020, Sinha et al., 2018, Staib and Jegelka, 2019]. We instead look at more global perturbations. Still, one would expect that modifications that help extrapolation in general also improve robustness to local perturbations.

8.4 Conclusion

This chapter studies the fundamental problem of formally understanding how neural networks trained by gradient descent extrapolate, and suggests implications for reasoning. We identify conditions under which MLPs and GNNs extrapolate as desired. We also suggest an explanation how GNNs may be able to extrapolate well in complex reasoning tasks: encoding appropriate non-linearity in architecture and features can help extrapolation. Our results and hypothesis agree with empirical results, in this work and in the literature.

An interesting future direction is to apply the ideas in this work to design better pre-training algorithms for both representation and reasoning. Other interesting directions include discovering frameworks and nonlinearities that are broadly helpful for reasoning.

Finally, we conclude that artificial intelligence is beyond human intelligence. While human intelligence has much to offer, we have also succeeded in building intelligence that achieves what humans cannot.

Part IV

Optimization

Chapter 9

Convergence and Implicit Acceleration

This part of the thesis complements all previous parts and addresses the optimization of GNNs. Optimization is an essential part of learning, and it closely relates to generalization. In the following two chapters, we study two main questions regarding the training of GNNs. First, we analyze the gradient descent dynamics to show global convergence and implicit acceleration results. We show the training of GNNs is implicitly acclerated by skip connections and more depth. Second, we practically improve the training of GNNs via normalization methods.

9.1 Introduction

We have studied the expressive power, generalization, and extrapolation of GNNs in the previous chapters. So far, the understanding of the optimization properties of GNNs has still remained limited. For example, we have built powerful GNNs in Chapter 3 and 4. Researchers working on this fundamental problem of designing more expressive GNNs hope and often empirically observe that more powerful GNNs better fit the training set [Xu et al., 2019, Sato et al., 2020, Vignac et al., 2020]. However, theoretically, given the non-convexity of GNN training, it is still an open question whether better representational power always translates into smaller training loss. This motivates the more general questions:

Can gradient descent find a global minimum for GNNs?

What affects the speed of convergence in training?

In this chapter, we take an initial step towards answering the questions above by analyzing the trajectory of gradient descent, i.e., *gradient dynamics* or *optimization dynamics*. A complete understanding of the dynamics of GNNs, and deep learning in general, is challenging. Following prior works on gradient dynamics [Saxe et al., 2014, Arora et al., 2019a, Bartlett et al., 2019], we consider the linearized regime, i.e., GNNs with *linear* activation. Despite the linearity, key properties of nonlinear GNNs are present: The objective function is *non-convex* and the dynamics are *nonlinear* [Saxe et al., 2014, Kawaguchi, 2016]. Moreover, we observe the learning curves of linear GNNs and ReLU GNNs are surprisingly similar, both converging to nearly zero training loss at the same linear rate (Figure 9-1). Similarly, prior works report comparable performance in node classification benchmarks even if we remove the non-linearities [Thekumparampil et al., 2018]. Hence, understanding the dynamics of linearized GNNs is a valuable step towards understanding the general GNNs.

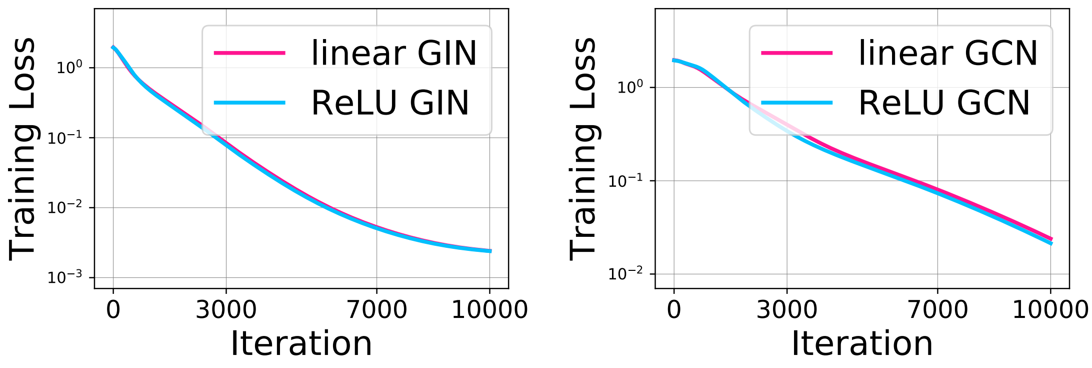


Figure 9-1: **Training curves of linearized GNNs vs. ReLU GNNs** on the Cora node classification dataset.

Our analysis leads to an affirmative answer to the first question. We establish that gradient descent training of a linearized GNN with squared loss converges to a global minimum at a linear rate. Experiments confirm that the assumptions of our theoretical results for global convergence hold on real-world datasets. The most significant contribution of our convergence analysis is on multiscale GNNs, i.e., GNN architectures that use *skip connections* to combine graph features at various scales [Xu et al., 2018, Li et al., 2019, Abu-El-Haija et al., 2020, Chen et al., 2020a, Li et al., 2020a]. The skip connections introduce complex interactions among layers, and thus the resulting dynamics are more intricate. To our knowledge, our results are the first convergence results for GNNs with *more than one* hidden layer, with or without skip connections.

We then study what may affect the training speed of GNNs. First, for any fixed depth, GNNs with skip connections train faster. Second, increasing the depth further accelerates the training of GNNs. Third, faster training is obtained when the labels are more correlated with the graph features, i.e., labels contain "signal" instead of "noise". Overall, experiments for nonlinear GNNs agree with the prediction of our theory for linearized GNNs.

Our results provide the first theoretical justification for the empirical success of multiscale GNNs in terms of optimization, and suggests that deeper GNNs with skip connections may be promising in practice. In the GNN literature, skip connections are initially motivated by the "over-smoothing" problem as we have seen in Chapter 4 [Xu et al., 2018]: via the recursive neighbor aggregation, node representations of a *deep* GNN on expander-like subgraphs would be mixing features from almost the entire graph, and may thereby "wash out" relevant local information. In this case, shallow GNNs may perform better. Multiscale GNNs with *skip connections* can combine and adapt to the graph features at various scales, i.e., the output of intermediate GNN layers, and such architectures are shown to help with this over-smoothing problem [Xu et al., 2018, Li et al., 2019, 2020a, Abu-El-Haija et al., 2020, Chen et al., 2020a]. However, the properties of multiscale GNNs have mostly been understood at a conceptual level. Xu et al. [2018] relate the learned representations to random walk distributions and Oono and Suzuki [2020b] take a boosting view, but they do not consider the optimization dynamics. We give an explanation from the lens of optimization. The training losses of deeper GNNs may be worse due to over-smoothing. In contrast, multiscale GNNs can express any shallower GNNs and fully exploit the power by converging to a global minimum. Hence, our results suggest that deeper GNNs with skip connections are guaranteed to train faster with smaller training losses.

9.2 Convergence Analysis

9.2.1 Problem Setup

We first introduce our notation for this chapter. Let $G = (V, E)$ be a graph with n vertices $V = \{v_1, v_2, \cdots, v_n\}$. Its adjacency matrix $A \in \mathbb{R}^{n\times n}$ has entries $A_{ij} = 1$ if $(v_i, v_j) \in E$ and 0 otherwise. The degree matrix associated with A is $D = \operatorname{diag}(d_1, d_2, \ldots, d_n)$ with $d_i = \sum_{j=1}^{n} A_{ij}$. For any matrix $M \in \mathbb{R}^{m\times m'}$, we denote its j-th column vector by $M_{*j} \in \mathbb{R}^m$, its i-th row vector by $M_{i*} \in \mathbb{R}^{m'}$, and its largest and smallest (i.e., $\min(m, m')$-th largest) singular values by $\sigma_{\max}(M)$ and $\sigma_{\min}(M)$, respectively. The data matrix $X \in \mathbb{R}^{m_x\times n}$ has columns X_{*j} corresponding to the feature vector of node v_j, with input dimension m_x.

The task of interest is node classification or regression. Each node $v_i \in V$ has an associated label $y_i \in \mathbb{R}^{m_y}$. In the transductive (semi-supervised) setting, we have access to training labels for only a subset $\mathcal{I} \subseteq [n]$ of nodes on G, and the goal is to predict the labels for the other nodes in $[n] \setminus \mathcal{I}$. Our problem formulation easily extends to the inductive setting by letting $\mathcal{I} = [n]$, and we can use the trained model for prediction on unseen graphs. Hence, we have access to $\bar{n} = |\mathcal{I}| \le n$ training labels $Y = [y_i]_{i\in\mathcal{I}} \in \mathbb{R}^{m_y\times\bar{n}}$, and we train the GNN using X, Y, G. Additionally, for any $M \in \mathbb{R}^{m\times m'}$, $\mathcal{I}$ may index sub-matrices $M_{*\mathcal{I}} = [M_{*i}]_{i\in\mathcal{I}} \in \mathbb{R}^{m\times\bar{n}}$ (when $m' \ge n$) and $M_{\mathcal{I}*} = [M_{i*}]_{i\in\mathcal{I}} \in \mathbb{R}^{\bar{n}\times m}$ (when $m \ge n$).

We let $X_{(l)} = \left[h^1_{(l)}, h^2_{(l)}, \cdots, h^n_{(l)}\right] \in \mathbb{R}^{m_l\times n}$ denote the hidden features of a GNN, and set $X_{(0)}$ as the input features X. Here, the node hidden representations $X_{(l)}$ are updated by aggregating and transforming the neighbor representations:

$$X_{(l)} = \sigma\Big(B_{(l)} X_{(l-1)} S\Big) \in \mathbb{R}^{m_l\times n}, \tag{9.1}$$

where σ is a nonlinearity such as ReLU, $B_{(l)} \in \mathbb{R}^{m_l\times m_{l-1}}$ is the weight matrix, and $S \in \mathbb{R}^{n\times n}$ is the GNN aggregation matrix, whose formula depends on the exact variant of GNN. In Graph Isomorphism Networks (GIN) [Xu et al., 2019], $S = A + I_n$ is the adjacency matrix of G with self-loop, where $I_n \in \mathbb{R}^{n\times n}$ is an identity matrix. In Graph Convolutional Networks (GCN) [Kipf and Welling, 2017], $S = \hat{D}^{-\frac{1}{2}}(A + I_n)\hat{D}^{-\frac{1}{2}}$ is the normalized adjacency matrix, where $\hat{D}$ is the degree matrix of $A + I_n$.

We first formally define linearized GNNs.

Definition 9.1. (Linear GNN). Given data matrix $X \in \mathbb{R}^{m_x \times n}$, aggregation matrix $S \in \mathbb{R}^{n \times n}$, weight matrices $W \in \mathbb{R}^{m_y \times m_H}$, $B_{(l)} \in \mathbb{R}^{m_l \times m_{l-1}}$, and their collection $B = (B_{(1)}, \ldots, B_{(H)})$, a linear GNN with H layers $f(X, W, B) \in \mathbb{R}^{m_y \times n}$ is defined as

$$f(X, W, B) = W X_{(H)}, \quad X_{(l)} = B_{(l)} X_{(l-1)} S. \tag{9.2}$$

Throughout this chapter, we use multiscale GNNs to refer to Jumping Knowledge Network (JK-Net) [Xu et al., 2018], which connects the output of all intermediate GNN layers to the final layer:

Definition 9.2. (Multiscale linear GNN). Given data $X \in \mathbb{R}^{m_x \times n}$, aggregation matrix $S \in \mathbb{R}^{n \times n}$, weight matrices $W_{(l)} \in \mathbb{R}^{m_y \times m_l}$, $B_{(l)} \in \mathbb{R}^{m_l \times m_{l-1}}$ with $W = (W_{(0)}, W_{(1)}, \ldots, W_{(H)})$, a multiscale linear GNN with H layers $f(X, W, B) \in \mathbb{R}^{m_y \times n}$ is defined as

$$f(X, W, B) = \sum_{l=0}^{H} W_{(l)} X_{(l)}, \tag{9.3}$$

$$X_{(l)} = B_{(l)} X_{(l-1)} S. \tag{9.4}$$

Given a GNN $f(\cdot)$ and a loss function $\ell(\cdot, Y)$, we can train the GNN by minimizing the training loss $L(W, B)$:

$$L(W, B) = \ell\Big(f(X, W, B)_{*\mathcal{I}}, Y\Big), \tag{9.5}$$

where $f(X, W, B)_{*\mathcal{I}}$ corresponds to the GNN's predictions on nodes that have training labels and thus incur training losses. The pair (W, B) represents the trainable weights:

$$L(W, B) = L(W_{(1)}, \ldots, W_{(H)}, B_{(1)}, \ldots, B_{(H)})$$

For completeness, we define the global minimum of GNNs.

Definition 9.3. (Global minimum). For any $H \in \mathbb{N}_0$, L_H^* is the global minimum value of the H-layer linear GNN f:

$$L_H^* = \inf_{W,B} \ell\Big(f(X, W, B)_{*\mathcal{I}}, Y\Big). \tag{9.6}$$

Similarly, we define $L_{1:H}^*$ as the global minimum value of the multiscale linear GNN f with H layers.

We are ready to present our main results on global convergence for linear GNNs and multiscale linear GNNs.

9.2.2 Convergence to Global Minima and Convergence Rates

In this section, we show that gradient descent training a linear GNN or a multiscale linear GNN with squared loss converges linearly to a global minimum. Our conditions for global convergence hold in practice on real-world datasets and provably hold under assumptions, e.g., initialization.

In linearized GNNs, the loss $L(W, B)$ is non-convex (and non-invex) despite the linearity. The graph aggregation S creates interaction among the data and poses additional challenges in the analysis. We show a fine-grained analysis of the GNN's gradient dynamics can overcome these challenges. Following previous works on gradient dynamics [Saxe et al., 2014, Huang and Yau, 2020, Ji and Telgarsky, 2020, Kawaguchi, 2021], we analyze the GNN learning process via the *gradient flow*, i.e., gradient descent with infinitesimal steps: $\forall t \geq 0$, the network weights evolve as

$$\frac{d}{dt}W_t = -\frac{\partial L}{\partial W}(W_t, B_t), \quad \frac{d}{dt}B_t = -\frac{\partial L}{\partial B}(W_t, B_t), \tag{9.7}$$

where (W_t, B_t) represents the trainable parameters at time t with initialization (W_0, B_0).

Linearized GNNs

Theorem 9.1 states our main result on global convergence for linear GNNs.

Theorem 9.1. *Let f be an H-layer linear GNN and $\ell(q, Y) = \|q - Y\|_F^2$ where $q, Y \in \mathbb{R}^{m_y \times \bar{n}}$. Then, for any $T > 0$,*

$$\begin{aligned} &L(W_T, B_T) - L_H^* \\ &\le (L(W_0, B_0) - L_H^*) e^{-4\lambda_T^{(H)} \sigma^2_{\min}(X(S^H)_{*\mathcal{I}})T}, \end{aligned} \tag{9.8}$$

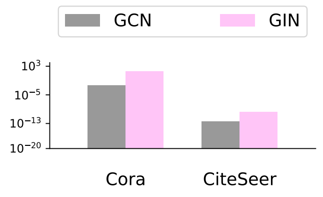
where $\lambda_T^{(H)}$ is the smallest eigenvalue $\lambda_T^{(H)} := \inf_{t \in [0,T]} \lambda_{\min}((\bar{B}_t^{(1:H)})^\top \bar{B}_t^{(1:H)})$ and $\bar{B}^{(1:l)} := B_{(l)} B_{(l-1)} \cdots B_{(1)}$ for any $l \in \{0, \ldots, H\}$ with $\bar{B}^{(1:0)} := I$.

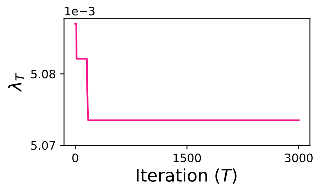
Proof. (Sketch) We decompose the gradient dynamics into three components: the graph interaction, non-convex factors, and convex factors. We then bound the effects of the graph interaction and non-convex factors through $\sigma^2_{\min}(X(S^H)_{*\mathcal{I}})$ and $\lambda_{\min}((\bar{B}_t^{(1:H)})^\top \bar{B}_t^{(1:H)})$ respectively. The complete proof is in Appendix F.1.1. □

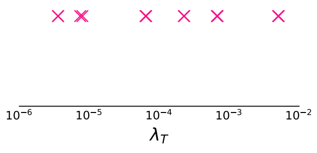
Theorem 9.1 implies that convergence to a global minimum at a linear rate is guaranteed if $\sigma^2_{\min}(X(S^H)_{*\mathcal{I}}) > 0$ and $\lambda_T > 0$. The first condition on the product of X and S^H indexed by $\mathcal{I}$ only depends on the node features X and the GNN aggregation matrix S. It is satisfied if $\text{rank}(X(S^H)_{*\mathcal{I}}) = \min(m_x, \bar{n})$, because $\sigma_{\min}(X(S^H)_{*\mathcal{I}})$ is the $\min(m_x, \bar{n})$-th largest singular value of $X(S^H)_{*\mathcal{I}} \in \mathbb{R}^{m_x \times \bar{n}}$. The second condition $\lambda_T^{(H)} > 0$ is time-dependent and requires a more careful treatment. Linear convergence is implied as long as $\lambda_{\min}((\bar{B}_t^{(1:H)})^\top \bar{B}_t^{(1:H)}) > 0$ and does not converge to 0 for all times t before stopping.

x

Empirical validation of conditions. We verify both the graph condition $\sigma^2_{\min}(X(S^H)_{*\mathcal{I}}) > 0$ and the time-dependent condition $\lambda_T^{(H)} > 0$ for (discretized) $T > 0$. First, on the popular graph datasets, Cora and Citeseer Sen et al. [2008], and the GNN models, GCN Kipf and Welling [2017] and GIN Xu et al. [2019], we have $\sigma^2_{\min}(X(S^H)_{*\mathcal{I}}) > 0$ (Figure 9-2a). Second, we train linear GCN and GIN on Cora and Citeseer to plot an example of how the $\lambda_T^{(H)} = \inf_{t \in [0,T]} \lambda_{\min}((\bar{B}_t^{(1:H)})^\top \bar{B}_t^{(1:H)})$ changes with respect to time T (Figure 9-2b). We further confirm that $\lambda_T^{(H)} > 0$ until convergence, $\lim_{T \to \infty} \lambda_T^{(H)} > 0$ across different settings, e.g., datasets, depths, models (Figure 9-2c). Our experiments use the squared loss, random initialization, learning rate 1e-4, and set the hidden dimension to the input dimension (note







(a) Graph $\sigma^2_{\min}(X(S^H)_{*\mathcal{I}})$ (b) Time-dependent $\lambda_T^{(H)}$ (c) $\lim_{T\to\infty}\lambda_T^{(H)}$

Figure 9-2: **Empirical validation of assumptions for global convergence of linear GNNs.** Left panel confirms the graph condition $\sigma^2_{\min}(X(S^H)_{*\mathcal{I}}) > 0$ for datasets Cora and Citeseer, and for models GCN and GIN. Middle panel shows the time-dependent $\lambda_T^{(H)}$ for one training setting (linear GCN on Cora). Each point in right panel is $\lambda_T^{(H)} > 0$ at the last iteration for different training settings.

that Theorem 9.1 assumes the hidden dimension is at least the input dimension). Further experimental details are in Appendix F.3. Along with Theorem 9.1, we conclude that linear GNNs converge linearly to a global minimum. Empirically, we indeed see both linear and ReLU GNNs converging at the same linear rate to nearly zero training loss in node classification tasks (Figure 9-1).

Guarantee via initialization. Besides the empirical verification, we theoretically show that a *good initialization* guarantees the time-dependent condition $\lambda_T > 0$ for any $T > 0$. Indeed, like other neural networks, GNNs do not converge to a global optimum with certain initializations: e.g., initializing all weights to zero leads to zero gradients and $\lambda_T^{(H)} = 0$ for all T, and hence no learning. We introduce a notion of *singular margin* and say an initialization is good if it has a positive singular margin.

Definition 9.4. (Singular margin). The initialization (W_0, B_0) is said to have singular margin $\gamma > 0$ with respect to a layer $l \in \{1, \dots, H\}$ if $\sigma_{\min}(B_{(l)}B_{(l-1)}\cdots B_{(1)}) \geq \gamma$ for all (W, B) such that $L(W, B) \leq L(W_0, B_0)$.

Proposition 9.2 then states that an initialization with positive singular margin γ guarantees $\lambda_T^{(H)} \geq \gamma^2 > 0$ for all T:

Proposition 9.2. *Let f be a linear GNN with H layers and $\ell(q, Y) = \|q - Y\|_F^2$. If the initialization (W_0, B_0) has singular margin $\gamma > 0$ with respect to the layer H and $m_H \geq m_x$,* *then* $\lambda_T^{(H)} \geq \gamma^2$ *for all* $T \in [0, \infty)$.

Proposition 9.2 follows since $L(W_t, B_t)$ is non-increasing with respect to time t (proof in Appendix F.1.2).

Relating to previous works, our singular margin is a generalized variant of the deficiency margin of linear feedforward networks [Arora et al., 2019a, Definition 2 and Theorem 1]:

Proposition 9.3. *(Informal) If initialization* (W_0, B_0) *has deficiency margin* $c > 0$*, then it has singular margin* $\gamma > 0$.

The formal version of Proposition 9.3 is in Appendix F.1.3.

To summarize, Theorem 9.1 along with Proposition 9.2 implies that we have a prior guarantee of linear convergence to a global minimum for any graph with $\text{rank}(X(S^H)_{*\mathcal{I}}) = \min(m_x, \bar{n})$ and initialization (W_0, B_0) with singular margin $\gamma > 0$: i.e., for any desired $\epsilon > 0$, we have that $L(W_T, B_T) - L_H^* \leq \epsilon$ for any T such that

$$T \geq \frac{1}{4\gamma^2 \sigma_{\text{min}}^2(X(S^H)_{*\mathcal{I}})} \log \frac{L(A_0, B_0) - L_H^*}{\epsilon}. \tag{9.9}$$

While the margin condition theoretically guarantees linear convergence, empirically, we have already seen that the convergence conditions of across different training settings for widely used random initialization.

Theorem 9.1 suggests that the convergence rate depends on a combination of data features X, the GNN architecture and graph structure via S and H, the label distribution and initialization via λ_T. For example, GIN has better such constants than GCN on the Cora dataset with everything else held equal (Figure 9-2a). Indeed, in practice, GIN converges faster than GCN on Cora (Figure 9-1). In general, the computation and comparison of the rates given by Theorem 9.1 requires computation such as those in Figure 9-2. In Section 9.3, we will study an alternative way of comparing the speed of training by directly comparing the gradient dynamics.

Multiscale Linear GNNs

Without skip connections, the GNNs under linearization still behave like linear feedforward networks with augmented graph features. With skip connections, the dynamics and analysis become much more intricate. The expressive power of multiscale linear GNNs changes significantly as depth increases. Moreover, the skip connections create complex interactions among different layers and graph structures of various scales in the optimization dynamics. Theorem 9.4 states our convergence results for multiscale linear GNNs in three cases: (i) a general form; (ii) a weaker condition for boundary cases that uses $\lambda_T^{H'}$ instead of $\lambda_T^{1:H}$; (iii) a faster rate if we have monotonic expressive power as depth increases.

Theorem 9.4. *Let f be a multiscale linear GNN with H layers and $\ell(q, Y) = \|q - Y\|_F^2$ where $q, Y \in \mathbb{R}^{m_y \times \bar{n}}$. Let $\lambda_T^{(1:H)} := \min_{0 \le l \le H} \lambda_T^{(l)}$. For any $T > 0$, the following hold:*

(i) (General). *Let $G_H := [X^\top, (XS)^\top, \ldots, (XS^H)^\top]^\top \in \mathbb{R}^{(H+1)m_x \times n}$. Then*

$$L(W_T, B_T) - L_{1:H}^* \le (L(W_0, B_0) - L_{1:H}^*) e^{-4\lambda_T^{(1:H)} \sigma_{\min}^2((G_H)_{*\mathcal{I}})T}. \tag{9.10}$$

(ii) (Boundary cases). *For any $H' \in \{0, 1, \ldots, H\}$,*

$$L(W_T, B_T) - L_{H'}^* \le (L(W_0, B_0) - L_{H'}^*) e^{-4\lambda_T^{(H')} \sigma_{\min}^2(X(S^{H'})_{*\mathcal{I}})T}. \tag{9.11}$$

(iii) (Monotonic expressive power). *If there exist $l, l' \in \{0, \ldots, H\}$ with $l < l'$ such that $L_l^* \ge L_{l+1}^* \ge \cdots \ge L_{l'}^*$ or $L_l^* \le L_{l+1}^* \le \cdots \le L_{l'}^*$, then*

$$L(W_T, B_T) - L_{l''}^* \le (L(W_0, B_0) - L_{l''}^*) e^{-4\sum_{k=l}^{l'} \lambda_T^{(k)} \sigma_{\min}^2(X(S^k)_{*\mathcal{I}})T}, \tag{9.12}$$

where $l'' = l$ if $L_l^* \ge L_{l+1}^* \ge \cdots \ge L_{l'}^*$, and $l'' = l'$ if $L_l^* \le L_{l+1}^* \le \cdots \le L_{l'}^*$.

Proof. (Sketch) A key observation in our proof is that the interactions of different scales cancel out to point towards a specific direction in the gradient dynamics induced in a space of the loss value. The complete proof is in Appendix F.1.4. □

Similar to Theorem 9.1 for linear GNNs, the most general form (i) of Theorem 9.4 implies that convergence to the global minimum value of the *entire* multiscale linear GNN $L^*_{1:H}$ at linear rate is guaranteed when $\sigma^2_{\min}((G_H)_{*\mathcal{I}}) > 0$ and $\lambda_T^{(1:H)} > 0$. The graph condition $\sigma^2_{\min}((G_H)_{*\mathcal{I}}) > 0$ is satisfied if $\operatorname{rank}((G_H)_{*\mathcal{I}}) = \min(m_x(H+1), \bar{n})$. The time-dependent condition $\lambda_T^{(1:H)} > 0$ is guaranteed if the initialization (W_0, B_0) has singular margin $\gamma > 0$ with respect to *every* layer (Proposition 9.5 is proved in Appendix F.1.5):

Proposition 9.5. *Let f be a multiscale linear GNN and $\ell(q, Y) = \|q - Y\|_F^2$. If the initialization (W_0, B_0) has singular margin $\gamma > 0$ with respect to every layer $l \in [H]$ and $m_l \geq m_x$ for $l \in [H]$, then $\lambda_T^{(1:H)} \geq \gamma^2$ for all $T \in [0, \infty)$.*

We demonstrate that the conditions of Theorem 9.4 (i) hold for real-world datasets, suggesting in practice multiscale linear GNNs converge linearly to a global minimum.

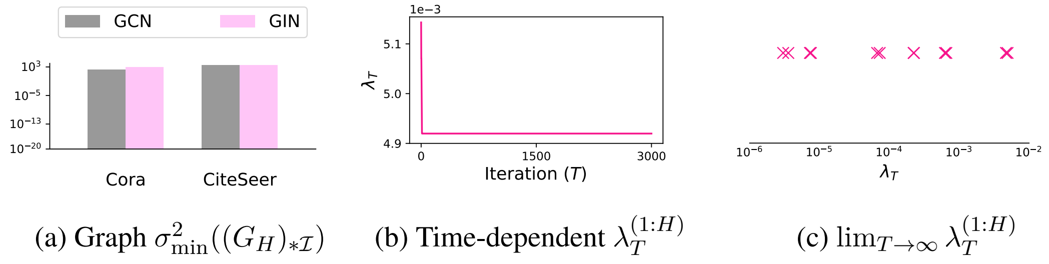


Figure 9-3: **Empirical validation of assumptions for global convergence of multiscale linear GNNs.** Left panel confirms the graph condition $\sigma^2_{\min}((G_H)_{*\mathcal{I}}) > 0$ for Cora and Citeseer, and for GCN and GIN. Middle panel shows the time-dependent $\lambda_T^{(1:H)}$ for one training setting (multiscale linear GCN on Cora). Each point in right panel is $\lambda_T^{(1:H)} > 0$ at the last iteration for different training settings.

Empirical validation of conditions. On datasets Cora and Citeseer and for GNN models GCN and GIN, we confirm that $\sigma^2_{\min}((G_H)_{*\mathcal{I}}) > 0$ (Figure 9-3a). Moreover, we train multiscale linear GCN and GIN on Cora and Citeseer to plot an example of how the $\lambda_T^{(1:H)}$ changes with respect to time T (Figure 9-3b), and we confirm that at convergence, $\lambda_T^{(1:H)} > 0$ across different settings (Figure 9-3c). Experimental details are in Appendix F.3.

Boundary cases. Because the global minimum value of multiscale linear GNNs $L^*_{1:H}$ can be smaller than that of linear GNNs L^*_H, the conditions in Theorem 9.4(i) may sometimes be stricter than those of Theorem 9.1. For example, in Theorem 9.4(i), we require $\lambda_T^{(1:H)} := \min_{0\le l\le H} \lambda_T^{(l)}$ rather than $\lambda_T^{(H)}$ to be positive. If $\lambda_T^{(l)} = 0$ for some l, then Theorem 9.4(i) will not guarantee convergence to $L^*_{1:H}$.

Although the boundary cases above did not occur on the tested real-world graphs (Figure 9-3), for theoretical interest, Theorem 9.4(ii) guarantees that in such cases, multiscale linear GNNs still converge to a value no worse than the global minimum value of *non-multiscale* linear GNNs. For any intermediate layer H', assuming $\sigma^2_{\min}(X(S^{H'})_{*\mathcal{I}}) > 0$ and $\lambda_T^{(H')} > 0$, Theorem 9.4(ii) bounds the loss of the multiscale linear GNN $L(W_T, B_T)$ at convergence by the global minimum value $L^*_{H'}$ of the corresponding linear GNN with H' layers.

Faster rate under monotonic expressive power. Theorem 9.4(iii) considers a special case that is likely in real graphs: the global minimum value of the non-multiscale linear GNN $L^*_{H'}$ is *monotonic* as H' increases. Then (iii) gives a *faster rate* than (ii) and linear GNNs. For example, if the globally optimal value decreases as linear GNNs get deeper. i.e., $L^*_0 \ge L^*_1 \ge \cdots \ge L^*_H$, or vice versa, $L^*_0 \le L^*_1 \le \cdots \le L^*_H$, then Theorem 9.4 (i) implies that

$$\begin{aligned} &L(W_T, B_T) - L^*_l \\ &\le (L(W_0, B_0) - L^*_l) e^{-4\sum_{k=0}^{H} \lambda_T^{(k)} \sigma^2_{\min}(X(S^k)_{*\mathcal{I}})T}, \end{aligned} \tag{9.13}$$

where $l = 0$ if $L^*_0 \ge L^*_1 \ge \cdots \ge L^*_H$, and $l = H$ if $L^*_0 \le L^*_1 \le \cdots \le L^*_H$. Moreover, if the globally optimal value does not change with respect to the depth as $L^*_{1:H} = L^*_1 = L^*_2 = \cdots = L^*_H$, then we have

$$\begin{aligned} &L(W_T, B_T) - L^*_{1:H} \\ &\le (L(W_0, B_0) - L^*_{1:H}) e^{-4\sum_{k=0}^{H} \lambda_T^{(k)} \sigma^2_{\min}(X(S^k)_{*\mathcal{I}})T}. \end{aligned} \tag{9.14}$$

We obtain a faster rate for multiscale linear GNNs than for linear GNNs, as

$$e^{-4\sum_{k=0}^{H} \lambda_T^{(k)} \sigma^2_{\min}(X(S^k)_{*\mathcal{I}})T} \le e^{-4\lambda_T^{(H)} \sigma^2_{\min}(X(S^H)_{*\mathcal{I}})T}.$$

Interestingly, unlike linear GNNs, multiscale linear GNNs in this case do not require any condition on initialization to obtain a prior guarantee on global convergence since $e^{-4\sum_{k=0}^{H}\lambda_T^{(k)}\sigma^2_{\min}(X(S^k)_{*\mathcal{I}})T} \leq e^{-4\lambda_T^{(0)}\sigma^2_{\min}(X(S^0)_{*\mathcal{I}})T}$ with $\lambda_T^{(0)} = 1$ and $X(S^0)_{*\mathcal{I}} = X_{*\mathcal{I}}$.

To summarize, we prove global convergence rates for multiscale linear GNNs (Thm. 9.4(i)) and experimentally validate the conditions. Part (ii) addresses boundary cases where the conditions of Part (i) do not hold. Part (iii) gives faster rates assuming monotonic expressive power with respect to depth. So far, we have shown multiscale linear GNNs converge faster than linear GNNs in the case of (iii). Next, we compare the training speed for more general cases.

9.3 Implicit Acceleration

In this section, we study how the multiscale skip connections, depth of GNN, and label distribution may affect the speed of training for GNNs. Similar to previous works [Arora et al., 2018b], we compare the training speed by comparing the per step loss reduction $\frac{d}{dt}L(W_t, B_t)$ for *arbitrary* differentiable loss functions $\ell(\cdot, Y) : \mathbb{R}^{m_y} \to \mathbb{R}$. Smaller $\frac{d}{dt}L(W_t, B_t)$ implies faster training. Loss reduction offers a complementary view to the convergence rates in Section 9.2, since it is instant and not an upper bound.

We present an analytical form of the loss reduction $\frac{d}{dt}L(W_t, B_t)$ for linear GNNs and multiscale linear GNNs. The comparison of training speed then follows from our formula for $\frac{d}{dt}L(W_t, B_t)$. For better exposition, we first introduce several notations. We let $\bar{B}^{(l':l)} = B_{(l)}B_{(l-1)}\cdots B_{(l')}$ for all l' and l where $\bar{B}^{(l':l)} = I$ if $l' > l$. We also define

$$J_{(i,l),t} := [\bar{B}_t^{(1:i-1)} \otimes (W_{(l),t}\bar{B}_t^{(i+1:l)})^\top],$$

$$F_{(l),t} := [(\bar{B}_t^{(1:l)})^\top \bar{B}_t^{(1:l)} \otimes I_{m_y}] \succeq 0,$$

$$V_t := \frac{\partial L(W_t, B_t)}{\partial \hat{Y}_t},$$

where $\hat{Y}_t := f(X, W_t, B_t)_{*\mathcal{I}}$. For any vector $v \in \mathbb{R}^m$ and positive semidefinite matrix

$M \in \mathbb{R}^{m\times m}$, we use $\|v\|_M^2 := v^\top Mv$.[1] Intuitively, V_t represents the derivative of the loss $L(W_t, B_t)$ with respect to the model output $\hat{Y} = f(X, W_t, B_t)_{*\mathcal{I}}$. $J_{(i,l),t}$ and $F_{(l),t}$ represent matrices that describe how the errors are propagated through the weights of the networks.

Theorem 9.6, proved in Appendix F.1.6, gives an analytical formula of loss reduction for linear GNNs and multiscale linear GNNs.

Theorem 9.6. *For any differentiable loss function $q \mapsto \ell(q, Y)$, the following hold for any $H \geq 0$ and $t \geq 0$:*

(i) (Non-multiscale) *For f as in Definition 9.1:*

$$\frac{d}{dt}L_1(W_t, B_t) = -\left\|\operatorname{vec}\left[V_t(X(S^H)_{*\mathcal{I}})^\top\right]\right\|^2_{F_{(H),t}} - \sum_{i=1}^{H}\left\|J_{(i,H),t}\operatorname{vec}\left[V_t(X(S^H)_{*\mathcal{I}})^\top\right]\right\|_2^2. \tag{9.15}$$

(ii) (Multiscale) *For f as in Definition 9.2:*

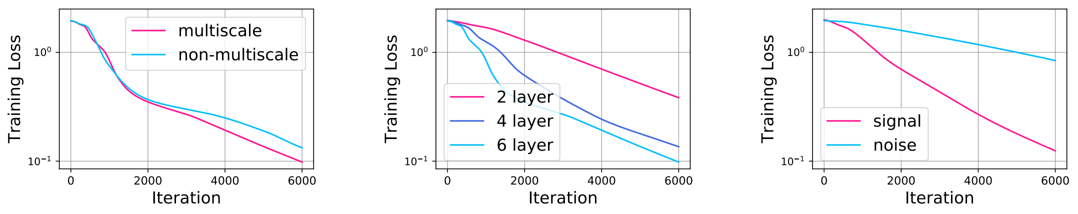
$$\frac{d}{dt}L_2(W_t, B_t) = -\sum_{l=0}^{H}\left\|\operatorname{vec}\left[V_t(X(S^l)_{*\mathcal{I}})^\top\right]\right\|^2_{F_{(l),t}} - \sum_{i=1}^{H}\left\|\sum_{l=i}^{H}J_{(i,l),t}\operatorname{vec}\left[V_t(X(S^l)_{*\mathcal{I}})^\top\right]\right\|_2^2. \tag{9.16}$$

In what follows, we apply Theorem 9.6 to predict how different factors affect the training speed of GNNs.

9.3.1 Acceleration with Skip Connections

We first show that multiscale linear GNNs tend to achieve faster loss reduction $\frac{d}{dt}L_2(W_t, B_t)$ compared to the corresponding linear GNN without skip connections, $\frac{d}{dt}L_1(W_t, B_t)$. It

[1]We use this Mahalanobis norm notation for conciseness without assuming it to be a norm, since M may be low rank.



(a) Multiscale. (b) Depth. (c) Signal vs. noise.

Figure 9-4: **Comparison of the training speed of GNNs.** Left: Multiscale GNNs train faster than non-multiscale GNNs. Middle: Deeper GNNs train faster. Right: GNNs train faster when the labels have signals instead of random noise. The patterns above hold for both ReLU and linear GNNs. Additional results are in Appendix F.2.

follows from Theorem 9.6 that

$$\begin{aligned}&\frac{d}{dt}L_2(W_t, B_t) - \frac{d}{dt}L_1(W_t, B_t) \\ &\leq - \sum_{l=0}^{H-1} \left\| \mathrm{vec}\left[V_t(X(S^l)_{*\mathcal{I}})^\top \right] \right\|_{F_{(l),t}}^2 ,\end{aligned} \tag{9.17}$$

if $\sum_{i=1}^{H}(\|a_i\|_2^2 + 2b_i^\top a_i) \geq 0$, where $a_i = \sum_{l=i}^{H-1} J_{(i,l),t} \,\mathrm{vec}[V_t(X(S^l)_{*\mathcal{I}})^\top]$, and $b_i = J_{(i,H),t}\,\mathrm{vec}[V_t(X(S^H)_{*\mathcal{I}})^\top]$. The assumption of $\sum_{i=1}^{H}(\|a_i\|_2^2 + 2b_i^\top a_i) \geq 0$ is satisfied in various ways: for example, it is satisfied if the last layer's term b_i and the other layers' terms a_i are aligned as $b_i^\top a_i \geq 0$, or if the last layer's term b_i is dominated by the other layers' terms a_i as $2\|b_i\|_2 \leq \|a_i\|_2$. Then equation (9.17) shows that the multiscale linear GNN decreases the loss value with strictly many more negative terms, suggesting faster training.

Empirically, we indeed observe that multiscale GNNs train faster (Figure 9-4a), both for (nonlinear) ReLU and linear GNNs. We verify this by training multiscale and non-multiscale, ReLU and linear GCNs on the Cora and Citeseer datasets with cross-entropy loss, learning rate 5e-5, and hidden dimension 32. Results are in Appendix F.2.

9.3.2 Acceleration with Depth

Our second finding is that deeper GNNs, with or without skip connections, train faster. For any differentiable loss function $q \mapsto \ell(q, Y)$, Theorem 9.6 states that the loss of the

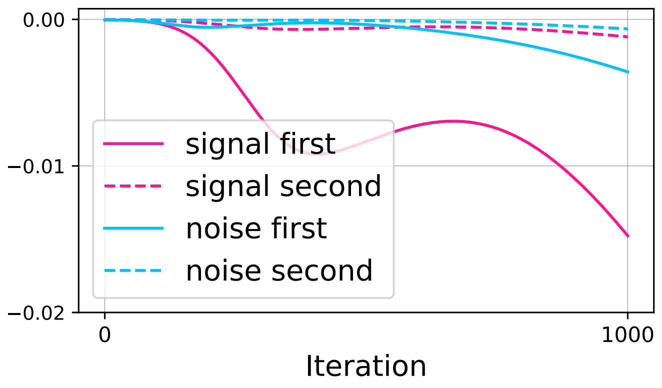


Figure 9-5: **The scale of the first term dominates the second term** of the loss reduction $\frac{d}{dt}L(W_t, B_t)$ for linear GNNs trained with the original labels vs. random labels on Cora.

multiscale linear GNN decreases as

$$\frac{d}{dt}L(W_t, B_t) = \underbrace{-\sum_{l=0}^{H} \underbrace{\left\| \text{vec}\left[V_t(X(S^l)_{*\mathcal{I}})^\top\right] \right\|_{F_{(l),t}}^2}_{\geq 0}}_{\text{further improvement as depth } H \text{ increases}} \underbrace{-\sum_{i=1}^{H} \underbrace{\left\| \sum_{l=i}^{H} J_{(i,l),t} \, \text{vec}\left[V_t(X(S^l)_{*\mathcal{I}})^\top\right] \right\|_2^2}_{\geq 0}}_{\text{further improvement as depth } H \text{ increases}} . \tag{9.18}$$

In equation (9.18), we can see that the multiscale linear GNN achieves faster loss reduction as depth H increases. A similar argument applies to non-multiscale linear GNNs.

Empirically too, deeper GNNs train faster (Figure 9-4b). Again, the acceleration applies to both (nonlinear) ReLU GNNs and linear GNNs. We verify this by training multiscale and non-multiscale, ReLU and linear GCNs with 2, 4, and 6 layers on the Cora and Citeseer datasets with learning rate 5e-5, hidden dimension 32, and cross-entropy loss. Results are in Appendix F.2.

9.3.3 Label Distribution: Signal vs. Noise

Finally, we study how the labels affect the training speed. For the loss reduction (9.15) and (9.16), we argue that the norm of $V_t(X(S^l)_{*\mathcal{I}})^\top$ tends to be larger for labels Y that are more correlated with the graph features $X(S^l)_{*\mathcal{I}}$, e.g., labels are signals instead of "noise".

Without loss of generality, we assume Y is normalized, e.g., one-hot labels. Here, $V_t = \frac{\partial L(A_t, B_t)}{\partial \hat{Y}_t}$ is the derivative of the loss with respect to the model output, e.g., $V_t = 2(\hat{Y}_t - Y)$ for squared loss. If the rows of Y are random noise vectors, then so are the rows of V_t, and they are expected to get more orthogonal to the columns of $(X(S^l)_{*\mathcal{I}})^\top$ as n increases. In contrast, if the labels Y are highly correlated with the graph features $(X(S^l)_{*\mathcal{I}})^\top$, i.e., the labels have signal, then the norm of $V_t(X(S^l)_{*\mathcal{I}})^\top$ will be larger, implying faster training.

Our argument above focuses on the first term of the loss reduction, $\|V_t(X(S^l)_{*\mathcal{I}})^\top\|_{\mathrm{F}}^2$. We empirically demonstrate that the scale of the second term, $\left\|\sum_{l=i}^{H} J_{(i,l),t} \operatorname{vec}\left[V_t(X(S^l)_{*\mathcal{I}})^\top\right]\right\|_2^2$, is dominated by that of the first term (Figure 9-5). Thus, we can expect GNNs to train faster with signals than noise.

We train GNNs with the original labels of the dataset and random labels (i.e., selecting a class with uniform probability), respectively. The prediction of our theoretical analysis aligns with practice: training is much slower for random labels (Figure 9-4c). We verify this for mutliscale and non-multiscale, ReLU and linear GCNs on the Cora and Citseer datasets with learning rate 1e-4, hidden dimension 32, and cross-entropy loss. Results are in Appendix F.2.

9.4 Related Work

Theoretical analysis of linearized networks. The theoretical study of neural networks with some linearized components has recently drawn much attention. Tremendous efforts have been made to understand linear *feedforward* networks, in terms of their loss landscape [Kawaguchi, 2016, Hardt and Ma, 2017, Laurent and Brecht, 2018] and optimization dynamics [Saxe et al., 2014, Arora et al., 2019a, Bartlett et al., 2019, Du and Hu, 2019, Zou et al., 2020]. Recent works prove global convergence rates for deep linear networks under certain conditions [Bartlett et al., 2019, Du and Hu, 2019, Arora et al., 2019a, Zou et al., 2020]. For example, Arora et al. [2019a] assume the data to be whitened. Zou et al. [2020] fix the weights of certain layers during training. Our work is inspired by these works but differs in that our analysis applies to all learnable weights and does not require these specific assumptions, and we study the more complex GNN architecture with skip connections. GNNs consider the interaction of graph structures via the recursive message passing, but such structured, locally varying interaction is not present in feedforward networks. Furthermore, linear feedforward networks, even with skip connections, have the same expressive power as shallow linear models, a crucial condition in previous proofs [Bartlett et al., 2019, Du and Hu, 2019, Arora et al., 2019a, Zou et al., 2020]. In contrast, the expressive power of multiscale linear GNNs can change significantly as depth increases. Accordingly, our proofs significantly differ from previous studies.

Another line of works studies the gradient dynamics of neural networks in the neural tangent kernel (NTK) regime [Jacot et al., 2018, Li and Liang, 2018, Allen-Zhu et al., 2019b, Arora et al., 2019b, Chizat et al., 2019, Du et al., 2019a,c, Nitanda and Suzuki, 2021]. With over-parameterization, the NTK remains almost constant during training. Hence, the corresponding neural network is implicitly linearized with respect to random features of the NTK at initialization [Lee et al., 2019, Yehudai and Shamir, 2019, Liu et al., 2020]. On the other hand, our work needs to address nonlinear dynamics and changing expressive power.

Learning dynamics and optimization of GNNs. Closely related to our work, Du et al. [2019b], Xu et al. [2021b] from previous chapters study the gradient dynamics of GNNs via the Graph NTK but focus on GNNs' generalization and extrapolation properties. We instead analyze optimization. Only Zhang et al. [2020b] also prove global convergence for GNNs, but for the one-hidden-layer case, and they assume a specialized tensor initialization and training algorithms. In contrast, our results work for any finite depth with no assumptions on specialized training. Other works aim to accelerate and stabilize the training of GNNs through normalization techniques [Cai et al., 2021] and importance sampling [Chen et al., 2018b,a, Huang et al., 2018, Chiang et al., 2019, Zou et al., 2019]. Our work complements these practical works with a better theoretical understanding of GNN training.

9.5 Discussion

This chapter studies the training properties of GNNs through the lens of optimization dynamics. For linearized GNNs with or without skip connections, despite the non-convex objective, we show that gradient descent training is guaranteed to converge to a global minimum at a linear rate. The conditions for global convergence are validated on real-world graphs. We further find out that skip connections, more depth, and/or a good label distribution implicitly accelerate the training of GNNs. Our results suggest deeper GNNs with skip connections may be promising in practice, and serve as a first foundational step for understanding the optimization of general GNNs.

Chapter 10

Accelerating Training with GraphNorm

In the last chapter, we have studied the convergence rates of GNNs. In practice, the training of GNNs is often unstable and the convergence is slow. In this chapter, we study how to practically improve the training of GNNs via normalization.

10.1 Introduction

Normalization methods shift and scale the hidden representations and are shown to help the optimization for deep neural networks [Ioffe and Szegedy, 2015, Ulyanov et al., 2016, Ba et al., 2016, Salimans and Kingma, 2016, Xiong et al., 2020, Salimans et al., 2016, Miyato et al., 2018, Wu and He, 2018, Santurkar et al., 2018]. Curiously, no single normalization helps in every domain, and different architectures require specialized methods. For example, Batch normalization (BatchNorm) is a standard component in computer vision [Ioffe and Szegedy, 2015]; Layer normalization (LayerNorm) is popular in natural language processing [Ba et al., 2016, Xiong et al., 2020]; Instance normalization (InstanceNorm) has been found effective for style transfer tasks [Ulyanov et al., 2016] . This motivates the question: *What normalization methods are effective for GNNs?*

We take an initial step towards answering the question above. First, we adapt the existing methods from other domains, including BatchNorm, LayerNorm, and InstanceNorm, to GNNs and evaluate their performance with extensive experiments on graph classification tasks. We observe that our adaptation of InstanceNorm to GNNs, which for each *individ-*

ual graph normalizes its node hidden representations, obtains much faster convergence compared to BatchNorm and LayerNorm. We provide an explanation for the success of InstanceNorm by showing that the shift operation in InstanceNorm serves as a preconditioner of the graph aggregation operation. Empirically, such preconditioning makes the optimization curvature smoother and makes the training more efficient. We also explain why the widely used BatchNorm does not bring the same level of acceleration. The variance of the batch-level statistics on graph datasets is much larger if we apply the normalization across graphs in a batch instead of across individual graphs. The noisy statistics during training may lead to unstable optimization.

Second, we show that the adaptation of InstanceNorm to GNNs, while being helpful in general, has limitations. The shift operation in InstanceNorm, which subtracts the mean statistics from node hidden representations, may lead to an expressiveness degradation for GNNs. Specifically, for highly regular graphs, the mean statistics contain graph structural information, and thus removing them could hurt the performance. Based on our analysis, we propose *GraphNorm* to address the issue of InstanceNorm with a learnable shift (Step 2 in Figure 10-1). The learnable shift could learn to control the ideal amount of information to preserve for mean statistics. Together, GraphNorm normalizes the hidden representations across nodes in each individual graph with a learnable shift to avoid the expressiveness degradation while inheriting the acceleration effect of the shift operation.

We validate the effectiveness of GraphNorm on eight popular graph classification benchmarks. Empirical results confirm that GraphNorm consistently improves the speed of converge and stability of training for GNNs compared to those with BatchNorm, InstanceNorm, LayerNorm, and those without normalization. Furthermore, GraphNorm helps GNNs achieve better generalization performance on most benchmarks.

Related work. Closely related to our work, InstanceNorm [Ulyanov et al., 2016] is originally proposed for real-time image generation. Variants of InstanceNorm are also studied in permutation equivalent data processing [Yi et al., 2018b, Sun et al., 2020]. We instead adapt InstanceNorm to GNNs and find it helpful for the training of GNNs. Few works have studied normalization in the GNN literature. Xu et al. [2019] adapts BatchNorm to GIN

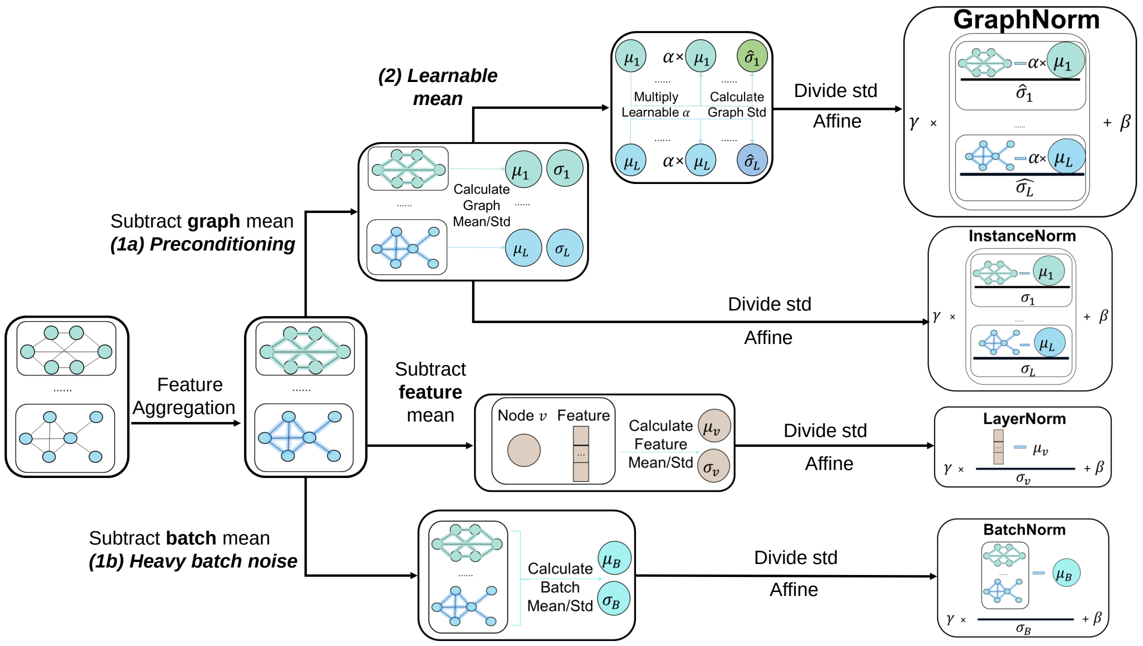


Figure 10-1: **Overview.** We evaluate and understand BatchNorm, LayerNorm, and InstanceNorm, when adapted to GNNs. InstanceNorm trains faster than LayerNorm and BatchNorm on most datasets (Section 10.2.1), as it serves as a preconditioner of the aggregation of GNNs (1a, Section 10.2.2). The preconditioning effect is weaker for BatchNorm due to heavy batch noise in graphs (1b, Section 10.2.3). We propose GraphNorm with a learnable shift to address the limitation of InstanceNorm. GraphNorm outperforms other normalization methods for both training speed (Figure 10-2) and generalization (Table 10.1, 10.2).

as a plug-in component. Our proposed GraphNorm builds on and improves InstanceNorm by addressing its expressiveness degradation with a learnable shift.

The reason behind the effectiveness of normalization has been intensively studied. While scale and shift are the main components of normalization, most existing works focus on the scale operation and the "scale-invariant" property: With a normalization layer after a linear (or convolutional) layer, the output values remain the same as the weights are scaled. Hence, normalization decouples the optimization of direction and length of the parameters [Kohler et al., 2019], implicitly tunes the learning rate [Ioffe and Szegedy, 2015, Hoffer et al., 2018, Arora et al., 2018c, Li and Arora, 2019], and smooths the optimization landscape [Santurkar et al., 2018]. Our work offers a different view by instead showing specific *shift* operation has the preconditioning effect and can accelerate the training of GNNs.

10.2 Evaluating and Understanding Normalization

We begin by introducing our notations for this chapter. For better exposition of our analysis, we denote GNNs in the matrix form. GIN can be defined in matrix form as:

$$H^{(k)} = \mathrm{MLP}^{(k)}\left(W^{(k)}H^{(k-1)}Q_{\mathrm{GIN}}\right), \tag{10.1}$$

where $H^{(k)} = \left[h_1^{(k)}, h_2^{(k)}, \cdots, h_n^{(k)}\right] \in \mathbb{R}^{d^{(k)}\times n}$ is the feature matrix at the k-th layer where $d^{(k)}$ denotes the feature dimension, and $W^{(k)}$ is the parameter matrix in layer k, $\xi^{(k)}$ is a learnable parameter and $Q_{\mathrm{GIN}} = A + I_n + \xi^{(k)} I_n$. GCN can be defined in matrix form as:

$$H^{(k)} = \mathrm{ReLU}\left(W^{(k)}H^{(k-1)}Q_{\mathrm{GCN}}\right), \tag{10.2}$$

where ReLU stands for rectified linear unit, $Q_{\mathrm{GCN}} = \widehat{D}^{-\frac{1}{2}}\widehat{A}\widehat{D}^{-\frac{1}{2}}$, where $\widehat{A} = A + I_n$ and $\widehat{D}$ is the degree matrix of $\widehat{A}$. I_n is the identity matrix.

Normalization. Generally, given a set of values $\{x_1, x_2, \cdots, x_m\}$, a normalization operation first shifts each x_i by the mean μ, and then scales them down by standard deviation σ: $x_i \to \gamma\frac{x_i-\mu}{\sigma} + \beta$, where γ and β are learnable parameters, $\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$ and

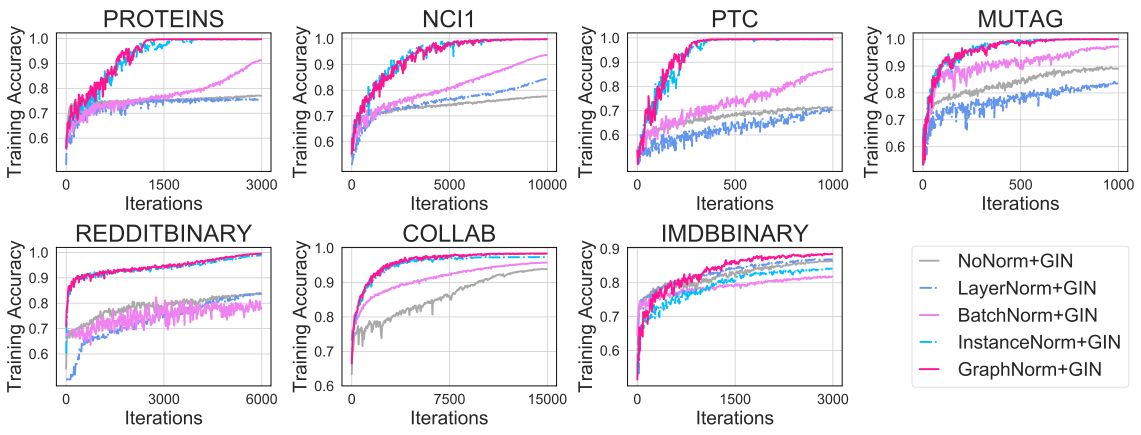


Figure 10-2: **Training performance** of GIN with different normalization methods and GIN without normalization in graph classification tasks. The convergence speed of our adaptation of InstanceNorm dominates BatchNorm and LayerNorm in most tasks. GraphNorm further improves the training over InstanceNorm especially on tasks with highly regular graphs, e.g., IMDB-BINARY (See Figure 10-5 for detailed illustration). Overall, GraphNorm converges faster than all other methods.

$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2$. The major difference among different existing normalization methods is which set of feature values the normalization is applied to. For example, in computer vision, BatchNorm normalizes the feature values in the same channel across different samples in a batch. In NLP, LayerNorm normalizes the feature values at each position in a sequence separately.

We first adapt and evaluate existing normalization methods to GNNs. Then we give an explanation of the effectiveness of the variant of InstanceNorm, and show why the widely used BatchNorm fails to have such effectiveness. The understanding inspires us to develop better normalization methods, e.g., GraphNorm.

10.2.1 Adapting and Evaluating Normalization for GNNs

To investigate what normalization methods are effective for GNNs, we first adapt three typical normalization methods, i.e., BatchNorm, LayerNorm, and InstanceNorm, developed in other domain to GNNs. We apply the normalization after the linear transformation as in previous works [Ioffe and Szegedy, 2015, Xiong et al., 2020, Xu et al., 2019]. The general

GNN structure equipped with a normalization layer can be represented as:

$$H^{(k)} = F^{(k)}\left(\mathrm{Norm}\left(W^{(k)}H^{(k-1)}Q\right)\right), \tag{10.3}$$

where $F^{(k)}$ is a function that applies to each node separately, Q is an $n \times n$ matrix representing the neighbor aggregation, and $W^{(k)}$ is the weight/parameter matrix in layer k. We can instantiate Eq. (10.3) as GCN and GIN, by setting proper $F^{(k)}$ and matrix Q. For example, if we set $F^{(k)}$ to be ReLU and set Q to be Q_{GCN} (Eq. (10.2)), then Eq. (10.3) becomes GCN with normalization; Similarly, by setting $F^{(k)}$ to be $\mathrm{MLP}^{(k)}$ and Q to be Q_{GIN} (Eq. (10.1)), we recover GIN with normalization.

We then describe the concrete operations of the adaptations of the normalization methods. Consider a batch of graphs $\{G_1, \cdots, G_b\}$ where b is the batch size. Let n_g be the number of nodes in graph G_g. We generally denote $\widehat{h}_{i,j,g}$ as the inputs to the normalization module, e.g., the j-th feature value of node v_i of graph G_g, $i = 1, \cdots, n_g, j = 1, \cdots, d, g = 1, \cdots, b$. The adaptations take the general form:

$$\mathrm{Norm}\left(\widehat{h}_{i,j,g}\right) = \gamma \cdot \frac{\widehat{h}_{i,j,g} - \mu}{\sigma} + \beta, \tag{10.4}$$

where the scopes of mean μ, standard deviation σ, and affine parameters γ, β differ for different normalization methods. For BatchNorm, normalization and the computation of μ and σ are applied to all values in the same feature dimension across the nodes of *all graphs in the batch* as in Xu et al. [2019], i.e., over dimensions g, i of $\widehat{h}_{i,j,g}$. To adapt LayerNorm to GNNs, we view each node as a basic component, resembling words in a sentence, and apply normalization to all feature values across different dimensions of each node, i.e., over dimension j of $\widehat{h}_{i,j,g}$.

For InstanceNorm, we regard each graph as an instance. The normalization is then applied to the feature values across all nodes for each *individual graph*, i.e., over dimension i of $\widehat{h}_{i,j,g}$.

In Figure 10-2 we show training curves of different normalization methods in graph classification tasks. We find that LayerNorm hardly improves the training process in most tasks, while our adaptation of InstanceNorm can largely boost the training speed compared to other normalization methods. The test performances have similar trends. We summarize the final test accuracies in Table 10.1. In the following subsections, we provide an explanation for the success of InstanceNorm and its benefits compared to BatchNorm, which is currently adapted in many GNNs.

10.2.2 Shift in InstanceNorm as a Preconditioner

As mentioned previously, the scale-invariant property of the normalization has been investigated and considered as one of the ingredients that make the optimization efficient. In our analysis of normalizations for GNNs, we instead take a closer look at the *shift* operation in the normalization. Compared to the image and sequential data, the graph is explicitly structured, and the neural networks exploit the structural information directly in the aggregation of the neighbors. Such uniqueness of GNNs makes it possible to study how the shift operation interplays with the graph data in detail.

We show that the shift operation in our adaptation of InstanceNorm serves as a preconditioner of the aggregation in GNNs and hypothesize this preconditioning effect can boost the training of GNNs. Though the current theory of deep learning has not been able to prove and compare the convergence rate in the real settings, we calculate the convergence rate of GNNs on a simple but fully characterizable setting to give insights on the benefit of the shift operation.

We first formulate our adaptation of InstanceNorm in the matrix form. Mathematically, for a graph of n nodes, denote $N = I_n - \frac{1}{n}\mathbf{1}\mathbf{1}^\top$. N is the matrix form of the shift operation, i.e., for any vector $\mathbf{z} = [z_1, z_2, \cdots, z_n]^\top \in \mathbb{R}^n$, $\mathbf{z}^\top N = \mathbf{z}^\top - \left(\frac{1}{n}\sum_{i=1}^n z_i\right)\mathbf{1}^\top$. Then the normalization together with the aggregation can be represented as[1]

$$\mathrm{Norm}\left(W^{(k)}H^{(k-1)}Q\right) = S\left(W^{(k)}H^{(k-1)}Q\right)N, \tag{10.5}$$

where $S = \mathrm{diag}\left(\frac{1}{\sigma_1}, \frac{1}{\sigma_2}, \cdots, \frac{1}{\sigma_{d^{(k)}}}\right)$ is the scaling, and Q is the GNN aggregation matrix.

[1]Standard normalization has an additional affine operation after shifting and scaling. Here we omit it in Eq. 10.5 for better demonstration. Adding this operation will not affect the theoretical analysis.

Each σ_i is the standard deviation of the values of the i-th features among the nodes in the graph we consider. We can see that, in the matrix form, shifting feature values on a single graph is equivalent to multiplying N as in Eq. (10.5). Therefore, we further check how this operation affects optimization. In particular, we examine the singular value distribution of QN. The following theorem shows that QN has a smoother singular value distribution than Q, i.e., N serves as a preconditioner of Q.

Theorem 10.1 (Shift Serves as a Preconditioner of Q)**.** *Let Q, N be defined as in Eq. (10.5), $0 \leq \lambda_1 \leq \cdots \leq \lambda_n$ be the singular values of Q. We have $\mu_n = 0$ is one of the singular values of QN, and let other singular values of QN be $0 \leq \mu_1 \leq \mu_2 \leq \cdots \leq \mu_{n-1}$. Then we have*

$$\lambda_1 \leq \mu_1 \leq \lambda_2 \leq \cdots \leq \lambda_{n-1} \leq \mu_{n-1} \leq \lambda_n, \tag{10.6}$$

where $\lambda_i = \mu_i$ or $\lambda_i = \mu_{i-1}$ only if there exists one of the right singular vectors α_i of Q associated with λ_i satisfying $\mathbf{1}^\top \alpha_i = 0$.

The proof may be found in Appendix G.1.1.

We hypothesize that precoditioning Q can help the optimization. In the case of optimizing the weight matrix $W^{(k)}$, we can see from Eq. (10.5) that after applying normalization, the term Q in the gradient of $W^{(k)}$ will become QN which makes the optimization curvature of $W^{(k)}$ smoother, see Appendix G.1.5 for more discussions. Similar preconditioning effects are believed to improve the training of deep learning models [Duchi et al., 2011, Kingma and Ba, 2015], and classic wisdom in optimization has also shown that preconditioning can accelerate the convergence of iterative methods [Axelsson, 1985, Demmel, 1997]. To support our hypothesis that preconditioning may suggest better training, we investigate a simple but characterizable setting of training a linear GNN using gradient descent in Appendix G.1.2. In this setting, we prove that:

Proposition 10.2 (Concrete Example Showing Shift can Accelerate Training (Informal))**.** *With high probability over randomness of data generation, the parameter $\mathbf{w}_t^{\text{Shift}}$ of the model*

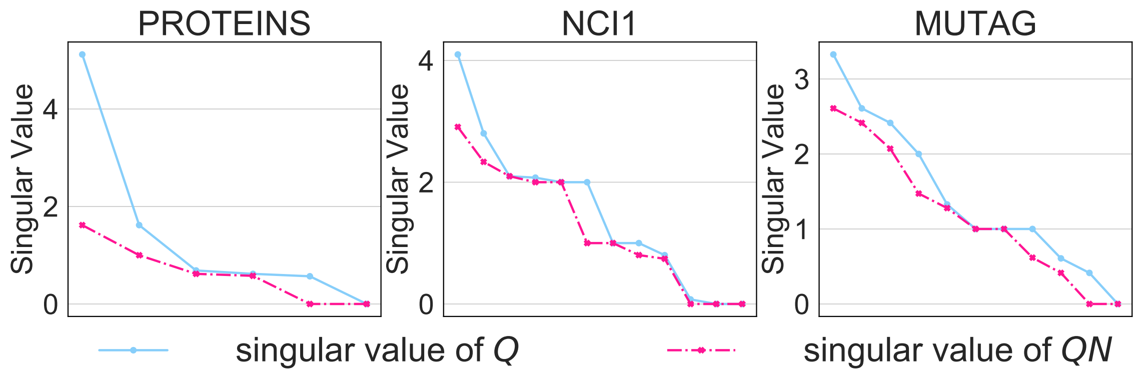


Figure 10-3: **Singular value distribution** of Q and QN for sampled graphs in different datasets using GIN. More visualizations can be found in Appendix G.3.1

with shift at step t converges to the optimal parameter $\mathbf{w}_*^{\text{Shift}}$ linearly:

$$\left\|\mathbf{w}_t^{\text{Shift}} - \mathbf{w}_*^{\text{Shift}}\right\|_2 = O\left(\rho_1^t\right),$$

where ρ_1 is the convergence rate.

Similarly, the parameter $\mathbf{w}_t^{\text{Vanilla}}$ of the vanilla model converges linearly, but with a slower rate:

$$\left\|\mathbf{w}_t^{\text{Vanilla}} - \mathbf{w}_*^{\text{Vanilla}}\right\|_2 = O\left(\rho_2^t\right) \text{ and } \rho_1 < \rho_2,$$

which indicates that the model with shift converges faster than the vanilla model.

The proof may be found in Appendix G.1.2.

To check how much the matrix N improves the distribution of the spectrum of matrix Q in real practice, we sample graphs from different datasets for illustration, as showed in Figure 10-3 (more visualizations for different types of graph can be found in Appendix G.3.1). We can see that the singular value distribution of QN is much smoother, and the condition number is improved. Note that for a multi-layer GNN, the normalization will be applied in each layer. Therefore, the overall improvement of such preconditioning can be more significant.

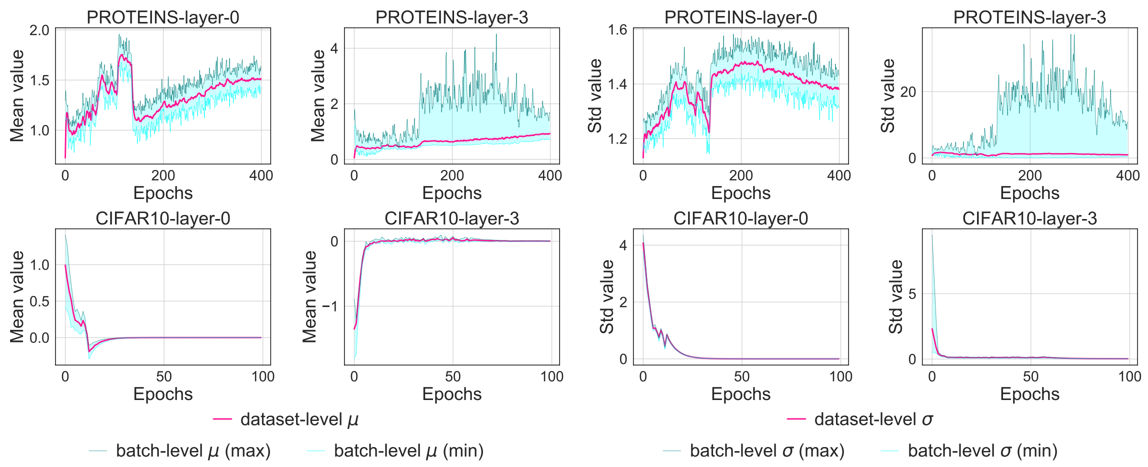


Figure 10-4: **Batch-level statistics are noisy for GNNs.** We plot the batch-level/dataset-level mean/standard deviation of models trained on PROTEINS (graph classification) and CIFAR10 (image classification). We observe that the deviation of batch-level statistics from dataset-level statistics is rather large for the graph task, while being negligible in image task.

10.2.3 Heavy Batch Noise in Graphs Makes BatchNorm Less Effective

The above analysis shows the adaptation of InstanceNorm has the effect of preconditioning the aggregation of GNNs. Then a natural question is whether a batch-level normalization for GNNs [Xu et al., 2019] has similar advantages. We show that BatchNorm is less effective in GNNs due to heavy batch noise on graph data.

In BatchNorm, the mean μ_B and standard deviation σ_B are calculated in a sampled batch during training, which can be viewed as random variables by the randomness of sampling. During testing, the estimated dataset-level statistics (running mean μ_D and standard deviation σ_D) are used instead of the batch-level statistics [Ioffe and Szegedy, 2015]. To apply Theorem 10.1 to BatchNorm for the preconditioning effect, one could potentially view all graphs in a dataset as subgraphs in a *super graph*. Hence, Theorem 10.1 applies to BatchNorm if the batch-level statistics are well-concentrated around dataset-level statistics, i.e., $\mu_B \approx \mu_D$ and $\sigma_B \approx \sigma_D$. However, the concentration of batch-level statistics is heavily *domain-specific*. While it is imaginable that the variation of batch-level statistics in typical networks is small for computer vision, the concentration of batch-level statistics is still unknown for GNNs.

We study how the batch-level statistics μ_B, σ_B deviate from the dataset-level statistics

μ_D, σ_D. For comparison, we train a 5-layer GIN with BatchNorm on the PROTEINS dataset and train a ResNet18 [He et al., 2016] on the CIFAR10 dataset. We set batch size to 128. For each epoch, we record the batch-level max/min mean and standard deviation for the first and the last BatchNorm layer on a randomly selected dimension across batches. In Figure 10-4, pink line denotes the dataset-level statistics, and green/blue line denotes the max/min value of the batch-level statistics. We observe that for image tasks, the maximal deviation of the batch-level statistics from the dataset-level statistics is negligible (Figure 10-4) after a few epochs. In contrast, for the graph tasks, the variation of batch-level statistics stays large during training. Intuitively, the graph structure can be quite diverse and the a single batch cannot well represent the entire dataset. Hence, the preconditioning property also may not hold for BatchNorm. In fact, the heavy batch noise may bring instabilities to the training. More results may be found in Appendix G.3.2.

10.3 Graph Normalization

Although we provide evidence on the indispensability and advantages of our adaptation of InstanceNorm, simply normalizing the values in each feature dimension within a graph does not consistently lead to improvement. We show that in some situations, e.g., for regular graphs, the standard shift (e.g., shifting by subtracting the mean) may cause information loss on graph structures.

We consider r-regular graphs, i.e., each node has a degree r. We first look into the case that there are no available node features, then X_i is set to be the one-hot encoding of the node degree [Xu et al., 2019]. In a r-regular graph, all nodes have the same encoding, and thus the columns of $H^{(0)}$ are the same. We study the output of the standard shift operation in the first layer, i.e., $k = 1$ in Eq. (10.5). From the following proposition, we can see that when the standard shift operation is applied to GIN for a r-regular graph described above, the information of degree is lost:

Proposition 10.3. *For a r-regular graph with features described above, we have for GIN, $\mathrm{Norm}\left(W^{(1)}H^{(0)}Q_{\mathrm{GIN}}\right) = S\left(W^{(1)}H^{(0)}Q_{\mathrm{GIN}}\right)N = 0$, i.e., the output of normalization layer is a zero matrix without any information of the graph structure.*

Such information loss not only happens when there are no node features. For complete graphs, we can further show that even each node has different features, the graph structural information, i.e., adjacency matrix A, will always be ignored after the standard shift operation in GIN:

Proposition 10.4. *For a complete graph ($r = n - 1$), we have for GIN, $Q_{\text{GIN}} N = \xi^{(k)} N$, i.e., graph structural information in Q will be removed after multiplying N.*

The proof of these two propositions can be found in Appendix G.1. Similar results can be easily derived for other architectures like GCN by substituting Q_{GIN} with Q_{GCN}. As we can see from the above analysis, in graph data, the mean statistics after the aggregation sometimes contain structural information. Discarding the mean will degrade the expressiveness of the neural networks. Note that the problem may not happen in image domain. The mean statistics of image data contains global information such as brightness. Removing such information in images will not change the semantics of the objects and thus will not hurt the classification performance.

This analysis inspires us to modify the current normalization method with a *learnable parameter* to automatically control how much the mean to preserve in the shift operation. Combined with the graph-wise normalization, we name our new method Graph Normalization, i.e., GraphNorm. For each graph G, we generally denote value $\widehat{h}_{i,j}$ as the inputs to GraphNorm, e.g., the j-th feature value of node v_i, $i = 1, \cdots, n$, $j = 1, \cdots, d$. GraphNorm takes the following form:

$$\text{GraphNorm}\left(\widehat{h}_{i,j}\right) = \gamma_j \cdot \frac{\widehat{h}_{i,j} - \alpha_j \cdot \mu_j}{\hat{\sigma}_j} + \beta_j, \tag{10.7}$$

where $\mu_j = \frac{\sum_{i=1}^n \widehat{h}_{i,j}}{n}$, $\hat{\sigma}_j^2 = \frac{\sum_{i=1}^n \left(\widehat{h}_{i,j} - \alpha_j \cdot \mu_j\right)^2}{n}$, and γ_j, β_j are the affine parameters as in other normalization methods. By introducing the learnable parameter α_j for each feature dimension j, we are able to learn how much the information we need to keep in the mean.

To validate our theory and the proposed GraphNorm in real-world data, we conduct an ablation study on two typical datasets, PROTEINS and IMDB-BINARY. As shown in Figure 10-5, the graphs from PROTEINS and IMDB-BINARY exhibit irregular-type and

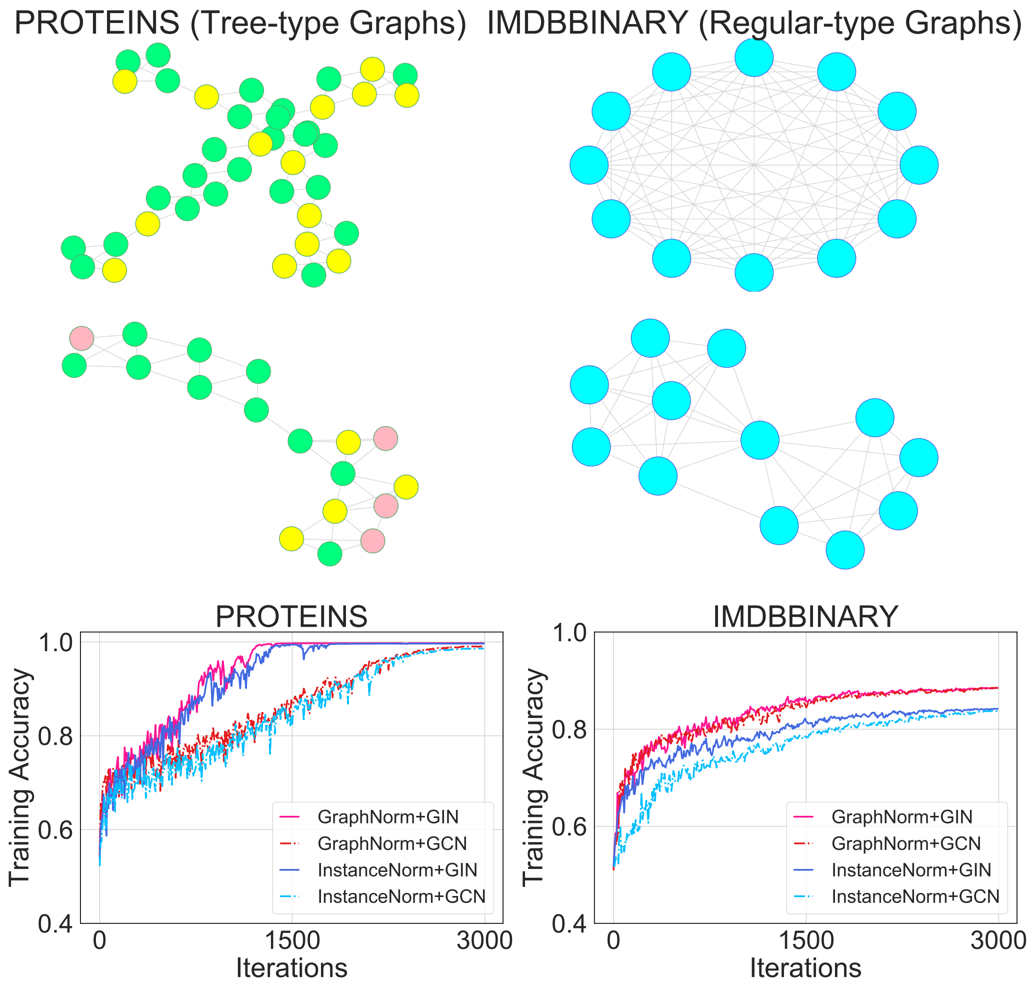


Figure 10-5: **Comparison of GraphNorm and InstanceNorm on different types of graphs.** Top: Sampled graphs with different topological structures. Bottom: Training curves of GIN/GCN using GraphNorm and InstanceNorm.

Table 10.1: **Test performance** of GIN/GCN with various normalization methods on graph classification tasks.

Datasets	MUTAG	PTC	PROTEINS	NCI1	IMDB-B	RDT-B	COLLAB
# graphs	188	344	1113	4110	1000	2000	5000
# classes	2	2	2	2	2	2	2
Avg # nodes	17.9	25.5	39.1	29.8	19.8	429.6	74.5
WL subtree [Shervashidze et al., 2011]	90.4 ± 5.7	59.9 ± 4.3	75.0 ± 3.1	$\mathbf{86.0 \pm 1.8}$	73.8 ± 3.9	81.0 ± 3.1	78.9 ± 1.9
DCNN [Atwood and Towsley, 2016]	67.0	56.6	61.3	62.6	49.1	-	52.1
DGCNN [Zhang et al., 2018a]	85.8	58.6	75.5	74.4	70.0	-	73.7
AWL [Ivanov and Burnaev, 2018]	87.9 ± 9.8	-	-	-	74.5 ± 5.9	87.9 ± 2.5	73.9 ± 1.9
GIN+LayerNorm	82.4 ± 6.4	62.8 ± 9.3	76.2 ± 3.0	$78.3 \pm 1,7$	$74.5 \pm 4,4$	82.8 ± 7.7	80.1 ± 0.8
GIN+BatchNorm ([Xu et al., 2019])	89.4 ± 5.6	64.6 ± 7.0	76.2 ± 2.8	82.7 ± 1.7	75.1 ± 5.1	92.4 ± 2.5	$\mathbf{80.2 \pm 1.9}$
GIN+InstanceNorm	90.5 ± 7.8	64.7 ± 5.9	76.5 ± 3.9	81.2 ± 1.8	74.8 ± 5.0	93.2 ± 1.7	80.0 ± 2.1
GIN+GraphNorm	$\mathbf{91.6 \pm 6.5}$	$\mathbf{64.9 \pm 7.5}$	$\mathbf{77.4 \pm 4.9}$	81.4 ± 2.4	$\mathbf{76.0 \pm 3.7}$	$\mathbf{93.5 \pm 2.1}$	$\mathbf{80.2 \pm 1.0}$

regular-type graphs, respectively. We train GIN/GCN using our adaptation of InstanceNorm and GraphNorm under the same setting in Section 10.4. The training curves are presented in Figure 10-5. The curves show that using a learnable α slightly improves the convergence on PROTEINS, while significantly boost the training on IMDB-BINARY. This observation verify that shifting the feature values by subtracting the mean may lose information, especially for regular graphs. And the introduction of learnable shift in GraphNorm can effectively mitigate the expressive degradation.

10.4 Experiments

In this section, we evaluate and compare both the training and test performance of GraphNorm with other normalization methods on graph classification benchmarks.

Settings. We use eight popularly used benchmark datasets of different scales in the experiments [Yanardag and Vishwanathan, 2015, Xu et al., 2019], including four medium-scale bioinformatics datasets (MUTAG, PTC, PROTEINS, NCI1), three medium-scale social network datasets (IMDB-BINARY, COLLAB, REDDIT-BINARY), and one large-scale bioinformatics dataset ogbg-molhiv, which is recently released on Open Graph Benchmark (OGB) [Hu et al., 2020a]. Dataset statistics are summarized in Table 10.1. We use two typical graph neural networks GIN [Xu et al., 2019] and GCN [Kipf and Welling, 2017] for our evaluations. Specifically, we use a five-layer GCN/GIN. For GIN, the number of

Table 10.2: **Test performance** on OGB.

Datasets	OGBG-MOLHIV
# graphs	41,127
# classes	2
Avg # nodes	25.5
GCN [Hu et al., 2020a]	76.06 ± 0.97
GIN [Hu et al., 2020a]	75.58 ± 1.40
GCN+LayerNorm	75.04 ± 0.48
GCN+BatchNorm	76.22 ± 0.95
GCN+InstanceNorm	78.18 ± 0.42
GCN+GraphNorm	**78.30** ± **0.69**
GIN+LayerNorm	74.79 ± 0.92
GIN+BatchNorm	76.61 ± 0.97
GIN+InstanceNorm	77.54 ± 1.27
GIN+GraphNorm	**77.73** ± **1.29**

sub-layers in MLP is set to 2. Normalization is applied to each layer. To aggregate global features on top of the network, we use SUM readout for MUTAG, PTC, PROTEINS and NCI1 datasets, and use MEAN readout for other datasets, as in Xu et al. [2019]. Details of the experimental settings are presented in Appendix G.2.

Results. We plot the training curves of GIN with GraphNorm and other normalization methods on different tasks in Figure 10-2. The results on GCN show similar trends, and are provided in Appendix G.3.3. As shown in Figure 10-2, GraphNorm enjoys the fastest convergence on all tasks. Compared to BatchNorm used in Xu et al. [2019], GraphNorm converges in roughly 5000/500 iterations on NCI1 and PTC datasets, while the model using BatchNorm does not even converge in 10000/1000 iterations. Remarkably, though InstanceNorm does *not* outperform other normalization methods on IMDB-BINARY, GraphNorm with learnable shift significantly boosts the training upon InstanceNorm and achieves the fastest convergence. We also validate the test performance and report the test accuracy in Table 10.1,10.2. The results show that GraphNorm also improves the generalization on most benchmarks.

10.4.1 Ablation Study

In this subsection, we summarize the results of some ablation studies. Due to the space limitation, the detailed results can be found in Appendix G.3.

BatchNorm with learnable shift. We conduct experiments on BatchNorm to investigate whether simply introducing a learnable shift can already improve the existing normalization methods without concrete motivation of overcoming expressiveness degradation. Specifically, we equip BatchNorm with a similar learnable shift as GraphNorm and evaluate its performance. We find that the learnable shift cannot further improve upon BatchNorm (See Appendix G.3), which suggests the introduction of learnable shift in GraphNorm is critical.

BatchNorm with running statistics. We study the variant of BatchNorm which uses running statistics to replace the batch-level mean and standard deviation. At first glance, this method may seem to be able to mitigate the problem of large batch noise. However, the running statistics change a lot during training, and using running statistics disables the model to back-propagate the gradients through mean and standard deviation. Results in Appendix G.3 show this variant has even worse performance than BatchNorm.

The effect of batch size. We further compare the GraphNorm with BatchNorm with different batch sizes (8, 16, 32, 64). As shown in Appendix G.3, our GraphNorm consistently outperforms the BatchNorm on all the settings.

10.5 Discussion

In this chapter, we adapt and evaluate three normalization methods, i.e., BatchNorm, LayerNorm, and InstanceNorm to GNNs. We give explanations for the successes and failures of these adaptations. Based on our understanding of the strengths and limitations of existing adaptations, we propose Graph Normalization, which builds upon the adaptation of InstanceNorm with a learnable shift to overcome the expressive degradation of the original InstanceNorm. Experimental results show GNNs with GraphNorm not only converge faster, but also achieve better generalization performance on several benchmark datasets.

Though seeking theoretical understanding of normalization methods in deep learning is challenging [Arora et al., 2018c] due to limited understanding on the optimization of deep learning models and characterization of real world data, we take an initial step towards finding effective normalization methods for GNNs with theoretical guidance in this chapter. The proposed theories and hypotheses are motivated by several simple models. Although we were not able to give concrete theoretical results to problems such as: the convergence rate of general GNNs with normalization, the spectrum of Q normalized by learnable shift, etc, we believe the analyses of more realistic but complicated settings, e.g., the dynamics of GraphNorm on deep GNNs, are good future directions.

Chapter 11

Conclusion

This thesis is about modeling intelligence that can learn to represent and reason about the world. We studied both representation and reasoning from the lens of graph neural networks. In Part I, we introduced theoretical frameworks for characterizing the representation power and developed maximally powerful GNNs. In Part II, we answered what reasoning a neural network can learn by analyzing how the interplay of network structure and task structure affects the generalization. In Part III, we studied how neural networks extrapolate, and showed implications for how neural models can possibly learn the reasoning outside the training distribution. Then in Part IV, we completed the picture of the theoretical landscape by analyzing and improving the optimization of GNNs. From a broader viewpoint, we have presented several theoretical limits of building artificial general intelligence in terms of representation power and generalization, but also have complemented those with promising directions to explore in the future. This concludes the thesis.

Appendix A

Puzzle of the Underperformance of Deeper GNNs

A.1 Proofs

A.1.1 Proof of Theorem 3.1

We proceed the proof by considering the following GCN architecture.

$$h_v^{(l)} = \text{ReLU}\Big(W_l \cdot \frac{1}{\widetilde{\deg}(v)} \sum_{u \in \widetilde{N}(v)} h_u^{(l-1)}\Big)$$

where $\widetilde{\deg}(v)$ is the degree of node v in $\widetilde{G}$, i.e., G with self-loops. It is straightforward to extend our proofs to the following GCN architecture as well.

$$h_v^{(l)} = \text{ReLU}\Big(W_l \cdot \sum_{u \in \widetilde{N}(v)} (\deg(v)\deg(u))^{-1/2}\, h_u^{(l-1)}\Big)$$

Denote by $f_x^{(l)}$ the pre-activated feature of $h_x^{(l)}$, i.e. $\frac{1}{\widetilde{\deg}(x)} \cdot \sum_{z \in \widetilde{N}(x)} W_l h_z^{(l-1)}$, for any $l = 1..k$, we have

$$\frac{\partial h_x^{(l)}}{\partial h_y^{(0)}} = \frac{1}{\widetilde{\deg}(x)} \cdot \text{diag}\left(1_{f_x^{(l)} > 0}\right) \cdot W_l \cdot \sum_{z \in \widetilde{N}(x)} \frac{\partial h_z^{(l-1)}}{\partial h_y^{(0)}}$$

By chain rule, we get

$$\begin{aligned}\frac{\partial h_x^{(k)}}{\partial h_y^{(0)}} &= \sum_{p=1}^{\Psi}\left[\frac{\partial h_x^{(k)}}{\partial h_y^{(0)}}\right]_p \\ &= \sum_{p=1}^{\Psi}\prod_{l=k}^{1}\frac{1}{\widetilde{\deg}(v_p^l)}\cdot \operatorname{diag}\left(1_{f_{v_p^l}^{(l)}>0}\right)\cdot W_l\end{aligned}$$

Here, Ψ is the total number of paths $v_p^k v_p^{k-1}, ..., v_p^1, v_p^0$ of length $k+1$ from node x to node y. For any path p, v_p^k is node x, v_p^0 is node y and for $l = 1..k-1$, $v_p^{l-1} \in \widetilde{N}(v_p^l)$.

As for each path p, the derivative $\left[\frac{\partial h_x^{(k)}}{\partial h_y^{(0)}}\right]_p$ represents a directed acyclic computation graph, where the input neurons are the same as the entries of W_1, and at a layer l. We can express an entry of the derivative as

$$\left[\frac{\partial h_x^{(k)}}{\partial h_y^{(0)}}\right]_p^{(i,j)} = \prod_{l=k}^{1}\frac{1}{\widetilde{\deg}(v_p^l)}\sum_{q=1}^{\Phi} Z_q \prod_{l=k}^{1} w_q^{(l)}$$

Here, Φ is the number of paths q from the input neurons to the output neuron (i,j), in the computation graph of $\left[\frac{\partial h_x^{(k)}}{\partial h_y^{(0)}}\right]_p$. For each layer l, w_q^l is the entry of W_l that is used in the q-th path. Finally, $Z_q \in \{0,1\}$ represents whether the q-th path is active $(Z_q = 1)$ or not $(Z_q = 0)$ as a result of the ReLU activation of the entries of $f_{v_p^l}^{(l)}$'s on the q-th path.

Under the assumption that the Z's are Bernoulli random variables with the same probability of success, for all q, $\Pr(Z_q = 1) = \rho$, we have $\mathbb{E}\left[\left[\frac{\partial h_x^{(k)}}{\partial h_y^{(0)}}\right]_p^{(i,j)}\right] = \rho \cdot \prod_{l=k}^{1}\frac{1}{\widetilde{\deg}(v_p^l)}\cdot w_q^{(l)}$. It follows that $\mathbb{E}\left[\frac{\partial h_x^{(k)}}{\partial h_y^{(0)}}\right] = \rho\cdot\prod_{l=k}^{1} W_l \cdot \left(\sum_{p=1}^{\Psi}\prod_{l=k}^{1}\frac{1}{\widetilde{\deg}(v_p^l)}\right)$. We know that the k-step random walk probability at y can be computed by summing up the probability of all paths of length k from x to y, which is exactly $\sum_{p=1}^{\Psi}\prod_{l=k}^{1}\frac{1}{\widetilde{\deg}(v_p^l)}$. Moreover, the random walk probability starting at x to other nodes sum up to 1. We know that the influence score $I(x,z)$ for any z in expectation is thus the random walk probability of being at z from x at the k-th step, multiplied by a term that is the same for all z. Normalizing the influence scores ends the proof.

Comment: ReLU is not differentiable at 0. For simplicity, we assume the (sub)gradient to be 0 at 0.

A.1.2 Proof of Proposition 3.2

Let $\left[h_x^{(final)}\right]_i$ be the i-th entry of $h_x^{(final)}$, the feature after layer aggregation. For any node y, we have

$$\begin{aligned} I(x,y) &= \sum_i \left\| \frac{\partial \left[h_x^{(final)}\right]_i}{\partial h_y^{(0)}} \right\|_1 \\ &= \sum_i \left\| \frac{\partial \left[h_x^{(j_i)}\right]_i}{\partial h_y^{(0)}} \right\|_1 \end{aligned}$$

where $j_i = \underset{l}{\operatorname{argmax}} \left(\left[h_x^{(l)}\right]_i \right)$. By Theorem 3.1, we have

$$\mathbb{E}\left[I(x,y)\right] = \sum_l c_x^l \cdot z_l \cdot \mathbb{E}\left[I_x(y)^{(l)}\right]$$

where $I_x(y)$ is the l-step random walk probability at y, z_l is a normalization factor and c_x^l is the fraction of entries of $h_x^{(l)}$ being chosen by max-pooling. By Theorem 3.1, $\mathbb{E}\left[I_x(y)^{(l)}\right]$ is equivalent to the l-step random walk probability at y starting at x.

A.2 Visualization Results

We describe the details of the heat maps and present more visualization results. The colors of the nodes in the heat maps correspond to their probability masses of either the influence distribution or random walk distribution as shown in Figure A-1. As we see, the shallower the color is, the smaller the probability mass. We use the same color for probabilities over 0.2 for better visual effects because there are few nodes with influence probability masses over 0.2. Nodes with probability mass less than 0.001 are not shown in the heat maps.

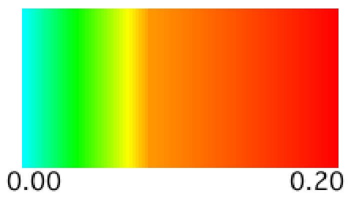


Figure A-1: Color and probability correpondency for the heat maps

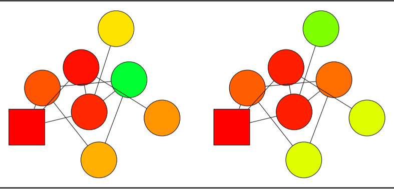
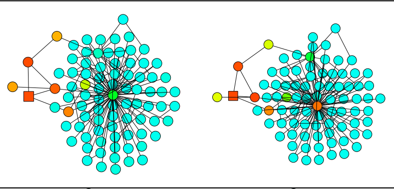
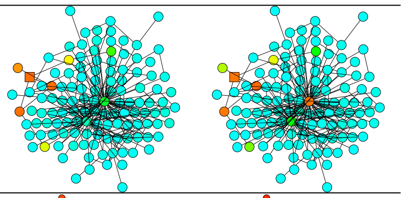
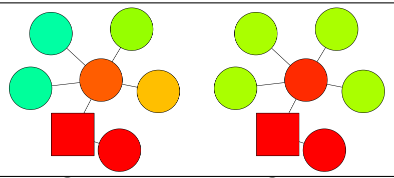
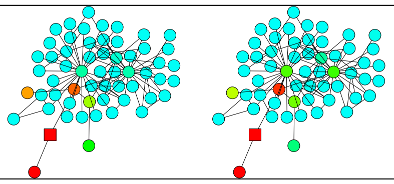
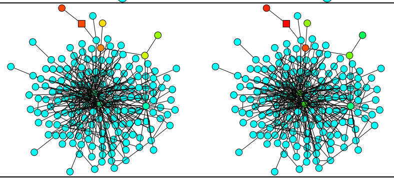
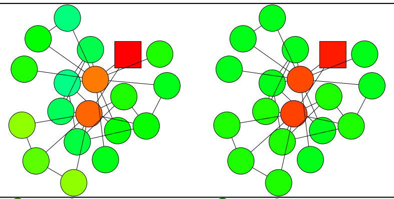
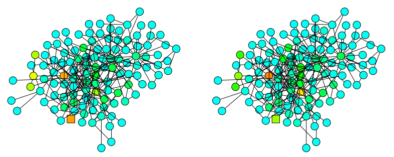
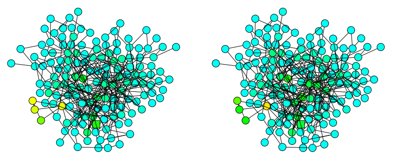
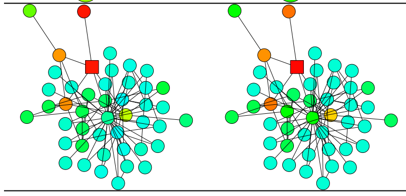
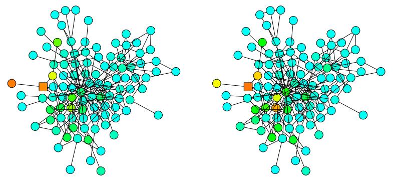
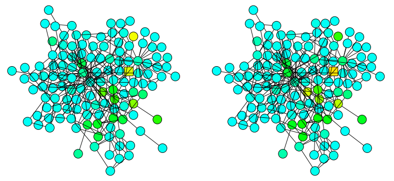
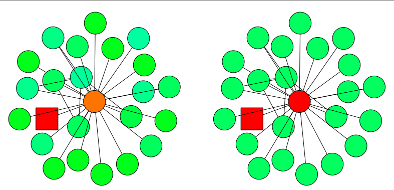
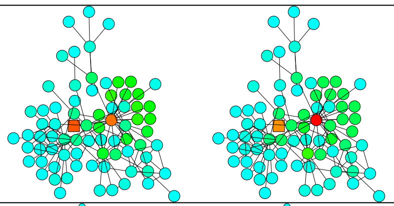
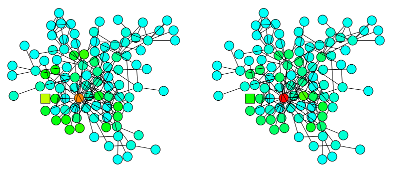
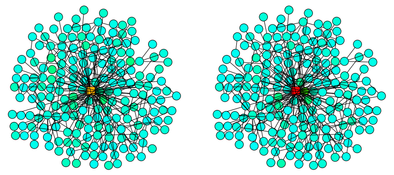
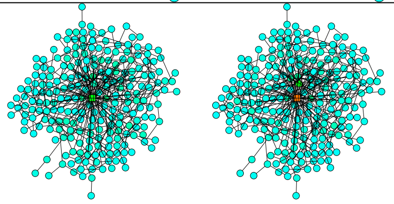
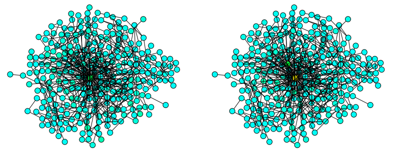
2 layers / steps	4 layers / steps	6 layers / steps
GCN / r.w.	GCN / r.w.	GCN / r.w.
		
		
		
		
		
		

Table A.1: Influence distributions under GCN and random walk distributions

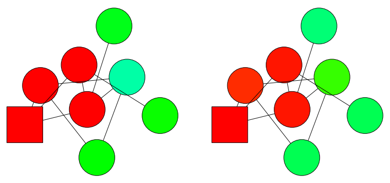
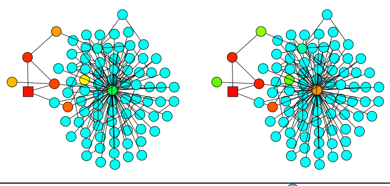
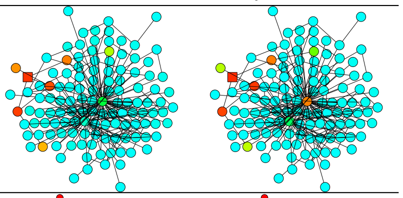
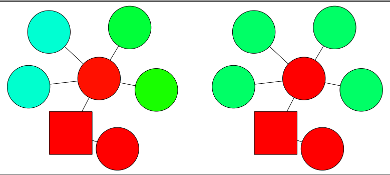
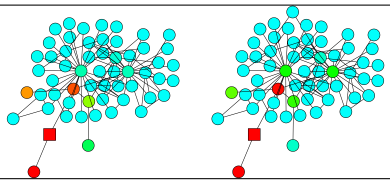
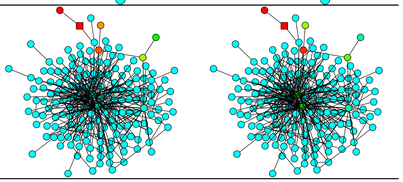
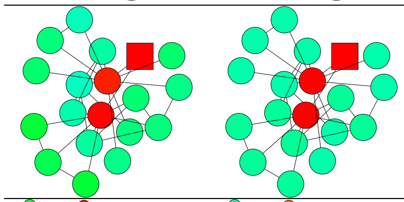
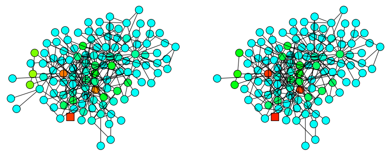
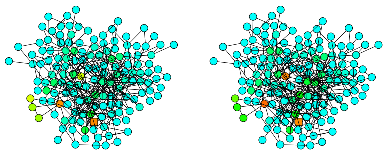
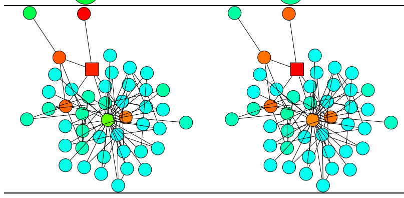
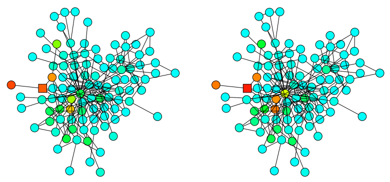
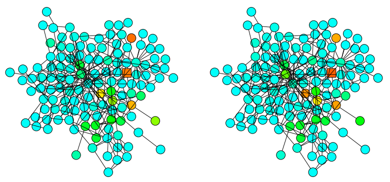
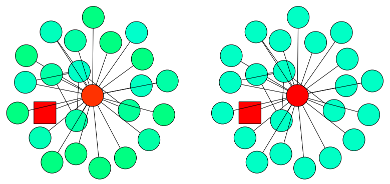
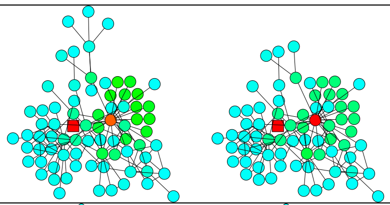
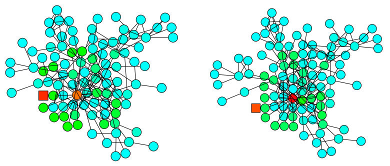
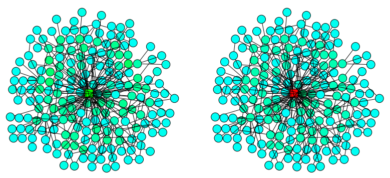
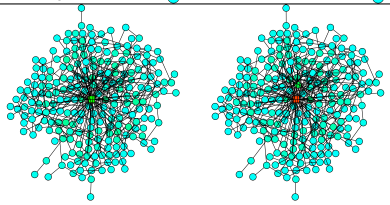
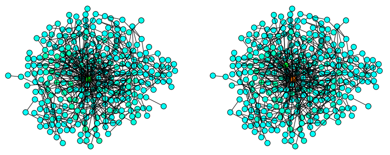
2 layers / steps	4 layers / steps	6 layers / steps
GCN-Res / lazy r.w.	GCN-Res / lazy r.w.	GCN-Res / lazy r.w.
		
		
		
		
		
		

Table A.2: Influence distributions under GCN with residual connections (GCN-Res) and lazy random walk distributions

In Table A.1 and Tabel A.2, we present more visualization results to compare the 1) influence distributions under GCNs and the random walk distributions, 2) influence distributions under GCNs with residual connections and lazy random walk distributions. The visualization results further empirically validate the equivalence of random walks and GCN influence distributions in Theorem 1. The nodes being influenced and the random walk starting node are labeled square. The influence distributions for the nodes in Figure A.1 are computed according to Definition 3.2, under the same trained GCN (Res) models with $2, 4, 6$ layers respectively. We use the hyper-parameters as described in Kipf and Welling [2017] for training the models. The graph (dataset) is taken from the Cora citation network. We compute the random walk distributions according to Definition 3.3 on the graph $\widetilde{G}$. The lazy random walks all share the same lazy factor 0.4, i.e. there's an extra 0.4 probability of staying at the current node at each step. This probability is chosen for visual comparison with the GCN-ResNet. The visualization in Figure 3-7 has the same setting as mentioned above. It is trained for the Cora dataset with a 6-layer JK Net with maxpooling layer aggregation.

Appendix B

How Powerful are Graph Neural Networks

B.1 Proofs

B.1.1 Proof of Lemma 4.1

Suppose after k iterations, a graph neural network $\mathcal{A}$ has $\mathcal{A}(G_1) \neq \mathcal{A}(G_2)$ but the WL test cannot decide G_1 and G_2 are non-isomorphic. It follows that from iteration 0 to k in the WL test, G_1 and G_2 always have the same collection of node labels. In particular, because G_1 and G_2 have the same WL node labels for iteration i and $i+1$ for any $i = 0, ..., k-1$, G_1 and G_2 have the same collection, i.e. multiset, of WL node labels $\left\{ l_v^{(i)} \right\}$ as well as the same collection of node neighborhoods $\left\{ \left(l_v^{(i)}, \left\{ l_u^{(i)} : u \in \mathcal{N}(v) \right\} \right) \right\}$. Otherwise, the WL test would have obtained different collections of node labels at iteration $i+1$ for G_1 and G_2 as different multisets get unique new labels.

The WL test always relabels different multisets of neighboring nodes into different new labels. We show that on the same graph $G = G_1$ or G_2, if WL node labels $l_v^{(i)} = l_u^{(i)}$, we always have GNN node features $h_v^{(i)} = h_u^{(i)}$ for any iteration i. This apparently holds for $i = 0$ because WL and GNN starts with the same node features. Suppose this holds for iteration j, if for any u, v, $l_v^{(j+1)} = l_u^{(j+1)}$, then it must be the case that

$$\left(l_v^{(j)}, \left\{l_w^{(j)} : w \in \mathcal{N}(v)\right\}\right) = \left(l_u^{(j)}, \left\{l_w^{(j)} : w \in \mathcal{N}(u)\right\}\right)$$

By our assumption on iteration j, we must have

$$\left(h_v^{(j)}, \left\{h_w^{(j)} : w \in \mathcal{N}(v)\right\}\right) = \left(h_u^{(j)}, \left\{h_w^{(j)} : w \in \mathcal{N}(u)\right\}\right)$$

In the aggregation process of the GNN, the same AGGREGATE and COMBINE are applied. The same input, i.e. neighborhood features, generates the same output. Thus, $h_v^{(j+1)} = h_u^{(j+1)}$. By induction, if WL node labels $l_v^{(i)} = l_u^{(i)}$, we always have GNN node features $h_v^{(i)} = h_u^{(i)}$ for any iteration i. This creates a valid mapping ϕ such that $h_v^{(i)} = \phi(l_v^{(i)})$ for any $v \in G$. It follows from G_1 and G_2 have the same multiset of WL neighborhood labels that G_1 and G_2 also have the same collection of GNN neighborhood features

$$\left\{\left(h_v^{(i)}, \left\{h_u^{(i)} : u \in \mathcal{N}(v)\right\}\right)\right\} = \left\{\left(\phi(l_v^{(i)}), \left\{\phi(l_u^{(i)}) : u \in \mathcal{N}(v)\right\}\right)\right\}$$

Thus, $\left\{h_v^{(i+1)}\right\}$ are the same. In particular, we have the same collection of GNN node features $\left\{h_v^{(k)}\right\}$ for G_1 and G_2. Because the graph level readout function is permutation invariant with respect to the collection of node features, $\mathcal{A}(G_1) = \mathcal{A}(G_2)$. Hence we have reached a contradiction.

B.1.2 Proof of Theorem 4.2

Let $\mathcal{A}$ be a graph neural network where the condition holds. Let G_1, G_2 be any graphs which the WL test decides as non-isomorphic at iteration K. Because the graph-level readout function is injective, i.e., it maps distinct multiset of node features into unique embeddings, it sufficies to show that $\mathcal{A}$'s neighborhood aggregation process, with sufficient iterations, embeds G_1 and G_2 into different multisets of node features. Let us assume $\mathcal{A}$ updates node representations as

$$h_v^{(k)} = \phi\left(h_v^{(k-1)}, f\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)\right)$$

with injective funtions f and ϕ. The WL test applies a predetermined injective hash function g to update the WL node labels $l_v^{(k)}$:

$$l_v^{(k)} = g\left(l_v^{(k-1)}, \left\{l_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)$$

We will show, by induction, that for any iteration k, there always exists an injective function φ such that $h_v^{(k)} = \varphi\left(l_v^{(k)}\right)$. This apparently holds for $k = 0$ because the initial node features are the same for WL and GNN $l_v^{(0)} = h_v^{(0)}$ for all $v \in G_1, G_2$. So φ could be the identity function for $k = 0$. Suppose this holds for iteration $k-1$, we show that it also holds for k. Substituting $h_v^{(k-1)}$ with $\varphi\left(l_v^{(k-1)}\right)$ gives us

$$h_v^{(k)} = \phi\left(\varphi\left(l_v^{(k-1)}\right), f\left(\left\{\varphi\left(l_u^{(k-1)}\right) : u \in \mathcal{N}(v)\right\}\right)\right).$$

Since the composition of injective functions is injective, there exists some injective function ψ so that

$$h_v^{(k)} = \psi\left(l_v^{(k-1)}, \left\{l_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)$$

Then we have

$$h_v^{(k)} = \psi \circ g^{-1} g\left(l_v^{(k-1)}, \left\{l_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right) = \psi \circ g^{-1}\left(l_v^{(k)}\right)$$

$\varphi = \psi \circ g^{-1}$ is injective because the composition of injective functions is injective. Hence for any iteration k, there always exists an injective function φ such that $h_v^{(k)} = \varphi\left(l_v^{(k)}\right)$. At the K-th iteration, the WL test decides that G_1 and G_2 are non-isomorphic, that is the multisets $\left\{l_v^{(K)}\right\}$ are different for G_1 and G_2. The graph neural network $\mathcal{A}$'s node embeddings $\left\{h_v^{(K)}\right\} = \left\{\varphi\left(l_v^{(K)}\right)\right\}$ must also be different for G_1 and G_2 because of the injectivity of φ.

B.1.3 Proof of Lemma 4.3

Before proving our lemma, we first show a well-known result that we will later reduce our problem to: $\mathbb{N}^k$ is countable for every $k \in \mathbb{N}$, i.e. finite Cartesian product of countable sets is countable. We observe that it suffices to show $\mathbb{N} \times \mathbb{N}$ is countable, because the proof then follows clearly from induction. To show $\mathbb{N} \times \mathbb{N}$ is countable, we construct a bijection ϕ from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$ as

$$\phi\left(m, n\right) = 2^{m-1} \cdot \left(2n - 1\right)$$

Now we go back to proving our lemma. If we can show that the range of any function g defined on multisets of bounded size from a countable set is also countable, then the lemma holds for any $g^{(k)}$ by induction. Thus, our goal is to show that the range of such g is countable. First, it is clear that the mapping from $g(X)$ to X is injective because g is a well-defined function. It follows that it suffices to show the set of all multisets $X \subset \mathcal{X}$ is countable.

Since the union of two countable sets is countable, the following set $\mathcal{X}'$ is also countable.

$$\mathcal{X}' = \mathcal{X} \cup \{e\}$$

where e is a dummy element that is not in $\mathcal{X}$. It follows from the result we showed above, *i.e.*, $\mathbb{N}^k$ is countable for every $k \in \mathbb{N}$, that $\mathcal{X}'^k$ is countable for every $k \in \mathbb{N}$. It remains to show there exists an injective mapping from the set of multisets in $\mathcal{X}$ to $\mathcal{X}'^k$ for some $k \in \mathbb{N}$.

We construct an injective mapping h from the set of multisets $X \subset \mathcal{X}$ to $\mathcal{X}'^k$ for some $k \in \mathbb{N}$ as follows. Because $\mathcal{X}$ is countable, there exists a mapping $Z : \mathcal{X} \to \mathbb{N}$ from $x \in \mathcal{X}$ to natural numbers. We can sort the elements $x \in X$ by $z(x)$ as $x_1, x_2, ..., x_n$, where $n = |X|$. Because the multisets X are of bounded size, there exists $k \in \mathbb{N}$ so that $|X| < k$ for all X. We can then define h as

$$h\left(X\right) = \left(x_1, x_2, ..., x_n, e, e, e...\right),$$

where the $k - n$ coordinates are filled with the dummy element e. It is clear that h is injective because for any multisets X and Y of bounded size, $h(X) = h(Y)$ only if X is equivalent to Y. Hence it follows that the range of g is countable as desired.

B.1.4 Proof of Lemma 4.4

We first prove that there exists a mapping f so that $\sum\limits_{x \in X} f(x)$ is unique for each multiset X of bounded size. Because $\mathcal{X}$ is countable, there exists a mapping $Z : \mathcal{X} \to \mathbb{N}$ from $x \in \mathcal{X}$ to natural numbers. Because the cardinality of multisets X is bounded, there exists a number $N \in \mathbb{N}$ so that $|X| < N$ for all X. Then an example of such f is $f(x) = N^{-Z(x)}$. This f can be viewed as a more compressed form of an one-hot vector or N-digit presentation. Thus, $h(X) = \sum\limits_{x \in X} f(x)$ is an injective function of multisets.

$\phi\left(\sum_{x \in X} f(x)\right)$ is permutation invariant so it is a well-defined multiset function. For any multiset function g, we can construct such ϕ by letting $\phi\left(\sum_{x \in X} f(x)\right) = g(X)$. Note that such ϕ is well-defined because $h(X) = \sum\limits_{x \in X} f(x)$ is injective.

B.1.5 Proof of Corollary 4.5

Following the proof of Lemma 4.4, we consider $f(x) = N^{-Z(x)}$, where N and $Z : \mathcal{X} \to \mathbb{N}$ are the same as defined in Appendix B.1.4. Let $h(c, X) \equiv (1 + \epsilon) \cdot f(c) + \sum_{x \in X} f(x)$. Our goal is show that for any $(c', X') \neq (c, X)$ with $c, c' \in \mathcal{X}$ and $X, X' \subset \mathcal{X}$, $h(c, X) \neq h(c', X')$ holds, if ϵ is an irrational number. We prove by contradiction. For any (c, X), suppose there exists (c', X') such that $(c', X') \neq (c, X)$ but $h(c, X) = h(c', X')$ holds. Let us consider the following two cases: (1) $c' = c$ but $X' \neq X$, and (2) $c' \neq c$. For the first case, $h(c, X) = h(c, X')$ implies $\sum_{x \in X} f(x) = \sum_{x \in X'} f(x)$. It follows from Lemma 4.4 that the equality will not hold, because with $f(x) = N^{-Z(x)}$, $X' \neq X$ implies $\sum_{x \in X} f(x) \neq \sum_{x \in X'} f(x)$. Thus, we reach a contradiction. For the second case, we can similarly rewrite $h(c, X) = h(c', X')$ as

$$\epsilon \cdot (f(c) - f(c')) = \left(f(c') + \sum_{x \in X'} f(x)\right) - \left(f(c) + \sum_{x \in X} f(x)\right). \tag{B.1}$$

Because ϵ is an irrational number and $f(c) - f(c')$ is a non-zero rational number, L.H.S. of B.1 is irrational. On the other hand, R.H.S. of B.1, the sum of a finite number of rational numbers, is rational. Hence the equality in B.1 cannot hold, and we have reached a contradiction.

For any function g over the pairs (c, X), we can construct such φ for the desired decomposition by letting $\varphi\left((1+\epsilon)\cdot f(c)+\sum_{x\in X} f(x)\right)=g(c,X)$. Note that such φ is well-defined because $h(c,X)=(1+\epsilon)\cdot f(c)+\sum_{x\in X} f(x)$ is injective.

B.1.6 Proof of Lemma 4.6

Let us consider the example $X_1=\{1,1,1,1,1\}$ and $X_2=\{2,3\}$, *i.e.* two different multisets of positive numbers that sum up to the same value. We will be using the homogeneity of ReLU.

Let W be an arbitrary linear transform that maps $x\in X_1, X_2$ into $\mathbb{R}^n$. It is clear that, at the same coordinates, Wx are either positive or negative for all x because all x in X_1 and X_2 are positive. It follows that ReLU(Wx) are either positive or 0 at the same coordinate for all x in X_1, X_2. For the coordinates where ReLU(Wx) are 0, we have $\sum_{x\in X_1}\text{ReLU}\,(Wx)=\sum_{x\in X_2}\text{ReLU}\,(Wx)$. For the coordinates where Wx are positive, linearity still holds. It follows from linearity that

$$\sum_{x\in X}\text{ReLU}\,(Wx)=\text{ReLU}\left(W\sum_{x\in X}x\right)$$

where X could be X_1 or X_2. Because $\sum_{x\in X_1}x=\sum_{x\in X_2}x$, we have the following as desired.

$$\sum_{x\in X_1}\text{ReLU}\,(Wx)=\sum_{x\in X_2}\text{ReLU}\,(Wx)$$

B.1.7 Proof of Corollary 4.7

Suppose multisets X_1 and X_2 have the same distribution, without loss of generality, let us assume $X_1=(S,m)$ and $X_2=(S,k\cdot m)$ for some $k\in\mathbb{N}_{\geq 1}$, i.e. X_1 and X_2 have the same underlying set and the multiplicity of each element in X_2 is k times of that in X_1. Then we have $|X_2|=k|X_1|$ and $\sum_{x\in X_2}f(x)=k\cdot\sum_{x\in X_1}f(x)$. Thus,

$$\frac{1}{|X_2|}\sum_{x\in X_2}f(x)=\frac{1}{k\cdot|X_1|}\cdot k\cdot\sum_{x\in X_1}f(x)=\frac{1}{|X_1|}\sum_{x\in X_1}f(x)$$

Now we show that there exists a function f so that $\frac{1}{|X|}\sum_{x\in X}f(x)$ is unique for distributionally equivalent X. Because $\mathcal{X}$ is countable, there exists a mapping $Z : \mathcal{X} \to \mathbb{N}$ from $x \in \mathcal{X}$ to natural numbers. Because the cardinality of multisets X is bounded, there exists a number $N \in \mathbb{N}$ so that $|X| < N$ for all X. Then an example of such f is $f(x) = N^{-2Z(x)}$.

B.1.8 Proof of Corollary 4.8

Suppose multisets X_1 and X_2 have the same underlying set S, then we have

$$\max_{x \in X_1} f(x) = \max_{x \in S} f(x) = \max_{x \in X_2} f(x)$$

Now we show that there exists a mapping f so that $\max_{x \in X} f(x)$ is unique for Xs with the same underlying set. Because $\mathcal{X}$ is countable, there exists a mapping $Z : \mathcal{X} \to \mathbb{N}$ from $x \in \mathcal{X}$ to natural numbers. Then an example of such $f : \mathcal{X} \to \mathbb{R}^\infty$ is defined as $f_i(x) = 1$ for $i = Z(x)$ and $f_i(x) = 0$ otherwise, where $f_i(x)$ is the i-th coordinate of $f(x)$. Such an f essentially maps a multiset to its one-hot embedding.

B.2 Details of datasets

We give detailed descriptions of datasets used in our experiments. Further details can be found in [Yanardag and Vishwanathan, 2015].

Social networks datasets. IMDB-BINARY and IMDB-MULTI are movie collaboration datasets. Each graph corresponds to an ego-network for each actor/actress, where nodes correspond to actors/actresses and an edge is drawn betwen two actors/actresses if they appear in the same movie. Each graph is derived from a pre-specified genre of movies, and the task is to classify the genre graph it is derived from. REDDIT-BINARY and REDDIT-MULTI5K are balanced datasets where each graph corresponds to an online discussion thread and nodes correspond to users. An edge was drawn between two nodes if at least one of them responded to another's comment. The task is to classify each graph to a community or a subreddit it belongs to. COLLAB is a scientific collaboration dataset, derived from 3 public collaboration datasets, namely, High Energy Physics, Condensed Matter Physics and

Astro Physics. Each graph corresponds to an ego-network of different researchers from each field. The task is to classify each graph to a field the corresponding researcher belongs to.

Bioinformatics datasets. MUTAG is a dataset of 188 mutagenic aromatic and heteroaromatic nitro compounds with 7 discrete labels. PROTEINS is a dataset where nodes are secondary structure elements (SSEs) and there is an edge between two nodes if they are neighbors in the amino-acid sequence or in 3D space. It has 3 discrete labels, representing helix, sheet or turn. PTC is a dataset of 344 chemical compounds that reports the carcinogenicity for male and female rats and it has 19 discrete labels. NCI1 is a dataset made publicly available by the National Cancer Institute (NCI) and is a subset of balanced datasets of chemical compounds screened for ability to suppress or inhibit the growth of a panel of human tumor cell lines, having 37 discrete labels.

Appendix C

Graph Neural Tangent Kernel

C.1 Proofs

C.1.1 Proof of Theorem 5.2

For two graph G and G', the GNTK kernel function that corresponds to the simple GNN can be described as

$$\Theta(G, G') = \sum_{u \in V, u' \in V'} \left(\left[\boldsymbol{\Sigma}_{(0)}^{(1)}(G, G')\right]_{uu'} \left[\dot{\boldsymbol{\Sigma}}_{(1)}^{(1)}(G, G')\right]_{uu'} + \left[\boldsymbol{\Sigma}_{(1)}^{(1)}(G, G')\right]_{uu'} \right).$$

Here, we have

$$\left[\boldsymbol{\Sigma}_{(0)}^{(1)}(G, G')\right]_{uu'} = c_u c_{u'} \left(\sum_{v \in \mathcal{N}(u) \cup \{u\}} \operatorname{vec} h_v \right)^\top \left(\sum_{v' \in \mathcal{N}(u') \cup \{u'\}} \operatorname{vec} h_{v'} \right) = \operatorname{vec} \overline{h}_u^\top \operatorname{vec} \overline{h}_{u'}.$$

Recall that

$$\begin{aligned}
\left[\boldsymbol{\Sigma}_{(r)}^{(\ell)}(G, G')\right]_{uu'} =& c_\sigma \mathbb{E}_{(a,b) \sim \mathcal{N}\left(\operatorname{vec} 0, \left[\boldsymbol{A}_{(r)}^{(\ell)}(G, G')\right]_{uu'}\right)} \left[\sigma(a) \sigma(b)\right], \\
\left[\dot{\boldsymbol{\Sigma}}_{(r)}^{(\ell)}(G, G')\right]_{uu'} =& c_\sigma \mathbb{E}_{(a,b) \sim \mathcal{N}\left(\operatorname{vec} 0, \left[\boldsymbol{A}_{(r)}^{(\ell)}(G, G')\right]_{uu'}\right)} \left[\dot{\sigma}(a) \dot{\sigma}(b)\right]
\end{aligned}$$

and

$$\left[\boldsymbol{A}^{(\ell)}_{(r)}\left(G, G'\right)\right]_{uu'} = \begin{pmatrix} \left[\boldsymbol{\Sigma}^{(\ell)}_{(r-1)}(G, G)\right]_{u,u} & \left[\boldsymbol{\Sigma}^{(\ell)}_{(r-1)}(G, G')\right]_{uu'} \\ \left[\boldsymbol{\Sigma}^{(\ell)}_{(r-1)}(G', G)\right]_{uu'} & \left[\boldsymbol{\Sigma}^{(\ell)}_{(r-1)}(G', G')\right]_{u'u'} \end{pmatrix} \in \mathbb{R}^{2\times 2}.$$

Since $\sigma(z) = \max\{0, z\}$ is the ReLU activation function, and $\dot{\sigma}(z) = \mathbb{1}[z \geq 0]$ is the derivative of the ReLU activation function, and $\|\overline{\operatorname{vec} h}_u\|_2 = 1$ for all nodes u, by calculation, we have

$$\begin{aligned} \left[\dot{\boldsymbol{\Sigma}}^{(1)}_{(1)}(G, G')\right]_{uu'} &= \frac{\pi - \arccos\left(\left[\boldsymbol{\Sigma}^{(1)}_{(0)}(G, G')\right]_{uu'}\right)}{2\pi}, \\ \left[\boldsymbol{\Sigma}^{(1)}_{(1)}(G, G')\right]_{uu'} &= \frac{\pi - \arccos\left(\left[\boldsymbol{\Sigma}^{(1)}_{(0)}(G, G')\right]_{uu'}\right) + \sqrt{1 - \left[\boldsymbol{\Sigma}^{(1)}_{(0)}(G, G')\right]^2_{uu'}}}{2\pi}. \end{aligned}$$

Since

$$\arcsin(x) = \sum_{l=0}^{\infty} \frac{(2l-1)!!}{(2l)!!} \cdot \frac{x^{2l+1}}{2l+1},$$

we have

$$\begin{aligned} \left[\boldsymbol{\Sigma}^{(1)}_{(0)}(G, G')\right]_{uu'} \left[\dot{\boldsymbol{\Sigma}}^{(1)}_{(1)}(G, G')\right]_{uu'} &= \frac{1}{4}\left[\boldsymbol{\Sigma}^{(1)}_{(0)}(G, G')\right]_{uu'} + \frac{1}{2\pi}\left[\boldsymbol{\Sigma}^{(1)}_{(0)}(G, G')\right]_{uu'} \arcsin\left(\left[\boldsymbol{\Sigma}^{(1)}_{(0)}(G, G')\right]_{uu'}\right) \\ &= \frac{1}{4}\left[\boldsymbol{\Sigma}^{(1)}_{(0)}(G, G')\right]_{uu'} + \frac{1}{2\pi}\sum_{l=1}^{\infty} \frac{(2l-3)!!}{(2l-2)!! \cdot (2l-1)} \cdot \left[\boldsymbol{\Sigma}^{(1)}_{(0)}(G, G')\right]^{2l}_{uu'} \\ &= \frac{1}{4} \operatorname{vec} \overline{h}_u^\top \operatorname{vec} \overline{h}_{u'} + \frac{1}{2\pi}\sum_{l=1}^{\infty} \frac{(2l-3)!!}{(2l-2)!! \cdot (2l-1)} \cdot \left(\operatorname{vec} \overline{h}_u^\top \operatorname{vec} \overline{h}_{u'}\right)^{2l}. \end{aligned}$$

Let $\operatorname{vec} \Phi^{(2l)}(\cdot)$ be the feature map of the polynomial kernel of degree $2l$, i.e.,

$$k^{(2l)}(\operatorname{vec} x, \operatorname{vec} y) = \left(\operatorname{vec} x^\top \operatorname{vec} y\right)^{2l} = \operatorname{vec} \Phi^{(2l)}(\operatorname{vec} x)^\top \operatorname{vec} \Phi^{(2l)}(\operatorname{vec} y).$$

We have

$$\begin{aligned}&\left[\mathbf{\Sigma}_{(0)}^{(1)}(G,G')\right]_{uu'}\left[\dot{\mathbf{\Sigma}}_{(1)}^{(1)}(G,G')\right]_{uu'}\\ =&\frac{1}{4}\operatorname{vec}\overline{h}_u^\top\operatorname{vec}\overline{h}_{u'}+\frac{1}{2\pi}\sum_{l=1}^{\infty}\frac{(2l-3)!!}{(2l-2)!!\cdot(2l-1)}\cdot\left(\operatorname{vec}\Phi^{(2l)}(\operatorname{vec}\overline{h}_u)\right)^\top\operatorname{vec}\Phi^{(2l)}(\operatorname{vec}\overline{h}_{u'}).\end{aligned}$$

Let

$$\Theta_1(G,G')=\sum_{u\in V,u'\in V'}\left[\mathbf{\Sigma}_{(0)}^{(1)}(G,G')\right]_{uu'}\left[\dot{\mathbf{\Sigma}}_{(1)}^{(1)}(G,G')\right]_{uu'},$$

we have

$$\begin{aligned}\Theta_1(G,G')=&\frac{1}{4}\left(\sum_{u\in V}\operatorname{vec}\overline{h}_u\right)^\top\left(\sum_{u'\in V'}\operatorname{vec}\overline{h}_{u'}\right)\\&+\frac{1}{2\pi}\sum_{l=1}^{\infty}\frac{(2l-3)!!}{(2l-2)!!\cdot(2l-1)}\cdot\left(\sum_{u\in V}\operatorname{vec}\Phi^{(2l)}(\operatorname{vec}\overline{h}_u)\right)^\top\left(\sum_{u'\in V'}\operatorname{vec}\Phi^{(2l)}(\operatorname{vec}\overline{h}_{u'})\right).\end{aligned}$$

Since $\mathbf{\Theta}=\mathbf{\Theta}_1+\mathbf{\Theta}_2$ where $\mathbf{\Theta}_2$ is a kernel matrix (and thus positive semi-definite), for any $y\in\mathbb{R}^n$, we have

$$\operatorname{vec}y^\top\mathbf{\Theta}^{-1}\operatorname{vec}y\le\operatorname{vec}y^\top\mathbf{\Theta}_1^{-1}\operatorname{vec}y.$$

Recall that

$$y_i=\alpha_1\sum_{u\in V}\left(\operatorname{vec}\overline{h}_u^\top\operatorname{vec}\beta_1\right)+\sum_{l=1}^{\infty}\alpha_{2l}\sum_{u\in V}\left(\operatorname{vec}\overline{h}_u^\top\operatorname{vec}\beta_{2l}\right)^{2l}.$$

We rewrite

$$y_i=y_i^{(0)}+\sum_{l=1}^{\infty}y_i^{(2l)},$$

where

$$y_i^{(0)}=\alpha_1\left(\sum_{u\in V}\operatorname{vec}\overline{h}_u\right)^\top\operatorname{vec}\beta_1,$$

and for each $l \geq 1$,

$$\begin{aligned}
y_i^{(2l)} &= \alpha_{2l} \sum_{u \in V} \left(\operatorname{vec} \overline{h}_u^\top \operatorname{vec} \beta_{2l} \right)^{2l} \\
&= \alpha_{2l} \sum_{u \in V} \left(\operatorname{vec} \Phi^{2l} \left(\operatorname{vec} \overline{h}_u \right) \right)^\top \operatorname{vec} \Phi^{2l} \left(\operatorname{vec} \beta_{2l} \right) \\
&= \alpha_{2l} \left(\sum_{u \in V} \operatorname{vec} \Phi^{2l} \left(\operatorname{vec} \overline{h}_u \right) \right)^\top \operatorname{vec} \Phi^{2l} \left(\operatorname{vec} \beta_{2l} \right).
\end{aligned}$$

We have

$$\operatorname{vec} y = \operatorname{vec} y^{(0)} + \sum_{l=1}^{\infty} \operatorname{vec} y^{(2l)}.$$

Thus,

$$\begin{aligned}
\sqrt{\operatorname{vec} y^\top \boldsymbol{\Theta}^{-1} \operatorname{vec} y} &\leq \sqrt{\operatorname{vec} y^\top \boldsymbol{\Theta}_1^{-1} \operatorname{vec} y} \\
&\leq \sqrt{\left(\operatorname{vec} y^{(0)} \right)^\top \boldsymbol{\Theta}_1^{-1} \operatorname{vec} y^{(0)}} + \sum_{l=1}^{\infty} \sqrt{\left(\operatorname{vec} y^{(2l)} \right)^\top \boldsymbol{\Theta}_1^{-1} \operatorname{vec} y^{(2l)}}.
\end{aligned}$$

When $l = 0$, we have

$$\sqrt{\left(\operatorname{vec} y^0 \right)^\top \boldsymbol{\Theta}_1^{-1} \operatorname{vec} y^0} \leq 2|\alpha_1| \| \operatorname{vec} \beta_1 \|_2.$$

When $l \geq 1$, we have

$$\sqrt{\left(\operatorname{vec} y^{2l} \right)^\top \boldsymbol{\Theta}_1^{-1} \operatorname{vec} y^{2l}} \leq \sqrt{2\pi}(2l-1)|\alpha_{2l}| \left\| \operatorname{vec} \Phi^{2l} \left(\operatorname{vec} \beta_{2l} \right) \right\|_2.$$

Notice that

$$\left\| \operatorname{vec} \Phi^{2l} \left(\operatorname{vec} \beta_{2l} \right) \right\|_2^2 = \left(\operatorname{vec} \Phi^{2l} \left(\operatorname{vec} \beta_{2l} \right) \right)^\top \operatorname{vec} \Phi^{2l} \left(\operatorname{vec} \beta_{2l} \right) = \left\| \operatorname{vec} \beta_{2l} \right\|_2^{4l}.$$

Thus,

$$\sqrt{\operatorname{vec} y^\top \boldsymbol{\Theta}^{-1} \operatorname{vec} y} \leq 2|\alpha_1| \| \operatorname{vec} \beta_1 \|_2 + \sum_{l=1}^{\infty} \sqrt{2\pi}(2l-1)|\alpha_{2l}| \| \operatorname{vec} \beta_{2l} \|_2^{2l}.$$

C.1.2 Proof of Theorem 5.3

Recall that

$$\Theta(G,G') = \sum_{u\in V, u'\in V'} \left(\left[\mathbf{\Sigma}_{(0)}^{(1)}(G,G')\right]_{uu'} \left[\dot{\mathbf{\Sigma}}_{(1)}^{(1)}(G,G')\right]_{uu'} + \left[\mathbf{\Sigma}_{(1)}^{(1)}(G,G')\right]_{uu'} \right),$$

where

$$\left[\mathbf{\Sigma}_{(0)}^{(1)}(G,G')\right]_{uu'} = c_u c_{u'} \left(\sum_{v\in\mathcal{N}(u)\cup\{u\}} \operatorname{vec} h_v \right)^\top \left(\sum_{v'\in\mathcal{N}(u')\cup\{u'\}} \operatorname{vec} h_{v'} \right) = \operatorname{vec} \overline{h}_u^\top \operatorname{vec} \overline{h}_{u'}$$

and

$$\begin{aligned} \left[\dot{\mathbf{\Sigma}}_{(1)}^{(1)}(G,G')\right]_{uu'} &= \frac{\pi - \arccos\left(\left[\mathbf{\Sigma}_{(0)}^{(1)}(G,G')\right]_{uu'}\right)}{2\pi}, \\ \left[\mathbf{\Sigma}_{(1)}^{(1)}(G,G')\right]_{uu'} &= \frac{\pi - \arccos\left(\left[\mathbf{\Sigma}_{(0)}^{(1)}(G,G')\right]_{uu'}\right) + \sqrt{1 - \left[\mathbf{\Sigma}_{(0)}^{(1)}(G,G')\right]_{uu'}^2}}{2\pi}. \end{aligned}$$

Since for each node u, $\operatorname{vec} \overline{h}_u = c_u \sum_{v\in\mathcal{N}(u)\cup\{u\}} \operatorname{vec} h_v$, and $c_u = \left(\left\| \sum_{v\in\mathcal{N}(u)\cup\{u\}} \operatorname{vec} h_v \right\|_2 \right)^{-1}$, we have $\|\operatorname{vec} \overline{h}_u\|_2 = 1$. Moreover,

$$\left[\dot{\mathbf{\Sigma}}_{(1)}^{(1)}(G,G')\right]_{uu'} = \frac{\pi - \arccos\left(\left[\mathbf{\Sigma}_{(0)}^{(1)}(G,G')\right]_{uu'}\right)}{2\pi} \leq 1/2$$

and

$$\left[\mathbf{\Sigma}_{(1)}^{(1)}(G,G')\right]_{uu'} = \frac{\pi - \arccos\left(\left[\mathbf{\Sigma}_{(0)}^{(1)}(G,G')\right]_{uu'}\right) + \sqrt{1 - \left[\mathbf{\Sigma}_{(0)}^{(1)}(G,G')\right]_{uu'}^2}}{2\pi} \leq \frac{1+\pi}{2\pi} \leq 1,$$

we have

$$\Theta(G,G') \leq 2|V||V'|.$$

Thus,

$$\operatorname{tr}(\mathbf{\Theta}) \leq 2n\overline{V}^2.$$

C.2 Experimental Setup

To calculate the expectation of the post-activation output, i.e., (5.5) and (5.6), we use the same approach as in Arora et al. [2019c] (cf. Section 4.3 in Arora et al. [2019c]).

For GNTKs, we tune the following hyperparameters.

1. The number of BLOCK operations. We search from candidate values $\{1, 2, \ldots, 14\}$.
2. The number of fully-connected layers in each BLOCK operation. We search from candidate values $\{1, 2, 3\}$.
3. The parameter c_u. We search from candidate values $\left\{1, \frac{1}{|\mathcal{N}(u)|+1}\right\}$.

To utilize the GNTKs we compute to perform graph classification, we test with kernel regression and C-SVM as the final classifier. In our experiments, the regularization parameter C in C-SVM is determined using grid search from 120 values evenly chosen from $[10^{-2}, 10^{4}]$, in log scale.

We would like to remark that GNTK has strictly smaller number of hyper-parameters than GNN since we do not need to tune the learning rate, momentum, weight decay, batch size and the width of the MLP layers for GNTK. Furthermore, we find on bioinformatics datasets, we get consistently good results by setting the number of BLOCK operations to be 10, the number of MLP layers to be 1 and c_u to be $1/|\mathcal{N}(u)|$. We get 75.3% accuracy on PROTEINS, 67.9% on PTC, and 83.6% on NCI1. For social network datasets, by setting the number of BLOCK operations to be 2, the number of MLP layers to be 2 and c_u to be 1, we get 76.7% accuracy on IMDB-B, 52.8% on IMDB-M, and 83.3% on COLLAB.

Appendix D

What Can Neural Networks Reason About

D.1 Proofs

D.1.1 Proof of Proposition 6.1

We proceed the proof of the universal approximation of GNNs on complete graphs, i.e., sets, by showing that GNNs are at least as power as Deep Sets. We then apply the universal approximation of Deep Sets for permutation invariant continuous functions.

We show any Deep Sets can be expressed by some GNN with one message passing iteration. The computation structure of one-layer GNNs is shown below.

$$h_s = \sum_{t \in S} \phi\left(X_s, X_t\right), \quad h_S = g\left(\sum_{s \in S} h_s\right), \tag{D.1}$$

where ϕ and g are parameterized by MLPs. If ϕ is a function that ignores X_t so that $\phi\left(X_s, X_t\right) = \rho(X_s)$ for some ρ, *e.g.*, by letting part of the weight matricies in ϕ be 0, then we essentially get a Deep Sets in the following form.

$$h_s = \rho\left(X_s\right), \quad h_S = g\left(\sum_{s \in S} h_s\right). \tag{D.2}$$

For any such ρ, we can get the corresponding ϕ via the construction above. Hence for any Deep Sets, we can express it with an one-layer GNN. The same result applies to GNNs with multiple layers (message passing iterations), because we can express a function $\rho(X_s)$ by the composition of multiple $\rho^{(k)}$'s, which we can express with a GNN layer via our construction above. It then follows that GNNs are universal approximators for permutation invariant continuous functions by the following fact.

Zaheer et al. [2017] prove the universal approximation of Deep Sets under the restriction that the set size is fixed and the hidden dimension is equal to the set size plus one. Wagstaff et al. [2019] extend the universal approximation result for Deep Sets by showing that the set size does not have to be fixed and the hidden dimension is only required to be at least as large as the set size. The results for our purposes can be summarized as follows. Assume the elements are from a compact set in $\mathbb{R}^d$. Any continuous function on a set S of size bounded by N, *i.e.*, $f : \mathbb{R}^{d \times N} \to \mathbb{R}$, that is permutation invariant to the elements in S can be approximated arbitrarily close by some Deep Sets model with sufficiently large width and output dimension for its MLPs.

D.1.2 Proof of Proposition 6.2

For any GNN $\mathcal{N}$, we construct an MLP that is able to do the exact same computation as $\mathcal{N}$. It will then follow that the MLP can represent any function $\mathcal{N}$ can represent. Suppose the computation structure of $\mathcal{N}$ is the following.

$$h_s^{(k)} = \sum_{t \in S} f^{(k)}\left(h_s^{(k-1)}, h_t^{(k-1)}\right), \quad h_S = g\left(\sum_{s \in S} h_s^{(K)}\right), \tag{D.3}$$

where f and g are parameterized by MLPs. Suppose the set size is bounded by M (the expressive power of GNNs also depend on M Wagstaff et al. [2019]). We first show the result for a fixed size input, *i.e.*, MLPs can simulate GNNs if the input set has a fixed size, and then apply an ensemble approach to deal with variable sized input.

Let the input to the MLP be a vector concatenated by $h_s^{(0)}$'s, in some arbitrary ordering. For each message passing iteration of $\mathcal{N}$, any $f^{(k)}$ can be represented by an MLP. Thus, for each pair of $(h_t^{(k-1)}, h_s^{(k-1)})$, we can set weights in the MLP so that the the concatenation of all $f(h_t^{(k-1)}, h_s^{(k-1)})$ become the hidden vector after some layers of the MLP. With the vector of $f(h_t^{(k-1)}, h_s^{(k-1)})$ as input, in the next few layers of the MLP we can construct weights so that we have the concatenation of $h_s^{(k)} = \sum_{t \in S} f^{(k)}\left(h_s^{(k-1)}, h_t^{(k-1)}\right)$ as the result of the hidden dimension, because we can encode summation with weights in MLPs. So far, we can simulate an iteration of GNN $\mathcal{N}$ with layers of MLP. We can repeat the process for K times by stacking the similar layers. Finally, with a concatenation of $h_s^{(K)}$ as our hidden dimension in the MLP, similarly, we can simulate $h_S = g\left(\sum_{s \in S} h_s^{(K)}\right)$ with layers of MLP. Stacking all layers together, we have obtained an MLP that can simulate $\mathcal{N}$.

To deal with variable sized inputs, we construct M MLPs that can simulate the GNN for each input set size $1, ..., M$. Then we construct a meta-layer, whose weights represent (universally approximate) the summation of the output of M MLPs multiplied by an indicator function of whether each MLPs has the same size as the set input (these need to be input information). The meta layer weights on top can then essentially select the output from of MLP that has the same size as the set input and then exactly simulate the GNN. Note that the MLP we construct here has the requirement for how we input the data and the information of set sizes etc. In practice, we can have M MLPs and decide which MLP to use depending on the input set size.

D.1.3 Proof of Theorem 6.3

Theorem 6.3 is a generalization of Theorem 6.1 in [Arora et al., 2019b], which addresses the scalar case. See [Arora et al., 2019b] for a complete list of assumptions.

Theorem D.1. *[Arora et al., 2019b] Suppose we have $g : \mathbb{R}^d \to \mathbb{R}$, $g(x) = \sum_j \alpha_j \left(\beta_j^\top x\right)^{p_j}$, where $\beta_j \in \mathbb{R}^d$, $\alpha \in \mathbb{R}$, and $p_j = 1$ or $p_j = 2l$ $(l \in \mathbb{N}_+)$. Let $\mathcal{A}$ be an overparameterized two-layer MLP that is randomly initialized and trained with gradient descent for a sufficient number of iterations. The sample complexity $\mathcal{C}_\mathcal{A}(g, \epsilon, \delta)$ is $O\left(\frac{\sum_j p_j |\alpha_j| \cdot \|\beta_j\|_2^{p_j} + \log(1/\delta)}{\epsilon^2}\right)$.*

To extend the sample complexity bound to vector-valued functions, we view each entry/component of the output vector as an independent scalar-valued output. We can then apply a union bound to bound the error rate and failure probability for the output vector, and thus, bound the overall sample complexity.

Let ϵ and δ be the given error rate and failure probability. Moreover, suppose we choose some error rate ϵ_0 and failure probability δ_0 for the output/function of each entry. Applying Theorem D.1 to each component

$$g(x)^{(i)} = \sum_j \alpha_j^{(i)} \left(\beta_j^{(i)\top} x\right)^{p_j^{(i)}} =: g_i(x) \tag{D.4}$$

yields a sample complexity bound of

$$\mathcal{C}_{\mathcal{A}}(g_i, \epsilon_0, \delta_0) = O\left(\frac{\sum_j p_j^{(i)} |\alpha_j^{(i)}| \cdot \|\beta_j^{(i)}\|_2^{p_j^{(i)}} + \log\left(1/\delta_0\right)}{\epsilon_0^2}\right) \tag{D.5}$$

for each $g_i(x)$. Now let us bound the overall error rate and failure probability given ϵ_0 and δ_0 for each entry. The probability that we fail to learn each of the g_i is at most δ_0. Hence, by a union bound, the probability that we fail to learn any of the g_i is at most $m \cdot \delta_0$. Thus, with probability at least $1 - m\delta_0$, we successfully learn all g_i for $i = 1, ..., m$, so the error for every entry is bounded by ϵ_0. The error for the vector output is then at most $\sum_{i=1}^m \epsilon_0 = m\epsilon_0$.

Setting $m\delta_0 = \delta$ and $m\epsilon_0 = \epsilon$ gives us $\delta_0 = \frac{\delta}{m}$ and $\epsilon_0 = \frac{\epsilon}{m}$. Thus, if we can successfully learn the function for each output entry independently with error ϵ/m and failure rate δ/m, we can successfully learn the entire vector-valued function with rate ϵ and δ. This yields the following overall sample complexity bound:

$$\mathcal{C}_{\mathcal{A}}(g, \epsilon, \delta) = O\left(\frac{\max_i \sum_j p_j^{(i)} |\alpha_j^{(i)}| \cdot \|\beta_j^{(i)}\|_2^{p_j^{(i)}} + \log\left(m/\delta\right)}{\left(\epsilon/m\right)^2}\right) \tag{D.6}$$

Regarding m as a constant, we can further simplify the sample complexity to

$$\mathcal{C}_{\mathcal{A}}(g, \epsilon, \delta) = O\left(\frac{\max_i \sum_j p_j^{(i)} |\alpha_j^{(i)}| \cdot \|\beta_j^{(i)}\|_2^{p_j^{(i)}} + \log\left(1/\delta\right)}{\epsilon^2}\right). \tag{D.7}$$

D.1.4 Proof of Theorem 6.4

We will show the learnability result by an inductive argument. Specifically, we will show that under our setting and assumptions, the error between the learned function and correct function on the test set will not blow up after the transform of another learned function $\hat{f}_j$, assuming learnability on previous $\hat{f}_1, ..., \hat{f}_{j-1}$ by induction. Thus, we can essentially provably learn at all layers/iterations and eventually learn g.

Suppose we have performed the sequential learning. Let us consider what happens at the test time. Let f_j be the *correct functions* as defined in the algorithmic alignment. Let $\hat{f}_j$ be the functions learned by algorithm $\mathcal{A}_j$ and MLP $\mathcal{N}_j$. We have input $S \sim \mathcal{D}$, and our goal is to bound $\|g(S) - \hat{g}(S)\|$ with high probability. To show this, we bound the error of the intermediate representation vectors, *i.e.*, the output of $\hat{f}_j$ and f_j, and thus, the input to $\hat{f}_{j+1}$ and f_{j+1}.

Let us first consider what happens for the first module $\mathcal{N}_1$. f_1 and $\hat{f}_1$ have the same input distribution $x \sim \mathcal{D}$, where x are obtained from S, *e.g.*, the pairwise object representations as in Eqn. 6.2. Hence, by the learnability assumption on $\mathcal{A}_1$, $\|f_1(x) - \hat{f}_1(x)\| < \epsilon$ with probability at least $1 - \delta$. The error for the input of $\mathcal{N}_2$ is then $O(\epsilon)$ with failure probability $O(\delta)$, because there are a constant number of terms of aggregation of f_1's output, and we can apply union bound to upper bound the failure probability.

Next, we proceed by induction. Let us fix a k. Let z denote the input for f_k, which are generated by the previous f_j's, and let $\hat{z}$ denote the input for $\hat{f}_k$, which are generated by the previous $\hat{f}_j$'s. Assume $\|z - \hat{z}\| \leq O(\epsilon)$ with failure probability at most $O(\delta)$. We aim to show that this holds for $k + 1$. For the simplicity of notation, let f denote the correct function f_k and let $\hat{f}$ denote the learned function $\hat{f}_k$. Since there are a constant number of terms for aggregation, our goal is then to bound $\|\hat{f}(\hat{z}) - f(z)\|$. By triangle inequality, we have

$$\|\hat{f}(\hat{z}) - f(z)\| = \|\hat{f}(\hat{z}) - \hat{f}(z) + \hat{f}(z) - f(z)\| \tag{D.8}$$

$$\leq \|\hat{f}(\hat{z}) - \hat{f}(z)\| + \|\hat{f}(z) - f(z)\| \tag{D.9}$$

We can bound the first term with the Lipschitzness assumption of $\hat{f}$ as the following.

$$\|\hat{f}(\hat{z}) - \hat{f}(z)\| \leq L_1 \|\hat{z} - z\| \tag{D.10}$$

To bound the second term, our key insight is that f is a *learnale correct function*, so by the learnability coefficients in algorithmic alignment, it is close to the function $\tilde{f}$ learned by the learning algorithm $\mathcal{A}$ on the *correct samples*, *i.e.*, f is close to $\tilde{f} = \mathcal{A}(\{z_i, y_i\})$. Moreover, $\hat{f}$ is generated by the learning algorithm $\mathcal{A}$ on the perturbed samples, *i.e.*, $\hat{f} = \mathcal{A}(\{\hat{z}_i, y_i\})$. By the algorithm stability assumption, $\hat{f}$ and $\tilde{f}$ should be close if the input samples are only slightly perturbed. It then follows that

$$\|\hat{f}(z) - f(z)\| = \|\hat{f}(z) - \tilde{f}(z) + \tilde{f}(z) - f(z)\| \tag{D.11}$$

$$\leq \|\hat{f}(z) - \tilde{f}(z)\| + \|\tilde{f}(z) - f(z)\| \tag{D.12}$$

$$\leq L_0 \max_i \|z_i - \hat{z}_i\| + \epsilon \quad \text{w.p. } \geq 1 - \delta \tag{D.13}$$

where z_i and $\hat{z}_i$ are the training samples at the same layer k. Here, we apply the same induction condition as what we had for z and $\hat{z}$: $\|z_i - \hat{z}_i\| \leq O(\epsilon)$ with failure probability at most $O(\delta)$. We can then apply union bound to bound the probability of any bad event happening. Here, we have 3 bad events each happening with probability at most $O(\delta)$. Thus, with probability at least $1 - O(\delta)$, we have

$$\|\hat{f}(\hat{z}) - f(z)\| \leq L_1 O(\epsilon) + L_0 O(\epsilon) + \epsilon = O(\epsilon) \tag{D.14}$$

This completes the proof.

D.1.5 Proof of Corollary 6.5

Our main insight is that a giant MLP learns the same function $(X_i - X_j)^2$ for ℓ^2 times and encode them in the weights. This leads to the $O(\ell^2)$ extra sample complexity through Theorem 6.3, because the number of polynomial terms $(X_i - X_j)^2$ is of order ℓ^2.

First of all, the function $f(x, y) = (x - y)^2$ can be expressed as the following polynomial.

$$(x - y)^2 = \left([1 \ \ -1]^\top [x \ y]\right)^2 \tag{D.15}$$

We have $\beta = [1 - 1]$, so $p \cdot \|\beta\|^p = 4$. Hence, by Theorem 6.3, it takes $O(\frac{\log(1/\delta)}{\epsilon^2})$ samples for an MLP to learn $f(x, y) = (x - y)^2$. Under the sequential training setting, an one-layer GNN applies an MLP to learn f, and then sums up the outcome of $f(X_i, X_j)$ for all pairs X_i, X_j. Here, we essentially get the aggregation error $O(\ell^2 \cdot \epsilon)$ from ℓ^2 pairs. However, we will see that applying an MLP to learn g will also incur the same aggregation error. Hence, we do not need to consider the aggregation error effect when we compare the sample complexities.

Now we consider using MLP to learn the function g. No matter in what order the objects X_i are concatenated, we can express g with the sum of polynomials as the following.

$$g(S) = \sum_{ij} (\beta_{ij}^\top [X_1, ..., X_n])^2, \tag{D.16}$$

where β_{ij} has 1 at the i-th entry, -1 at the j-th entry and 0 elsewhere. Hence $\|\beta_{ij}\|^p \cdot p = 4$. It then follows from Theorem 6.3 and union bound that it takes $O((\ell^2 + \log(1/\hat{\delta}))/\hat{\epsilon}^2)$ to learn g, where $\hat{\epsilon} = \ell^2 \epsilon$ and $\hat{\delta} = \ell^2 \delta$. Here, as we have discussed above, the same aggregation error $\hat{\epsilon}$ occurs in the aggregation process of f, so we can simply consider $\hat{\epsilon}$ for both. Thus, comparing $O(\log(1/\hat{\delta})/\hat{\epsilon}^2)$ and $O((\ell^2 + \log(1/\hat{\delta}))/\hat{\epsilon}^2)$ gives us the $O(\ell^2)$ difference.

D.1.6 Proof of Claim 6.6

We prove the claim by contradiction. Suppose there exists f such that $f(x) + f(y) = g(x, y)$ for any x and y. This implies that for any x, we have $f(x) + f(x) = g(x, x) = 0$. It follows that $f(x) = 0$ for any x. Now consider some x and y so that $x \neq y$. We must have $f(x) + f(y) = 0 + 0 = 0$. However, $g(x, y) \neq 0$ because $x \neq y$. Hence, there exists x and y so that $f(x) + f(y) \neq g(x, y)$. We have reached a contradiction.

D.2 Experimental Details

D.2.1 Fantastic Treasure: Maximum Value Difference

Dataset generation. In the dataset, we sample $50,000$ training data, $5,000$ validation data, and $5,000$ test data. For each model, we report the test accuracy with the hyperparameter setting that achieves the best validation accuracy. In each training sample, the input universe consists of 25 treasures $X_1, ..., X_{25}$. For each treasure X_i, we have $X_i = [h_1, h_2, h_3]$, where the location h_1 is sampled uniformly from $[0..20]^8$, the value h_2 is sample uniformly form $[0..100]$, and the color h_3 is sampled uniformly from $[1..6]$. The task is to answer what the difference is in value between the most and least valuable treasure. We generate the answer label y for a universe S as follows: we find the the maximum difference in value among all treasures and set it to y. Then we make the label y into one-hot encoding with $100 + 1 = 101$ classes.

Hyperparameter setting. We train all models with the Adam optimizer, with learning rate from $1e-3, 5e-4$, and $1e-4$, and we decay the learning rate by 0.5 every 50 steps. We use cross-entropy loss. We train all models for 150 epochs. We tune batch size of 128 and 64.

For GNNs and HRN, we choose the hidden dimension of MLP modules from 128 and 256. For DeepSet and MLP, we choose the hidden dimension of MLP modules from 128, 256, 2500, 5000. For the MLP and DeepSet model, we choose the number of of hidden layers for MLP moduels from 4 and 8, 16. For GNN and HRN, we set the number of hidden layers of the MLP modules to 3, 4. Moreover, dropout with rate 0.5 is applied before the last two hidden layers of MLP_1, *i.e.*, the last MLP module in all models.

D.2.2 Fantastic Treasure: Furthest Pair

Dataset generation. In the dataset, we sample $60,000$ training data, $6,000$ validation data, and $6,000$ test data. For each model, we report the test accuracy with the hyperparameter setting that achieves the best validation accuracy. In each training sample, the input universe consists of 25 treasures $X_1, ..., X_{25}$. For each treasure X_i, we have $X_i = [h_1, h_2, h_3]$, where the location h_1 is sampled uniformly from $[0..20]^8$, the value h_2 is sample uniformly form $[0..100]$, and the color h_3 is sampled uniformly from $[1..6]$. The task is to answer what are the colors of the two treasure that are the most distant from each other. We generate the answer label y for a universe S as follows: we find the pair of treasures that are the most distant from each other, say (X_i, X_j). Then we order the pair $(h_3(X_i), h_3(X_j))$ to obtain an ordered pair (a, b) with $a \leq b$ (aka. $a = \min\{h_3(X_i), h_3(X_j)\}$ and ($b = \max\{h_3(X_i), h_3(X_j)\}$), where $h_3(X_i)$ denotes the color of X_i. Then we compute the label y from (a, b) by counting how many valid pairs of colors are smaller than (a, b) (a pair (k, l) is smaller than (a, b) iff i). $k < a$ or ii). $k = a$ and $l < b$). The label y is one-hot encoding of the minimum cost with $6 \times (6-1)/2 + 6 = 21$ classes.

Hyperparameter setting. We train all models with the Adam optimizer, with learning rate from $1e-3, 5e-4$, and $1e-4$, and we decay the learning rate by 0.5 every 50 steps. We use cross-entropy loss. We train all models for 150 epochs. We tune batch size of 128 and 64.

For the MLP and DeepSet model, we choose the number of of hidden layers of MLP modules from 4 and 8, 16. For GNN and HRN models, we set the number of hidden layers of the MLP modules from 3 and 4. For DeepSet and MLP models, we choose the hidden dimension of MLP modules from 128, 256, 2500, 5000. For GNNs and HRN, we choose the hidden dimension of MLP modules from 128 and 256. Moreover, dropout with rate 0.5 is applied before the last two hidden layers of MLP_1, *i.e.*, the last MLP module in all models.

D.2.3 Monster Trainer

Task description. We are a monster trainer who lives in a world S with 10 monsters. Each monster $X = [h_1, h_2]$ has a location $h_1 \in [0..10]^2$ and a unique combat level $h_2 \in [1..10]$. In each game, the trainer starts at a random location with level zero, $X_{\text{trainer}} = [p_0, 0]$, and receives a quest to defeat the level-k monster. At each time step, the trainer can challenge any *more powerful* monster X, with a cost equal to the product of the travel distance and the level difference $c(X_{\text{trainer}}, X) = \|h_1(X_{\text{trainer}}) - h_1(X)\|_{\ell_1} \times (h_2(X) - h_2(X_{\text{trainer}}))$. After defeating monster X, the trainer's level upgrades to $h_2(X)$, and the trainer moves to $h_1(X)$. We ask the minimum cost of completing the quest, *i.e.*, defeating the level-k monster. The range of cost (number of classes for prediction) is 200. To make games even more challenging, we sample games whose optimal solution involves defeating three to seven non-quest monsters.

A DP algorithm for shortest paths that needs half of the iterations of Bellman-

Ford. We provide a DP algorithm as the following. To compute a shortest-path from a source object s to a target object t with at most seven stops, we run the following updates for four iterations:

$$\text{distance}_s[1][u] = \text{cost}(s, u), \qquad \text{distance}_s[k][u] = \min_v \Big\{ \text{distance}_s[k-1][v] + \text{cost}(v, u) \Big\}, \tag{D.17}$$

$$\text{distance}_t[1][u] = \text{cost}(u, t), \qquad \text{distance}_t[k][u] = \min_v \Big\{ \text{distance}_t[k-1][v] + \text{cost}(u, v) \Big\}. \tag{D.18}$$

Update Eqn. D.17 is identical to the Bellman-Ford algorithm Eqn. 6.6, and distance$_s[k][u]$ is the shortest distance from s to u with at most k stops. Update Eqn. D.18 is a *reverse* Bellman-Ford algorithm, and distance$_t[k][u]$ is the shortest distance from u to t with at most k stops. After running Eqn. D.17 and Eqn. D.18 for k iterations, we can compute a shortest path with at most $2k$ stops by enumerating a mid-point and aggregating the results of the two Bellman-Ford algorithms:

$$\min_u \Big\{ \text{distance}_s[k][u] + \text{distance}_t[k][u] \Big\}. \tag{D.19}$$

Thus, this algorithm needs *half of the iterations of Bellman-Ford*.

Dataset generation. In the dataset, we sample $200,000$ training data, $6,000$ validation data, and $6,000$ test data. For each model, we report the test accuracy with the hyperparameter setting that achieves the best validation accuracy. In each training sample, the input universe consists of the trainer and 10 monsters $X_0, ..., X_{10}$, and the request level k, *i.e.*, we need to challenge monster k. We have $X_i = [h_1, h_2]$, where $h_1 = i$ indicates the combat level, and the location $h_2 \in [0..10]^2$ is sampled uniformly from $[0..10]^2$. We generate the answer label y for a universe S as follows. We implement a shortest path algorithm to compute the minimum cost from the trainer to monster k, where the cost is defined in task description. Then the label y is a one-hot encoding of minimum cost with 200 classes. Moreover, when we sample the data, we apply rejection sampling to ensure that the minimum cost's shortest path is of length $3, 4, 5, 6, 7$ with equal probability. That is, we eliminate the trivial questions.

Hyperparameter setting. We train all models with the Adam optimizer, with learning rate from $2e-4$ and $5e-4$, and we decay the learning rate by 0.5 every 50 steps. We use cross-entropy loss. We train all models for 300 epochs. We tune batch size of 128 and 64.

For the MLP model, we choose the number of layers from 4 and 8, 16. For other models, we choose the number of hidden layers of MLP modules from 3 and 4. For GNN models, we choose the hidden dimension of MLP modules from 128 and 256. For DeepSet and MLP models, we choose the hidden dimension of MLP modules from 128, 256, 2500. Moreover, dropout with rate 0.5 is applied before the last two hidden layers of MLP$_1$, *i.e.*, the last MLP module in all models.

D.2.4 Subset Sum

Dataset generation. In the dataset, we sample $40,000$ training data, $4,000$ validation data, and $4,000$ test data. For each model, we report the test accuracy with the hyperparameter setting that achieves the best validation accuracy. In each training sample, the input universe S consists of 6 numbers $X_1, ..., X_6$, where each X_i is uniformly sampled from [-200..200]. The goal is to decide if there exists a subset that sums up to 0. In the data generation, we carefully decrease the number of questions that have trivial answers: 1)we control the number of samples where $0 \in \{X_1, ..., X_6\}$ to be around 1% of the total training data; 2) we further control the number of samples where $X_1 + ... + X_6 = 0$ or $\exists i, j \in [1..6]$ so that $X_i = -X_j$ to be around 1.5% of the total training data. In addition, we apply rejection sampling to make sure that the questions with answer yes (aka. such subset exists) and answer no (aka. no such subset exists) are balanced (*i.e.*, 20,000 samples for each class in the training data).

Hyperparameter setting. We train all models with the Adam optimizer, with learning rate from $1e-3$, $5e-4$, and $1e-4$, and we decay the learning rate by 0.5 every 50 steps. We use cross-entropy loss. We train all models for 300 epochs. The batch size we use for all models is 64.

For DeepSets and MLP models, we choose the number of of hidden layers of the MLP modules from 4, 8, 16. For GNN and HRN models, we set the number of hidden layers of the last MLP modules to 4. For DeepSets and MLP, we choose the hidden dimension of MLP modules from 128, 256, 2500, 5000. For GNN and HRN models, we choose the hidden dimension of MLP modules from 128 and 256. Moreover, dropout with rate 0.5 is applied before the last two hidden layers of MLP_1, *i.e.*, the last MLP module in all models.

The model Neural Exhaustive Search (NES) enumerates all possible non-empty subsets τ of S, and passes the numbers of τ to an MLP, in a random order, to obtain the hidden feature. The hidden feature is then passed to a single-direction one-layer LSTM of hidden dimension 128. Afterwards, NES applies an aggregation function to these $2^6 - 1$ hidden states obtained by the LSTM to obtain the final output. For NES, we set the number of hidden layers of the last MLP, *i.e.*, MLP_2, to 4, the number of hidden layers of the MLPs prior to the last MLP, *i.e.*, MLP_1, to 3, and we choose the hidden dimension of all MLP modules from 128 and 256.

Appendix E

How Neural Networks Extrapolate

E.1 Theoretical Background

In this section, we introduce theoretical background on neural tangent kernel (NTK), which draws an equivalence between the training dynamics of infinitely-wide (or ultra-wide) neural networks and that of kernel regression with respect to the neural tangent kernel.

Consider a general neural network $f(\boldsymbol{\theta}, \boldsymbol{x}) : \mathcal{X} \to \mathbb{R}$ where $\boldsymbol{\theta} \in \mathbb{R}^m$ is the parameters in the network and $\boldsymbol{x} \in \mathcal{X}$ is the input. Suppose we train the neural network by minimizing the squared loss over training data, $\ell(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{n}(f(\boldsymbol{\theta}, \boldsymbol{x}_i) - y_i)^2$, by gradient descent with infinitesimally small learning rate, i.e., $\frac{d\boldsymbol{\theta}(t)}{dt} = -\nabla\ell(\boldsymbol{\theta}(t))$. Let $\boldsymbol{u}(t) = (f(\boldsymbol{\theta}(t), \boldsymbol{x}_i))_{i=1}^{n}$ be the network outputs. $\boldsymbol{u}(t)$ follows the dynamics

$$\frac{d\boldsymbol{u}(t)}{dt} = -\boldsymbol{H}(t)(\boldsymbol{u}(t) - y), \tag{E.1}$$

where $\boldsymbol{H}(t)$ is an $n \times n$ matrix whose (i, j)-th entry is

$$\boldsymbol{H}(t)_{ij} = \left\langle \frac{\partial f(\boldsymbol{\theta}(t), \boldsymbol{x}_i)}{\partial \boldsymbol{\theta}}, \frac{\partial f(\boldsymbol{\theta}(t), \boldsymbol{x}_j)}{\partial \boldsymbol{\theta}} \right\rangle. \tag{E.2}$$

A line of works show that for sufficiently wide networks, $\boldsymbol{H}(t)$ stays almost constant during training, i.e., $\boldsymbol{H}(t) = \boldsymbol{H}(0)$ in the limit [Arora et al., 2019b,c, Allen-Zhu et al., 2019a, Du et al., 2019c,a, Li and Liang, 2018, Jacot et al., 2018]. Suppose network parameters are randomly initialized with certain scaling, as network width goes to infinity, $\boldsymbol{H}(0)$ converges to a fixed matrix, the neural tangent kernel (NTK) [Jacot et al., 2018]:

$$\mathrm{NTK}(\boldsymbol{x}, \boldsymbol{x}') = \mathop{\mathbb{E}}_{\boldsymbol{\theta} \sim \mathcal{W}} \left\langle \frac{\partial f(\boldsymbol{\theta}(t), \boldsymbol{x})}{\partial \boldsymbol{\theta}}, \frac{\partial f(\boldsymbol{\theta}(t), \boldsymbol{x}')}{\partial \boldsymbol{\theta}} \right\rangle, \tag{E.3}$$

where $\mathcal{W}$ is Gaussian.

Therefore, the learning dynamics of sufficiently wide neural networks in this regime is equivalent to that of kernel gradient descent with respect to the NTK. This implies the function learned by a neural network at convergence on any specific training set, denoted by $f_{\text{NTK}}(\boldsymbol{x})$, can be precisely characterized, and is equivalent to the following kernel regression solution

$$f_{\text{NTK}}(\boldsymbol{x}) = (\mathrm{NTK}(\boldsymbol{x}, \boldsymbol{x}_1), ..., \mathrm{NTK}(\boldsymbol{x}, \boldsymbol{x}_n)) \cdot \mathrm{NTK}_{\text{train}}^{-1} \boldsymbol{Y}, \tag{E.4}$$

where $\mathrm{NTK}_{\text{train}}$ is the $n \times n$ kernel for training data, $\mathrm{NTK}(\boldsymbol{x}, \boldsymbol{x}_i)$ is the kernel value between test data $\boldsymbol{x}$ and training data $\boldsymbol{x}_i$, and $\boldsymbol{Y}$ is the training labels.

We can in fact exactly calculate the neural tangent kernel matrix for certain architectures and activation functions. The exact formula of NTK with ReLU activation has been derived for feedforward neural networks [Jacot et al., 2018], convolutional neural networks [Arora et al., 2019c], and Graph Neural Networks [Du et al., 2019b].

Our theory builds upon this equivalence of network learning and kernel regression to more precisely characterize the function learned by a sufficiently-wide neural network given any specific training set. In particular, the difference between the learned function and true function over the domain of $\mathcal{X}$ determines the extrapolation error.

However, in general it is non-trivial to compute or analyze the *functional form* of what a neural network learns using (E.4), because the kernel regression solution using neural tangent kernel only gives point-wise evaluation. Thus, we instead analyze the function learned by a network in the NTK's induced *feature space*, because representations in the feature space would give a functional form.

Lemma E.1 makes this connection more precise: the solution to the kernel regression using neural tangent kernel, which also equals over-parameterized network learning, is equivalent to a min-norm solution among functions in the NTK's induced feature space that fits all training data. Here the min-norm refers to the RKHS norm.

Lemma E.1. *Let $\phi(\boldsymbol{x})$ be a feature map induced by a neural tangent kernel, for any $\boldsymbol{x} \in \mathbb{R}^d$. The solution to kernel regression* (E.4) *is equivalent to $f_{NTK}(\boldsymbol{x}) = \phi(\boldsymbol{x})^\top \boldsymbol{\beta}_{NTK}$, where $\boldsymbol{\beta}_{NTK}$ is*

$$\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2$$
$$\text{s.t.} \quad \phi(\boldsymbol{x}_i)^\top \boldsymbol{\beta} = y_i, \quad \text{for } i = 1, ..., n.$$

We prove Lemma E.1 in Appendix E.2.6. To analyze the learned functions as the min-norm solution in feature space, we also need the explicit formula of an induced feature map of the corresponding neural tangent kernel. The following lemma gives a NTK feature space for two-layer MLPs with ReLU activation. It follows easily from the kernel formula described in Jacot et al. [2018], Arora et al. [2019c], Bietti and Mairal [2019].

Lemma E.2. *An infinite-dimensional feature map $\phi(\boldsymbol{x})$ induced by the neural tangent kernel of a two-layer multi-layer perceptron with ReLU activation function is*

$$\phi(\boldsymbol{x}) = c\left(\boldsymbol{x} \cdot \mathbb{I}\left(\boldsymbol{w}^{(k)^\top} \boldsymbol{x} \geq 0\right), \boldsymbol{w}^{(k)^\top} \boldsymbol{x} \cdot \mathbb{I}\left(\boldsymbol{w}^{(k)^\top} \boldsymbol{x} \geq 0\right), ...\right), \tag{E.5}$$

where $\boldsymbol{w}^{(k)} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, with k going to infinity. c is a constant, and $\mathbb{I}$ is the indicator function.

We prove Lemma E.2 in Appendix E.2.7. The feature maps for other architectures, e.g., Graph Neural Networks (GNNs) can be derived similarly. We analyze the Graph Neural Tangent Kernel (GNTK) for a simple GNN architecture in Theorem 8.2.

We then use Lemma E.1 and E.2 to characterize the properties of functions learned by an over-parameterized neural network. We precisely characterize the neural networks' learned functions in the NTK regime via solving the constrained optimization problem corresponding to the min-norm function in NTK feature space with the constraint of fitting the training data.

However, there still remains many technical challenges. For example, provable extrapolation (exact or asymptotic) is often *not* achieved with most training data distribution. Understanding the desirable condition requires significant insights into the geometry properties of training data distribution, and how they interact with the solution learned by neural networks. Our insights and refined analysis shows in $\mathbb{R}^d$ space, we need to consider the directions of training data. In graphs, we need to consider, in addition, the graph structure of training data. We refer readers to detailed proofs for the intuition of data conditions. Moreover, since NTK corresponds to infinitely wide neural networks, the feature space is of infinite dimension. The analysis of infinite dimensional spaces poses non-trivial technical challenges too.

Since different theorems have their respective challenges and insights/techniques, we refer the interested readers to the respective proofs for details. In Lemma 7.2 (proof in Appendix E.2.2), Theorem 7.3 (proof in Appendix E.2.3), and Theorem 7.1 (proof in Appendix E.2.1) we analyze over-parameterized MLPs. The proof of Corollary 8.1 is in Appendix E.2.4. In Theorem 8.2 we analyze Graph Neural Networks (proof in Appendix E.2.5).

E.2 Proofs

E.2.1 Proof of Theorem 7.1

To show neural network outputs $f(\boldsymbol{x})$ converge to a linear function along all directions $\boldsymbol{v}$, we will analyze the function learned by a neural network on the training set $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, by studying the functional representation in the network's neural tangent kernel RKHS space.

Recall from Section E.1 that in the NTK regime, i.e., networks are infinitely wide, randomly initialized, and trained by gradient descent with infinitesimally small learning rate, the learning dynamics of the neural network is equivalent to that of a kernel regression with respect to its neural tangent kernel.

For any $\boldsymbol{x} \in \mathbb{R}^d$, the network output is given by

$$f(\boldsymbol{x}) = \left(\left\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}_1) \right\rangle, ..., \left\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}_n) \right\rangle\right) \cdot \text{NTK}_{\text{train}}^{-1} \boldsymbol{Y},$$

where $\text{NTK}_{\text{train}}$ is the $n \times n$ kernel for training data, $\left\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}_i) \right\rangle$ is the kernel value between test data $\boldsymbol{x}$ and training data $\boldsymbol{x}_i$, and $\boldsymbol{Y}$ is training labels. By Lemma E.1, the kernel regression solution is also equivalent to the min-norm solution in the NTK RKHS space that fits all training data

$$f(\boldsymbol{x}) = \phi(\boldsymbol{x})^\top \boldsymbol{\beta}_{\text{NTK}}, \tag{E.6}$$

where the representation coefficient $\boldsymbol{\beta}_{\text{NTK}}$ is

$$\begin{aligned} &\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2 \\ \text{s.t.} \quad &\phi(\boldsymbol{x}_i)^\top \boldsymbol{\beta} = y_i, \quad \text{for } i = 1, ..., n. \end{aligned}$$

The feature map $\phi(\boldsymbol{x})$ for a two-layer MLP with ReLU activation is given by Lemma E.2

$$\phi\left(\boldsymbol{x}\right) = c' \left(\boldsymbol{x} \cdot \mathbb{I}\left(\boldsymbol{w}^{(k)^\top} \boldsymbol{x} \geq 0\right), \boldsymbol{w}^{(k)^\top} \boldsymbol{x} \cdot \mathbb{I}\left(\boldsymbol{w}^{(k)^\top} \boldsymbol{x} \geq 0\right), ...\right), \tag{E.7}$$

where $\boldsymbol{w}^{(k)} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$, with k going to infinity. c' is a constant, and $\mathbb{I}$ is the indicator function. Without loss of generality, we assume the bias term to be 1. For simplicity of notations, we denote each data $\boldsymbol{x}$ plus bias term by, i.e., $\hat{\boldsymbol{x}} = [\boldsymbol{x}|1]$ [Bietti and Mairal, 2019], and assume constant term is 1.

Given any direction $\boldsymbol{v}$ on the unit sphere, the network outputs for out-of-distribution data $\boldsymbol{x}_0 = t\boldsymbol{v}$ and $\boldsymbol{x} = \boldsymbol{x}_0 + h\boldsymbol{v} = (1 + \lambda)\boldsymbol{x}_0$, where we introduce the notation of $\boldsymbol{x}$ and λ for convenience, are given by (E.6) and (E.7)

$$\begin{aligned} f(\hat{\boldsymbol{x}}_0) =& \boldsymbol{\beta}_{\text{NTK}}^\top \left(\hat{\boldsymbol{x}}_0 \cdot \mathbb{I}\left(\boldsymbol{w}^{(k)^\top} \hat{\boldsymbol{x}}_0 \geq 0\right), \boldsymbol{w}^{(k)^\top} \hat{\boldsymbol{x}}_0 \cdot \mathbb{I}\left(\boldsymbol{w}^{(k)^\top} \hat{\boldsymbol{x}}_0 \geq 0\right), ...\right), \\ f(\hat{\boldsymbol{x}}) =& \boldsymbol{\beta}_{\text{NTK}}^\top \left(\hat{\boldsymbol{x}} \cdot \mathbb{I}\left(\boldsymbol{w}^{(k)^\top} \hat{\boldsymbol{x}} \geq 0\right), \boldsymbol{w}^{(k)^\top} \hat{\boldsymbol{x}} \cdot \mathbb{I}\left(\boldsymbol{w}^{(k)^\top} \hat{\boldsymbol{x}} \geq 0\right), ...\right), \end{aligned}$$

where we have $\hat{\boldsymbol{x}}_0 = [\boldsymbol{x}_0|1]$ and $\hat{\boldsymbol{x}} = [(1+\lambda)\boldsymbol{x}_0|1]$. It follows that

$$f(\hat{\boldsymbol{x}}) - f(\hat{\boldsymbol{x}}_0) = \boldsymbol{\beta}_{\text{NTK}}^\top \Big(\hat{\boldsymbol{x}} \cdot \mathbb{I}\left(\boldsymbol{w}^{(k)^\top} \hat{\boldsymbol{x}} \geq 0\right) - \hat{\boldsymbol{x}}_0 \cdot \mathbb{I}\left(\boldsymbol{w}^{(k)^\top} \hat{\boldsymbol{x}}_0 \geq 0\right), \tag{E.8}$$

$$\boldsymbol{w}^{(k)^\top} \hat{\boldsymbol{x}} \cdot \mathbb{I}\left(\boldsymbol{w}^{(k)^\top} \hat{\boldsymbol{x}} \geq 0\right) - \boldsymbol{w}^{(k)^\top} \hat{\boldsymbol{x}}_0 \cdot \mathbb{I}\left(\boldsymbol{w}^{(k)^\top} \hat{\boldsymbol{x}}_0 \geq 0\right), \ldots \Big) \tag{E.9}$$

By re-arranging the terms, we get the following equivalent form of the entries:

$$\begin{aligned}
& \hat{\boldsymbol{x}} \cdot \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}} \geq 0\right) - \hat{\boldsymbol{x}}_0 \cdot \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right) && \text{(E.10)} \\
= \; & \hat{\boldsymbol{x}} \cdot \left(\mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}} \geq 0\right) - \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right) + \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right)\right) - \hat{\boldsymbol{x}}_0 \cdot \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right) && \text{(E.11)} \\
= \; & \hat{\boldsymbol{x}} \cdot \left(\mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}} \geq 0\right) - \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right)\right) + (\hat{\boldsymbol{x}} - \hat{\boldsymbol{x}}_0) \cdot \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right) && \text{(E.12)} \\
= \; & [\boldsymbol{x}|1] \cdot \left(\mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}} \geq 0\right) - \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right)\right) + [h\boldsymbol{v}|0] \cdot \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right) && \text{(E.13)}
\end{aligned}$$

Similarly, we have

$$\begin{aligned}
& \boldsymbol{w}^\top \hat{\boldsymbol{x}} \cdot \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}} \geq 0\right) - \boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \cdot \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right) && \text{(E.14)} \\
= \; & \boldsymbol{w}^\top \hat{\boldsymbol{x}} \cdot \left(\mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}} \geq 0\right) - \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right) + \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right)\right) - \boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \cdot \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right) && \text{(E.15)} \\
= \; & \boldsymbol{w}^\top \hat{\boldsymbol{x}} \cdot \left(\mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}} \geq 0\right) - \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right)\right) + \boldsymbol{w}^\top \left(\hat{\boldsymbol{x}} - \hat{\boldsymbol{x}}_0\right) \cdot \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right) && \text{(E.16)} \\
= \; & \boldsymbol{w}^\top [\boldsymbol{x}|1] \cdot \left(\mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}} \geq 0\right) - \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right)\right) + \boldsymbol{w}^\top [h\boldsymbol{v}|0] \cdot \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right) && \text{(E.17)}
\end{aligned}$$

Again, let us denote the part of $\boldsymbol{\beta}_{\text{NTK}}$ corresponding to each $\boldsymbol{w}$ by $\boldsymbol{\beta}_{\boldsymbol{w}}$. Moreover, let us denote the part corresponding to (E.13) by $\boldsymbol{\beta}_{\boldsymbol{w}}^1$ and the part corresponding to (E.17) by $\boldsymbol{\beta}_{\boldsymbol{w}}^2$.

Then we have

$$\frac{f(\hat{\boldsymbol{x}}) - f(\hat{\boldsymbol{x}}_0)}{h} \tag{E.18}$$

$$= \int \boldsymbol{\beta}_{\boldsymbol{w}}^{1^\top} [\boldsymbol{x}/h|1/h] \cdot \left(\mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}} \geq 0\right) - \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right)\right) \mathrm{d}\mathbb{P}(\boldsymbol{w}) \tag{E.19}$$

$$+ \int \boldsymbol{\beta}_{\boldsymbol{w}}^{1^\top} [\boldsymbol{v}|0] \cdot \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right) \mathrm{d}\mathbb{P}(\boldsymbol{w}) \tag{E.20}$$

$$+ \int \boldsymbol{\beta}_{\boldsymbol{w}}^{2} \cdot \boldsymbol{w}^\top [\boldsymbol{x}/h|1/h] \cdot \left(\mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}} \geq 0\right) - \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right)\right) \mathrm{d}\mathbb{P}(\boldsymbol{w}) \tag{E.21}$$

$$+ \int \boldsymbol{\beta}_{\boldsymbol{w}}^{2} \cdot \boldsymbol{w}^\top [\boldsymbol{v}|0] \cdot \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right) \mathrm{d}\mathbb{P}(\boldsymbol{w}) \tag{E.22}$$

Note that all $\boldsymbol{\beta}_w$ are finite constants that depend on the training data. Next, we show that as $t \to \infty$, each of the terms above converges in $O(1/\epsilon)$ to some constant coefficient $\boldsymbol{\beta}_v$ that depend on the training data and the direction $\boldsymbol{v}$. Let us first consider (E.20). We have

$$\int \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right) d\mathbb{P}(\boldsymbol{w}) = \int \mathbb{I}\left(\boldsymbol{w}^\top [\boldsymbol{x}_0|1] \geq 0\right) \mathrm{d}\mathbb{P}(\boldsymbol{w}) \tag{E.23}$$

$$= \int \mathbb{I}\left(\boldsymbol{w}^\top [\boldsymbol{x}_0/t|1/t] \geq 0\right) \mathrm{d}\mathbb{P}(\boldsymbol{w}) \tag{E.24}$$

$$\to \int \mathbb{I}\left(\boldsymbol{w}^\top [\boldsymbol{v}|0] \geq 0\right) \mathrm{d}\mathbb{P}(\boldsymbol{w}) \qquad \text{as } t \to \infty \tag{E.25}$$

Because $\boldsymbol{\beta}_w^1$ are finite constants, it follows that

$$\int \boldsymbol{\beta}_{\boldsymbol{w}}^{1^\top} [\boldsymbol{v}|0] \cdot \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right) \mathrm{d}\mathbb{P}(\boldsymbol{w}) \to \int \boldsymbol{\beta}_{\boldsymbol{w}}^{1^\top} [\boldsymbol{v}|0] \cdot \mathbb{I}\left(\boldsymbol{w}^\top [\boldsymbol{v}|0] \geq 0\right) \mathrm{d}\mathbb{P}(\boldsymbol{w}), \tag{E.26}$$

where the right hand side is a constant that depends on training data and direction $\boldsymbol{v}$. Next, we show the convergence rate for (E.26). Given error $\epsilon > 0$, because $\boldsymbol{\beta}_{\boldsymbol{w}}^{1^\top} [\boldsymbol{v}|0]$ are finite constants, we need to bound the following by $C \cdot \epsilon$ for some constant C,

$$|\int \mathbb{I}\left(\boldsymbol{w}^\top \hat{\boldsymbol{x}}_0 \geq 0\right) - \mathbb{I}\left(\boldsymbol{w}^\top [\boldsymbol{v}|0] \geq 0\right) \mathrm{d}\mathbb{P}(\boldsymbol{w})| \tag{E.27}$$

$$= |\int \mathbb{I}\left(\boldsymbol{w}^\top [\boldsymbol{x}_0|1] \geq 0\right) - \mathbb{I}\left(\boldsymbol{w}^\top [\boldsymbol{x}_0|0] \geq 0\right) \mathrm{d}\mathbb{P}(\boldsymbol{w})| \tag{E.28}$$

Observe that the two terms in (E.28) represent the volume of half-(balls) that are orthogonal to vectors $[\boldsymbol{x}_0|1]$ and $[\boldsymbol{x}_0|0]$. Hence, (E.28) is the volume of the non-overlapping part of the two (half)balls, which is created by rotating an angle θ along the last coordinate. By symmetry, (E.28) is linear in θ. Moreover, the angle $\theta = \arctan(C/t)$ for some constant C. Hence, it follows that

$$|\int \mathbb{I}\left(\boldsymbol{w}^\top[\boldsymbol{x}_0|1] \geq 0\right) - \mathbb{I}\left(\boldsymbol{w}^\top[\boldsymbol{x}_0|0] \geq 0\right) \mathrm{d}\mathbb{P}(\boldsymbol{w})| = C_1 \cdot \arctan(C_2/t) \tag{E.29}$$

$$\leq C_1 \cdot C_2/t \tag{E.30}$$

$$= O(1/t) \tag{E.31}$$

In the last inequality, we used the fact that $\arctan x < x$ for $x > 0$. Hence, $O(1/t) < \epsilon$ implies $t = O(1/\epsilon)$ as desired. Next, we consider (E.19).

$$\int \boldsymbol{\beta}_{\boldsymbol{w}}^{1^\top}[\boldsymbol{x}/h|1/h] \cdot \left(\mathbb{I}\left(\boldsymbol{w}^\top\hat{\boldsymbol{x}} \geq 0\right) - \mathbb{I}\left(\boldsymbol{w}^\top\hat{\boldsymbol{x}}_0 \geq 0\right)\right) \mathrm{d}\mathbb{P}(\boldsymbol{w}) \tag{E.32}$$

Let us first analyze the convergence of the following:

$$|\int \mathbb{I}\left(\boldsymbol{w}^\top\hat{\boldsymbol{x}} \geq 0\right) - \mathbb{I}\left(\boldsymbol{w}^\top\hat{\boldsymbol{x}}_0 \geq 0\right) \mathrm{d}\mathbb{P}(\boldsymbol{w})| \tag{E.33}$$

$$= \quad |\int \mathbb{I}\left(\boldsymbol{w}^\top[(1+\lambda)\boldsymbol{x}_0|1] \geq 0\right) - \mathbb{I}\left(\boldsymbol{w}^\top[\boldsymbol{x}_0|1] \geq 0\right) \mathrm{d}\mathbb{P}(\boldsymbol{w})\mathrm{d}\mathbb{P}(\boldsymbol{w})| \tag{E.34}$$

$$= \quad |\int \mathbb{I}\left(\boldsymbol{w}^\top[\boldsymbol{x}_0|\frac{1}{1+\lambda}] \geq 0\right) - \mathbb{I}\left(\boldsymbol{w}^\top[\boldsymbol{x}_0|1] \geq 0\right) \mathrm{d}\mathbb{P}(\boldsymbol{w})\mathrm{d}\mathbb{P}(\boldsymbol{w})| \to 0 \tag{E.35}$$

The convergence to 0 follows from (E.29). Now we consider the convergence rate. The angle θ is at most $1 - \frac{1}{1+\lambda}$ times of that in (E.29). Hence, the rate is as follows

$$\left(1 - \frac{1}{1+\lambda}\right) \cdot O\left(\frac{1}{t}\right) = \frac{\lambda}{1+\lambda} \cdot O\left(\frac{1}{t}\right) = \frac{h/t}{1+h/t} \cdot O\left(\frac{1}{t}\right) = O\left(\frac{h}{(h+t)t}\right) \tag{E.36}$$

Now we get back to (E.19), which simplifies as the following.

$$\int \boldsymbol{\beta}_{\boldsymbol{w}}^{1^\top}\left[\boldsymbol{v} + \frac{t\boldsymbol{v}}{h}|\frac{1}{h}\right] \cdot \left(\mathbb{I}\left(\boldsymbol{w}^\top\hat{\boldsymbol{x}} \geq 0\right) - \mathbb{I}\left(\boldsymbol{w}^\top\hat{\boldsymbol{x}}_0 \geq 0\right)\right) \mathrm{d}\mathbb{P}(\boldsymbol{w}) \tag{E.37}$$

We compare the rate of growth of left hand side and the rate of decrease of right hand side (indicators).

$$\frac{t}{h} \cdot \frac{h}{(h+t)t} = \frac{1}{h+t} \to 0 \quad \text{as } t \to \infty \tag{E.38}$$

$$\frac{1}{h} \cdot \frac{h}{(h+t)t} = \frac{1}{(h+t)t} \to 0 \quad \text{as } t \to \infty \tag{E.39}$$

Hence, the indicators decrease faster, and it follows that (E.19) converges to 0 with rate $O(\frac{1}{\epsilon})$. Moreover, we can bound $\boldsymbol{w}$ with standard concentration techniques. Then the proofs for (E.21) and (E.22) follow similarly. This completes the proof.

E.2.2 Proof of Lemma 7.2

Overview of proof. To prove exact extrapolation given the conditions on training data, we analyze the function learned by the neural network in a functional form. The network's learned function can be precisely characterized by a solution in the network's neural tangent kernel feature space which has a minimum RKHS norm among functions that can fit all training data, i.e., it corresponds to the optimum of a constrained optimization problem. We show that the global optimum of this constrained optimization problem, given the conditions on training data, is precisely the same function as the underlying true function.

Setup and preparation. Let $\boldsymbol{X} = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\}$ and $\boldsymbol{Y} = \{y_1, ..., y_n\}$ denote the training set input features and their labels. Let $\boldsymbol{\beta}_g \in \mathbb{R}^d$ denote the true parameters/weights for the underlying linear function g, i.e.,

$$g(\boldsymbol{x}) = \boldsymbol{\beta}_g^\top \boldsymbol{x} \quad \text{for all } \boldsymbol{x} \in \mathbb{R}^d$$

Recall from Section E.1 that in the NTK regime, where networks are infinitely wide, randomly initialized, and trained by gradient descent with infinitesimally small learning rate, the learning dynamics of a neural network is equivalent to that of a kernel regression with respect to its neural tangent kernel. Moreover, Lemma E.1 tells us that this kernel regression solution can be expressed in the functional form in the neural tangent kernel's feature space. That is, the function learned by the neural network (in the ntk regime) can be precisely characterized as

$$f(\boldsymbol{x}) = \phi(\boldsymbol{x})^\top \boldsymbol{\beta}_{\text{NTK}},$$

where the representation coefficient $\boldsymbol{\beta}_{\text{NTK}}$ is

$$\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2 \tag{E.40}$$

$$\text{s.t.} \quad \phi(\boldsymbol{x}_i)^\top \boldsymbol{\beta} = y_i, \quad \text{for } i = 1, ..., n. \tag{E.41}$$

An infinite-dimensional feature map $\phi(\boldsymbol{x})$ for a two-layer ReLU network is described in Lemma E.2

$$\phi\left(\boldsymbol{x}\right) = c'\left(\boldsymbol{x} \cdot \mathbb{I}\left(\boldsymbol{w}^{(k)\top}\boldsymbol{x} \geq 0\right), \boldsymbol{w}^{(k)\top}\boldsymbol{x} \cdot \mathbb{I}\left(\boldsymbol{w}^{(k)\top}\boldsymbol{x} \geq 0\right), ...\right),$$

where $\boldsymbol{w}^{(k)} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, with k going to infinity. c' is a constant, and $\mathbb{I}$ is the indicator function. That is, there are infinitely many directions $\boldsymbol{w}$ with Gaussian density, and each direction comes with two features. Without loss of generality, we can assume the scaling constant to be 1.

Constrained optimization in NTK feature space. The representation or weight of the neural network's learned function in the neural tangent kernel feature space, $\boldsymbol{\beta}_{\text{NTK}}$, consists of weight vectors for each $\boldsymbol{x} \cdot \mathbb{I}\left(\boldsymbol{w}^{(k)\top}\boldsymbol{x} \geq 0\right) \in \mathbb{R}^d$ and $\boldsymbol{w}^{(k)\top}\boldsymbol{x} \cdot \mathbb{I}\left(\boldsymbol{w}^{(k)\top}\boldsymbol{x} \geq 0\right) \in \mathbb{R}$. For simplicity of notation, we will use $\boldsymbol{w}$ to refer to a particular $\boldsymbol{w}$, without considering the index (k), which does not matter for our purposes. For any $\boldsymbol{w} \in \mathbb{R}^d$, we denote by $\hat{\boldsymbol{\beta}}_{\boldsymbol{w}} = (\hat{\boldsymbol{\beta}}_{\boldsymbol{w}}^{(1)}, ..., \hat{\boldsymbol{\beta}}_{\boldsymbol{w}}^{(d)}) \in \mathbb{R}^d$ the weight vectors corresponding to $\boldsymbol{x} \cdot \mathbb{I}\left(\boldsymbol{w}^\top\boldsymbol{x} \geq 0\right)$, and denote by $\hat{\boldsymbol{\beta}}'_{\boldsymbol{w}} \in \mathbb{R}^d$ the weight for $\boldsymbol{w}^\top\boldsymbol{x} \cdot \mathbb{I}\left(\boldsymbol{w}^\top\boldsymbol{x} \geq 0\right)$.

Observe that for any $\boldsymbol{w} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}) \in \mathbb{R}^d$, any other vectors in the same direction will activate the same set of $\boldsymbol{x}_i \in \mathbb{R}^d$. That is, if $\boldsymbol{w}^\top\boldsymbol{x}_i \geq 0$ for any $\boldsymbol{w} \in \mathbb{R}^d$, then $(k \cdot \boldsymbol{w})^\top\boldsymbol{x}_i \geq 0$ for any $k > 0$. Hence, we can reload our notation to combine the effect of weights for $\boldsymbol{w}$'s in the same direction. This enables simpler notations and allows us to change the distribution of $\boldsymbol{w}$ in NTK features from Gaussian distribution to uniform distribution on the unit sphere.

More precisely, we reload our notation by using $\boldsymbol{\beta}_{\boldsymbol{w}}$ and $\boldsymbol{\beta}'_{\boldsymbol{w}}$ to denote the combined effect of all weights $(\hat{\boldsymbol{\beta}}^{(1)}_{k\boldsymbol{w}}, ..., \hat{\boldsymbol{\beta}}^{(d)}_{k\boldsymbol{w}}) \in \mathbb{R}^d$ and $\hat{\boldsymbol{\beta}}'_{k\boldsymbol{w}} \in \mathbb{R}$ for all $k\boldsymbol{w}$ with $k > 0$ in the same direction of $\boldsymbol{w}$. That is, for each $\boldsymbol{w} \sim$ Uni(unit sphere) $\in \mathbb{R}^d$, we define $\boldsymbol{\beta}^{(j)}_{\boldsymbol{w}}$ as the total effect of weights in the same direction

$$\boldsymbol{\beta}^{(j)}_{\boldsymbol{w}} = \int \hat{\boldsymbol{\beta}}^{(j)}_{\boldsymbol{u}} \mathbb{I}\left(\frac{\boldsymbol{w}^\top \boldsymbol{u}}{\|\boldsymbol{w}\| \cdot \|\boldsymbol{u}\|} = 1\right) \mathrm{d}\mathbb{P}(\boldsymbol{u}), \quad \text{for } j = [d] \tag{E.42}$$

where $\boldsymbol{u} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$. Note that to ensure the $\boldsymbol{\beta}_{\boldsymbol{w}}$ is a well-defined number, here we can work with the polar representation and integrate with respect to an angle. Then $\boldsymbol{\beta}_{\boldsymbol{w}}$ is well-defined. But for simplicity of exposition, we use the plain notation of integral. Similarly, we define $\boldsymbol{\beta}'_{\boldsymbol{w}}$ as reloading the notation of

$$\boldsymbol{\beta}'_{\boldsymbol{w}} = \int \hat{\boldsymbol{\beta}}_{\boldsymbol{u}} \mathbb{I}\left(\frac{\boldsymbol{w}^\top \boldsymbol{u}}{\|\boldsymbol{w}\| \cdot \|\boldsymbol{u}\|} = 1\right) \cdot \frac{\|\boldsymbol{u}\|}{\|\boldsymbol{w}\|} \mathrm{d}\mathbb{P}(\boldsymbol{u}) \tag{E.43}$$

Here, in (E.43) we have an extra term of $\frac{\|\boldsymbol{u}\|}{\|\boldsymbol{w}\|}$ compared to (E.42) because the NTK features that (E.43) corresponds to, $\boldsymbol{w}^\top \boldsymbol{x} \cdot \mathbb{I}\left(\boldsymbol{w}^\top \boldsymbol{x} \geq 0\right)$, has an extra $\boldsymbol{w}^\top$ term. So we need to take into account the scaling. This abstraction enables us to make claims on the high-level parameters $\boldsymbol{\beta}_{\boldsymbol{w}}$ and $\boldsymbol{\beta}'_{\boldsymbol{w}}$ only, which we will show to be sufficient to determine the learned function.

Then we can formulate the constrained optimization problem whose solution gives a functional form of the neural network's learned function. We rewrite the min-norm solution in (E.40) as

$$\min_{\boldsymbol{\beta}} \int \left(\boldsymbol{\beta}^{(1)}_{\boldsymbol{w}}\right)^2 + \left(\boldsymbol{\beta}^{(2)}_{\boldsymbol{w}}\right)^2 + ... + \left(\boldsymbol{\beta}^{(d)}_{\boldsymbol{w}}\right)^2 + (\boldsymbol{\beta}'_{\boldsymbol{w}})^2 \,\mathrm{d}\mathbb{P}(\boldsymbol{w}) \tag{E.44}$$

$$\text{s.t.} \quad \int_{\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0} \boldsymbol{\beta}^\top_{\boldsymbol{w}} \boldsymbol{x}_i + \boldsymbol{\beta}'_{\boldsymbol{w}} \cdot \boldsymbol{w}^\top \boldsymbol{x}_i \;\, \mathrm{d}\mathbb{P}(\boldsymbol{w}) = \boldsymbol{\beta}^\top_g \boldsymbol{x}_i \quad \forall i \in [n], \tag{E.45}$$

where the density of $\boldsymbol{w}$ is now uniform on the unit sphere of $\mathbb{R}^d$. Observe that since $\boldsymbol{w}$ is from a uniform distribution, the probability density function $\mathbb{P}(\boldsymbol{w})$ is a constant. This means every $\boldsymbol{x}_i$ is activated by half of the $\boldsymbol{w}$ on the unit sphere, which implies we can now write the right hand side of (E.45) in the form of left hand side, i.e., integral form. This allows us to further simplify (E.45) as

$$\int\limits_{\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0} \left(\boldsymbol{\beta}_{\boldsymbol{w}}^\top + \boldsymbol{\beta}'_{\boldsymbol{w}} \cdot \boldsymbol{w}^\top - 2 \cdot \boldsymbol{\beta}_g^\top\right) \boldsymbol{x}_i \ \mathrm{d}\mathbb{P}(\boldsymbol{w}) = 0 \quad \forall i \in [n], \tag{E.46}$$

where (E.46) follows from the following steps of simplification

$$\begin{aligned}
&\int\limits_{\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0} \boldsymbol{\beta}_{\boldsymbol{w}}^{(1)} \boldsymbol{x}_i^{(1)} + ..\boldsymbol{\beta}_{\boldsymbol{w}}^{(d)} \boldsymbol{x}_i^{(d)} + \boldsymbol{\beta}'_{\boldsymbol{w}} \cdot \boldsymbol{w}^\top \boldsymbol{x}_i \mathrm{d}\mathbb{P}(\boldsymbol{w}) = \boldsymbol{\beta}_g^{(1)} \boldsymbol{x}_i^{(1)} + ...\boldsymbol{\beta}_g^{(d)} \boldsymbol{x}_i^{(d)} \quad \forall i \in [n], \\
\Longleftrightarrow &\int\limits_{\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0} \boldsymbol{\beta}_{\boldsymbol{w}}^{(1)} \boldsymbol{x}_i^{(1)} + ... + \boldsymbol{\beta}_{\boldsymbol{w}}^{(d)} \boldsymbol{x}_i^{(d)} + \boldsymbol{\beta}'_{\boldsymbol{w}} \cdot \boldsymbol{w}^\top \boldsymbol{x}_i \ \mathrm{d}\mathbb{P}(\boldsymbol{w}) \\
&= \frac{1}{\int\limits_{\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0} \mathrm{d}\mathbb{P}(\boldsymbol{w})} \cdot \int\limits_{\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0} \mathrm{d}\mathbb{P}(\boldsymbol{w}) \cdot \left(\boldsymbol{\beta}_g^{(1)} \boldsymbol{x}_i^{(1)} + ... + \boldsymbol{\beta}_g^{(d)} \boldsymbol{x}_i^{(d)}\right) \quad \forall i \in [n], \\
\Longleftrightarrow &\int\limits_{\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0} \boldsymbol{\beta}_{\boldsymbol{w}}^{(1)} \boldsymbol{x}_i^{(1)} + ... + \boldsymbol{\beta}_{\boldsymbol{w}}^{(d)} \boldsymbol{x}_i^{(d)} + \boldsymbol{\beta}'_{\boldsymbol{w}} \cdot \boldsymbol{w}^\top \boldsymbol{x}_i \mathrm{d}\mathbb{P}(\boldsymbol{w}) \\
&= 2 \cdot \int\limits_{\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0} \boldsymbol{\beta}_g^{(1)} \boldsymbol{x}_i^{(1)} + ... + \boldsymbol{\beta}_g^{(d)} \boldsymbol{x}_i^{(d)} \mathrm{d}\mathbb{P}(\boldsymbol{w}) \quad \forall i \in [n], \\
\Longleftrightarrow &\int\limits_{\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0} \left(\boldsymbol{\beta}_{\boldsymbol{w}}^\top + \boldsymbol{\beta}'_{\boldsymbol{w}} \cdot \boldsymbol{w}^\top - 2 \cdot \boldsymbol{\beta}_g^\top\right) \boldsymbol{x}_i \ \mathrm{d}\mathbb{P}(\boldsymbol{w}) = 0 \quad \forall i \in [n].
\end{aligned}$$

Claim E.3. *Without loss of generality, assume the scaling factor c in NTK feature map $\phi(\boldsymbol{x})$ is 1. Then the global optimum to the constraint optimization problem* (E.44) *subject to* (E.46)*, i.e.,*

$$\min_{\boldsymbol{\beta}} \int \left(\boldsymbol{\beta}_{\boldsymbol{w}}^{(1)}\right)^2 + \left(\boldsymbol{\beta}_{\boldsymbol{w}}^{(2)}\right)^2 + ... + \left(\boldsymbol{\beta}_{\boldsymbol{w}}^{(d)}\right)^2 + \left(\boldsymbol{\beta}'_{\boldsymbol{w}}\right)^2 \mathrm{d}\mathbb{P}(\boldsymbol{w}) \tag{E.47}$$

$$\text{s.t.} \quad \int\limits_{\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0} \left(\boldsymbol{\beta}_{\boldsymbol{w}}^\top + \boldsymbol{\beta}'_{\boldsymbol{w}} \cdot \boldsymbol{w}^\top - 2 \cdot \boldsymbol{\beta}_g^\top\right) \boldsymbol{x}_i \ \mathrm{d}\mathbb{P}(\boldsymbol{w}) = 0 \quad \forall i \in [n]. \tag{E.48}$$

satisfies $\boldsymbol{\beta}_{\boldsymbol{w}} + \boldsymbol{\beta}'_{\boldsymbol{w}} \cdot \boldsymbol{w} = 2\boldsymbol{\beta}_g$ for all $\boldsymbol{w}$.

This claim implies the exact extrapolation we want to prove, i.e., $f_{\text{NTK}}(\boldsymbol{x}) = g(\boldsymbol{x})$. This is because, if our claim holds, then for any $\boldsymbol{x} \in \mathbb{R}^d$

$$\begin{aligned} f_{\text{NTK}}(\boldsymbol{x}) &= \int_{\boldsymbol{w}^\top \boldsymbol{x} \geq 0} \boldsymbol{\beta}_w^\top \boldsymbol{x} + \boldsymbol{\beta}'_{\boldsymbol{w}} \cdot \boldsymbol{w}^\top \boldsymbol{x} \ \ \mathrm{d}\mathbb{P}(\boldsymbol{w}) \\ &= \int_{\boldsymbol{w}^\top \boldsymbol{x} \geq 0} 2 \cdot \boldsymbol{\beta}_g^\top \boldsymbol{x} \ \ \mathrm{d}\mathbb{P}(\boldsymbol{w}) \\ &= \int_{\boldsymbol{w}^\top \boldsymbol{x} \geq 0} \mathrm{d}\mathbb{P}(\boldsymbol{w}) \cdot 2\boldsymbol{\beta}_g^\top \boldsymbol{x} \\ &= \frac{1}{2} \cdot 2\boldsymbol{\beta}_g^\top \boldsymbol{x} = g(\boldsymbol{x}) \end{aligned}$$

Thus, it remains to prove Claim E.3. To compute the optimum to the constrained optimization problem (E.47), we consider the Lagrange multipliers. It is clear that the objective (E.47) is convex. Moreover, the constraint (E.48) is affine. Hence, by KKT, solution that satisfies the Lagrange condition will be the global optimum. We compute the Lagrange multiplier as

$$\mathcal{L}(\boldsymbol{\beta}, \lambda) = \int \left(\boldsymbol{\beta}_{\boldsymbol{w}}^{(1)}\right)^2 + \left(\boldsymbol{\beta}_{\boldsymbol{w}}^{(2)}\right)^2 + ... + \left(\boldsymbol{\beta}_{\boldsymbol{w}}^{(d)}\right)^2 + (\boldsymbol{\beta}'_{\boldsymbol{w}})^2 \, \mathrm{d}\mathbb{P}(\boldsymbol{w}) \tag{E.49}$$

$$- \sum_{i=1}^n \lambda_i \cdot \left(\int\limits_{\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0} \left(\boldsymbol{\beta}_{\boldsymbol{w}}^\top + \boldsymbol{\beta}'_{\boldsymbol{w}} \cdot \boldsymbol{w}^\top - 2 \cdot \boldsymbol{\beta}_g^\top\right) \boldsymbol{x}_i \ \ \mathrm{d}\mathbb{P}(\boldsymbol{w}) \right) \tag{E.50}$$

Setting the partial derivative of $\mathcal{L}(\boldsymbol{\beta}, \lambda)$ with respect to each variable to zero gives

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\beta}_{\boldsymbol{w}}^{(k)}} = 2\boldsymbol{\beta}_{\boldsymbol{w}}^{(k)} \mathbb{P}(\boldsymbol{w}) + \sum_{i=1}^n \lambda_i \cdot \boldsymbol{x}_i^{(k)} \cdot \mathbb{I}\left(\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0\right) = 0 \tag{E.51}$$

$$\frac{\partial \mathcal{L}}{\boldsymbol{\beta}'_{\boldsymbol{w}}} = 2\boldsymbol{\beta}'_{\boldsymbol{w}} \mathbb{P}(\boldsymbol{w}) + \sum_{i=1}^n \lambda_i \cdot \boldsymbol{w}^\top \boldsymbol{x}_i \cdot \mathbb{I}\left(\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0\right) = 0 \tag{E.52}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_i} = \int\limits_{\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0} \left(\boldsymbol{\beta}_{\boldsymbol{w}}^\top + \boldsymbol{\beta}'_{\boldsymbol{w}} \cdot \boldsymbol{w}^\top - 2 \cdot \boldsymbol{\beta}_g^\top\right) \boldsymbol{x}_i \ \ \mathrm{d}\mathbb{P}(\boldsymbol{w}) = 0 \tag{E.53}$$

It is clear that the solution in Claim E.3 immediately satisfies (E.53). Hence, it remains to show there exist a set of λ_i for $i \in [n]$ that satisfies (E.51) and (E.52). We can simplify (E.51) as

$$\boldsymbol{\beta}_{\boldsymbol{w}}^{(k)} = c \cdot \sum_{i=1}^n \lambda_i \cdot \boldsymbol{x}_i^{(k)} \cdot \mathbb{I}\left(\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0\right), \tag{E.54}$$

where c is a constant. Similarly, we can simplify (E.52) as

$$\boldsymbol{\beta}'_{\boldsymbol{w}} = c \cdot \sum_{i=1}^{n} \lambda_i \cdot \boldsymbol{w}^\top \boldsymbol{x}_i \cdot \mathbb{I}\left(\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0\right) \tag{E.55}$$

Observe that combining (E.54) and (E.55) implies that the constraint (E.55) can be further simplified as

$$\boldsymbol{\beta}'_{\boldsymbol{w}} = \boldsymbol{w}^\top \boldsymbol{\beta}_{\boldsymbol{w}} \tag{E.56}$$

It remains to show that given the condition on training data, there exists a set of λ_i so that (E.54) and (E.56) are satisfied.

Global optimum via the geometry of training data. Recall that we assume our training data $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ satisfies for any $\boldsymbol{w} \in \mathbb{R}^d$, there exist d linearly independent $\{\boldsymbol{x}_i^{\boldsymbol{w}}\}_{i=1}^d \subset \boldsymbol{X}$, where $\boldsymbol{X} = \{\boldsymbol{x}_i\}_{i=1}^n$, so that $\boldsymbol{w}^\top \boldsymbol{x}_i^{\boldsymbol{w}} \geq 0$ and $-\boldsymbol{x}_i^{\boldsymbol{w}} \in \boldsymbol{X}$ for $i = 1..d$, e.g., an orthogonal basis of $\mathbb{R}^d$ and their opposite vectors. We will show that under this data regime, we have

(a) for any particular $\boldsymbol{w}$, there indeed exist a set of λ_i that can satisfy the constraints (E.54) and (E.56) for this particular $\boldsymbol{w}$.

(b) For any $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ that activate the exact same set of $\{\boldsymbol{x}_i\}$, the same set of λ_i can satisfy the constraints (E.54) and (E.56) of both $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$.

(c) Whenever we rotate a $\boldsymbol{w}_1$ to a $\boldsymbol{w}_2$ so that the set of $\boldsymbol{x}_i$ being activated changed, we can still find λ_i that satisfy constraint of both $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$.

Combining (a), (b) and (c) implies there exists a set of λ that satisfy the constraints for all w. Hence, it remains to show these three claims.

We first prove Claim (a). For each $\boldsymbol{w}$, we must find a set of λ_i so that the following hold.

$$\begin{aligned}
&\boldsymbol{\beta}_{\boldsymbol{w}}^{(k)} = c \cdot \sum_{i=1}^{n} \lambda_i \cdot \boldsymbol{x}_i^{(k)} \cdot \mathbb{I}\left(\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0\right), \\
&\boldsymbol{\beta}'_{\boldsymbol{w}} = \boldsymbol{w}^\top \boldsymbol{\beta}_{\boldsymbol{w}} \\
&\boldsymbol{\beta}_{\boldsymbol{w}} + \boldsymbol{\beta}'_{\boldsymbol{w}} \cdot \boldsymbol{w} = 2\boldsymbol{\beta}_g
\end{aligned}$$

Here, $\boldsymbol{\beta}_g$ and $\boldsymbol{w}$ are fixed, and $\boldsymbol{w}$ is a vector on the unit sphere. It is easy to see that $\boldsymbol{\beta}_{\boldsymbol{w}}$ is then determined by $\boldsymbol{\beta}_g$ and $\boldsymbol{w}$, and there indeed exists a solution (solving a consistent linear system). Hence we are left with a linear system with d linear equations

$$\boldsymbol{\beta}_{\boldsymbol{w}}^{(k)} = c \cdot \sum_{i=1}^{n} \lambda_i \cdot \boldsymbol{x}_i^{(k)} \cdot \mathbb{I}\left(\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0\right) \quad \forall k \in [d]$$

to solve with free variables being λ_i so that $\boldsymbol{w}$ activates $\boldsymbol{x}_i$, i.e., $\boldsymbol{w}^\top \boldsymbol{x}_i \geq 0$. Because the training data $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ satisfies for any $\boldsymbol{w}$, there exist at least d linearly independent $\boldsymbol{x}_i$ that activate $\boldsymbol{w}$. This guarantees for any $\boldsymbol{w}$ we must have at least d free variables. It follows that there must exist solutions λ_i to the linear system. This proves Claim (a).

Next, we show that (b) for any $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ that activate the exact same set of $\{\boldsymbol{x}_i\}$, the same set of λ_i can satisfy the constraints (E.54) and (E.56) of both $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$. Because $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ are activated by the same set of $\boldsymbol{x}_i$, this implies

$$\boldsymbol{\beta}_{\boldsymbol{w}_1} = c \cdot \sum_{i=1}^{n} \lambda_i \cdot \boldsymbol{x}_i \cdot \mathbb{I}\left(\boldsymbol{w}_1^\top \boldsymbol{x}_i \geq 0\right) = c \cdot \sum_{i=1}^{n} \lambda_i \cdot \boldsymbol{x}_i \cdot \mathbb{I}\left(\boldsymbol{w}_2^\top \boldsymbol{x}_i \geq 0\right) = \boldsymbol{\beta}_{\boldsymbol{w}_2}$$

Since λ_i already satisfy constraint (E.54) for $\boldsymbol{w}_1$, they also satisfy that for $\boldsymbol{w}_2$. Thus, it remains to show that $\boldsymbol{\beta}_{\boldsymbol{w}_1} + \boldsymbol{\beta}'_{\boldsymbol{w}_1} \cdot \boldsymbol{w}_1 = \boldsymbol{\beta}_{\boldsymbol{w}_2} + \boldsymbol{\beta}'_{\boldsymbol{w}_2} \cdot \boldsymbol{w}_1$ assuming $\boldsymbol{\beta}_{\boldsymbol{w}_1} = \boldsymbol{\beta}_{\boldsymbol{w}_2}$, $\boldsymbol{\beta}'_{\boldsymbol{w}_1} = \boldsymbol{w}_1^\top \boldsymbol{\beta}_{\boldsymbol{w}_1}$, and $\boldsymbol{\beta}'_{\boldsymbol{w}_2} = \boldsymbol{w}_2^\top \boldsymbol{\beta}_{\boldsymbol{w}_2}$. This indeed holds because

$$\begin{aligned}
& \boldsymbol{\beta}_{\boldsymbol{w}_1} + \boldsymbol{\beta}'_{\boldsymbol{w}_1} \cdot \boldsymbol{w}_1 = \boldsymbol{\beta}_{\boldsymbol{w}_2} + \boldsymbol{\beta}'_{\boldsymbol{w}_2} \cdot \boldsymbol{w}_2 \\
\iff \quad & \boldsymbol{\beta}'_{\boldsymbol{w}_1} \cdot \boldsymbol{w}_1^\top = \boldsymbol{\beta}'_{\boldsymbol{w}_2} \cdot \boldsymbol{w}_2^\top \\
\iff \quad & \boldsymbol{w}_1^\top \boldsymbol{\beta}_{\boldsymbol{w}_1} \boldsymbol{w}_1^\top = \boldsymbol{w}_2^\top \boldsymbol{\beta}_{\boldsymbol{w}_2} \boldsymbol{w}_2^\top \\
\iff \quad & \boldsymbol{w}_1^\top \boldsymbol{w}_1 \boldsymbol{\beta}_{\boldsymbol{w}_1}^\top = \boldsymbol{w}_2^\top \boldsymbol{w}_2 \boldsymbol{\beta}_{\boldsymbol{w}_2}^\top \\
\iff \quad & 1 \cdot \boldsymbol{\beta}_{\boldsymbol{w}_1}^\top = 1 \cdot \boldsymbol{\beta}_{\boldsymbol{w}_2}^\top \\
\iff \quad & \boldsymbol{\beta}_{\boldsymbol{w}_1} = \boldsymbol{\beta}_{\boldsymbol{w}_1}
\end{aligned}$$

Here, we used the fact that $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ are vectors on the unit sphere. This proves Claim (b).

Finally, we show (c) that Whenever we rotate a $\boldsymbol{w}_1$ to a $\boldsymbol{w}_2$ so that the set of $\boldsymbol{x}_i$ being activated changed, we can still find λ_i that satisfy constraint of both $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$. Suppose we rotate $\boldsymbol{w}_1$ to $\boldsymbol{w}_2$ so that $\boldsymbol{w}_2$ lost activation with $\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_p$ which in the set of linearly independent $\boldsymbol{x}_i$'s being activated by $\boldsymbol{w}_1$ and their opposite vectors $-\boldsymbol{x}_i$ are also in the training set (without loss of generality). Then $\boldsymbol{w}_2$ must now also get activated by $-\boldsymbol{x}_1, -\boldsymbol{x}_2, ..., -\boldsymbol{x}_p$. This is because if $\boldsymbol{w}_2^\top \boldsymbol{x}_i < 0$, we must have $\boldsymbol{w}_2^\top(-\boldsymbol{x}_i) > 0$.

Recall that in the proof of Claim (a), we only needed the λ_i from linearly independent $\boldsymbol{x}_i$ that we used to solve the linear systems, and their opposite as the free variables to solve the linear system of d equations. Hence, we can set λ to 0 for the other $\boldsymbol{x}_i$ while still satisfying the linear system. Then, suppose there exists λ_i that satisfy

$$\boldsymbol{\beta}_{\boldsymbol{w}_1}^{(k)} = c \cdot \sum_{i=1}^{d} \lambda_i \cdot \boldsymbol{x}_i^{(k)}$$

where the $\boldsymbol{x}_i$ are the linearly independent vectors that activate $\boldsymbol{w}_1$ with opposite vectors in the training set, which we have proved in (a). Then we can satisfy the constraint for $\boldsymbol{\beta}_{\boldsymbol{w}_2}$ below

$$\boldsymbol{\beta}_{\boldsymbol{w}_2}^{(k)} = c \cdot \sum_{i=1}^{p} \hat{\lambda}_i \cdot (-\boldsymbol{x}_i)^{(k)} + \sum_{i=p+1}^{d} \lambda_i \cdot \boldsymbol{x}_i^{(k)}$$

by setting $\hat{\lambda}_i = -\lambda_i$ for $i = 1...p$. Indeed, this gives

$$\begin{aligned}\boldsymbol{\beta}_{\boldsymbol{w}_2}^{(k)} &= c \cdot \sum_{i=1}^{p} (-\lambda_i) \cdot (-\boldsymbol{x}_i)^{(k)} + \sum_{i=p+1}^{d} \lambda_i \cdot \boldsymbol{x}_i^{(k)} \\ &= c \cdot \sum_{i=1}^{d} \lambda_i \cdot \boldsymbol{x}_i^{(k)}\end{aligned}$$

Thus, we can also find λ_i that satisfy the constraint for $\boldsymbol{\beta}_{\boldsymbol{w}_2}$. Here, we do not consider the case where $\boldsymbol{w}_2$ is parallel with an $\boldsymbol{x}_i$ because such $\boldsymbol{w}_2$ has measure zero. Note that we can apply this argument iteratively because the flipping the sign always works and will not create any inconsistency.

Moreover, we can show that the constraint for $\boldsymbol{\beta}'_{\boldsymbol{w}2}$ is satisfied by a similar argument as in proof of Claim (b). This follows from the fact that our construction makes $\boldsymbol{\beta}_{\boldsymbol{w}_1} = \boldsymbol{\beta}_{\boldsymbol{w}_2}$. Then we can follow the same argument as in (b) to show that $\boldsymbol{\beta}_{\boldsymbol{w}_1} + \boldsymbol{\beta}'_{\boldsymbol{w}_1} \cdot \boldsymbol{w}_1 = \boldsymbol{\beta}_{\boldsymbol{w}_2} + \boldsymbol{\beta}'_{\boldsymbol{w}_2} \cdot \boldsymbol{w}_1$. This completes the proof of Claim (c).

In summary, combining Claim (a), (b) and (c) gives that Claim E.3 holds. That is, given our training data, the global optimum to the constrained optimization problem of finding the min-norm solution among functions that fit the training data satisfies $\boldsymbol{\beta}_{\boldsymbol{w}} + \boldsymbol{\beta}'_{\boldsymbol{w}} \cdot \boldsymbol{w} = 2\boldsymbol{\beta}_g$. We also showed that this claim implies exact extrapolation, i.e., the network's learned function $f(\boldsymbol{x})$ is equal to the true underlying function $g(\boldsymbol{x})$ for all $\boldsymbol{x} \in \mathbb{R}^d$. This completes the proof.

E.2.3 Proof of Theorem 7.3

Proof of the asymptotic convergence to extrapolation builds upon our proof of exact extrapolation, i.e., Lemma 7.2. The proof idea is that if the training data distribution has support at all directions, when the number of samples $n \to \infty$, asymptotically the training set will converge to some imaginary training set that satisfies the condition for exact extrapolation. Since if training data are close the neural tangent kernels are also close, the predictions or learned function will converge to a function that achieves perfect extrapolation, that is, the true underlying function.

Asymptotic convergence of data sets. We first show the training data converge to a data set that satisfies the exact extrapolation condition in Lemma 7.2. Suppose training data $\{\boldsymbol{x}_i\}_{i=1}^n$ are sampled from a distribution whose support contains a connected set $\mathcal{S}$ that intersects all directions, i.e., for any non-zero $\boldsymbol{w} \in \mathbb{R}^d$, there exists $k > 0$ so that $k\boldsymbol{w} \in \mathcal{S}$.

Let us denote by $\mathcal{S}$ the set of datasets that satisfy the condition in Lemma 7.2. In fact, we will use a relaxed condition in the proof of Lemma 7.2 (Lemma 7.2 in the main text uses a stricter condition for simplicity of exposition). Given a general dataset $\boldsymbol{X}$ and a dataset $\boldsymbol{S} \in \mathcal{S}$ of the same size n, let $\sigma(\boldsymbol{X}, \boldsymbol{S})$ denote a matching of their data points, i.e., σ outputs a sequence of pairs

$$\begin{aligned} &\sigma(\boldsymbol{X}, \boldsymbol{S})_i = (\boldsymbol{x}_i, \boldsymbol{s}_i) \quad \text{for } i \in [n] \\ &s.t. \quad \boldsymbol{X} = \{\boldsymbol{x}_i\}_{i=1}^n \\ &\qquad\quad \boldsymbol{S} = \{\boldsymbol{s}_i\}_{i=1}^n \end{aligned}$$

Let $\ell : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ be the $l2$ distance that takes in a pair of points. We then define the distance between the datasets $d(\boldsymbol{X}, \boldsymbol{S})$ as the minimum sum of $l2$ distances of their data points over all possible matching.

$$d(\boldsymbol{X}, \boldsymbol{S}) = \begin{cases} \min\limits_{\sigma} \sum\limits_{i=1}^{n} \ell\left(\sigma\left(\boldsymbol{X}, \boldsymbol{S}\right)_i\right) & |\boldsymbol{X}| = |\boldsymbol{S}| = n \\ \infty & |\boldsymbol{X}| \neq |\boldsymbol{S}| \end{cases}$$

We can then define a "closest distance to perfect dataset" function $\mathcal{D}^* : \mathcal{X} \to \mathbb{R}$ which maps a dataset $\boldsymbol{X}$ to the minimum distance of $\boldsymbol{X}$ to any dataset in $\mathcal{S}$

$$\mathcal{D}^*\left(\boldsymbol{X}\right) = \min_{\boldsymbol{S} \in \mathcal{S}} d\left(\boldsymbol{X}, \boldsymbol{S}\right)$$

It is easy to see that for any dataset $\boldsymbol{X} = \{\boldsymbol{x}_i\}_{i=1}^n$, $\mathcal{D}^*\left(\boldsymbol{X}\right)$ can be bounded by the minimum of the closest distance to perfect dataset $\mathcal{D}^*$ of sub-datasets of $\boldsymbol{X}$ of size $2d$.

$$\mathcal{D}^*\left(\{\boldsymbol{x}_i\}_{i=1}^n\right) \leq \min_{k=1}^{\lfloor n/2d \rfloor} \mathcal{D}^*\left(\{\boldsymbol{x}_j\}_{j=(k-1)*2d+1}^{k*2d}\right) \tag{E.57}$$

This is because for any $\boldsymbol{S} \in \mathcal{S}$, and any $\boldsymbol{S} \subseteq \boldsymbol{S}'$, we must have $\boldsymbol{S}' \in \mathcal{S}$ because a dataset satisfies exact extrapolation condition as long as it contains some key points. Thus, adding more data will not hurt, i.e., for any $\boldsymbol{X}_1 \subseteq \boldsymbol{X}_2$, we always have

$$\mathcal{D}^*\left(\boldsymbol{X}_{\boldsymbol{1}}\right) \leq \mathcal{D}^*\left(\boldsymbol{X}_2\right)$$

Now let us denote by $\boldsymbol{X}_n$ a random dataset of size n where each $\boldsymbol{x}_i \in \boldsymbol{X}_n$ is sampled from the training distribution. Recall that our training data $\{\boldsymbol{x}_i\}_{i=1}^n$ are sampled from a distribution whose support contains a connected set $\mathcal{S}^*$ that intersects all directions, i.e., for any non-zero $\boldsymbol{w} \in \mathbb{R}^d$, there exists $k > 0$ so that $k\boldsymbol{w} \in \mathcal{S}^*$. It follows that for a random dataset $\boldsymbol{X}_{2d}$ of size $2d$, the probability that $\mathcal{D}^*(\boldsymbol{X}_{2d}) > \epsilon$ happens is less than 1 for any $\epsilon > 0$.

First there must exist $\boldsymbol{S}_0 = \{\boldsymbol{s}_i\}_{i=1}^{2d} \in \mathcal{S}$ of size $2d$, e.g., orthogonal basis and their opposite vectors. Observe that if we scale any $\boldsymbol{s}_i$ by $k > 0$, the resulting dataset is still in $\mathcal{S}$ by the definition of $\mathcal{S}$. We denote the set of datasets where we are allowed to scale elements of $\boldsymbol{S}_0$ by $\mathcal{S}_0$. It follows that

$$\begin{aligned}
\mathbb{P}\left(\mathcal{D}^*(\boldsymbol{X}_{2d}) > \epsilon\right) &= \mathbb{P}\left(\min_{\boldsymbol{S}\in\mathcal{S}} d\left(\boldsymbol{X}_{2d}, \boldsymbol{S}\right) > \epsilon\right) \\
&\leq \mathbb{P}\left(\min_{\boldsymbol{S}\in\mathcal{S}_0} d\left(\boldsymbol{X}_{2d}, \boldsymbol{S}\right) > \epsilon\right) \\
&= \mathbb{P}\left(\min_{\boldsymbol{S}\in\mathcal{S}_0} \min_{\sigma} \sum_{i=1}^{n} \ell\left(\sigma\left(\boldsymbol{X}_{2d}, \boldsymbol{S}\right)_i\right) > \epsilon\right) \\
&= 1 - \mathbb{P}\left(\min_{\boldsymbol{S}\in\mathcal{S}_0} \min_{\sigma} \sum_{i=1}^{n} \ell\left(\sigma\left(\boldsymbol{X}_{2d}, \boldsymbol{S}\right)_i\right) \leq \epsilon\right) \\
&\leq 1 - \mathbb{P}\left(\min_{\boldsymbol{S}\in\mathcal{S}_0} \min_{\sigma} \max_{i=1}^{n} \ell\left(\sigma\left(\boldsymbol{X}_{2d}, \boldsymbol{S}\right)_i\right) \leq \epsilon\right) \\
&\leq \delta < 1
\end{aligned}$$

where we denote the bound of $\mathbb{P}\left(\mathcal{D}^*(\boldsymbol{X}_{2d}) > \epsilon\right)$ by $\delta < 1$, and the last step follows from

$$\mathbb{P}\left(\min_{\boldsymbol{S}\in\mathcal{S}_0} \min_{\sigma} \max_{i=1}^{n} \ell\left(\sigma\left(\boldsymbol{X}_{2d}, \boldsymbol{S}\right)_i\right) \leq \epsilon\right) > 0$$

which further follows from the fact that for any $\boldsymbol{s}_i \in \mathcal{S}_0$, by the assumption on training distribution, we can always find $k > 0$ so that $k\boldsymbol{s}_i \in \mathcal{S}^*$, a connected set in the support of training distribution. By the connectivity of support $\mathcal{S}^*$, $k\boldsymbol{s}_i$ cannot be an isolated point in $\mathcal{S}^*$, so for any $\epsilon > 0$, we must have

$$\int_{\|\boldsymbol{x}-k\boldsymbol{s}_i\|\leq\epsilon, \boldsymbol{x}\in\mathcal{S}^*} f_{\boldsymbol{X}}(\boldsymbol{x})\mathrm{d}\boldsymbol{x} > 0$$

Hence, we can now apply (E.57) to bound $\mathcal{D}^*(\boldsymbol{X}_n)$. Given any $\epsilon > 0$, we have

$$\begin{aligned}
\mathbb{P}\left(\mathcal{D}^*(\boldsymbol{X}_n) > \epsilon\right) &= 1 - \mathbb{P}\left(\mathcal{D}^*(\boldsymbol{X}_n) \leq \epsilon\right) \\
&\leq 1 - \mathbb{P}\left(\min_{k=1}^{\lfloor n/2d \rfloor} \mathcal{D}^*\left(\{\boldsymbol{x}_j\}_{j=(k-1)*2d+1}^{k*2d}\right) \leq \epsilon\right) \\
&\leq 1 - \left(1 - \prod_{k=1}^{\lfloor n/2d \rfloor} \mathbb{P}\left(\mathcal{D}^*\left(\{\boldsymbol{x}_j\}_{j=(k-1)*2d+1}^{k*2d}\right) > \epsilon\right)\right) \\
&= \prod_{k=1}^{\lfloor n/2d \rfloor} \mathbb{P}\left(\mathcal{D}^*\left(\{\boldsymbol{x}_j\}_{j=(k-1)*2d+1}^{k*2d}\right) > \epsilon\right) \\
&\leq \delta^{\lfloor n/2d \rfloor}
\end{aligned}$$

Here $\delta < 1$. This implies $\mathcal{D}^*(\boldsymbol{X}_n) \xrightarrow{p} 0$, i.e.,

$$\lim_{n\to\infty} \mathbb{P}\left(\mathcal{D}^*(\boldsymbol{X}_n) > \epsilon\right) = 0 \quad \forall \epsilon > 0 \tag{E.58}$$

(E.58) says as the number of training samples $n \to \infty$, our training set will converge in probability to a dataset that satisfies the requirement for exact extrapolation.

Asymptotic convergence of predictions. Let $\text{NTK}(\boldsymbol{x}, \boldsymbol{x}') : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ denote the neural tangent kernel for a two-layer ReLU MLP. It is easy to see that if $\boldsymbol{x} \to \boldsymbol{x}^*$, then $\text{NTK}(\boldsymbol{x}, \cdot) \to \text{NTK}(\boldsymbol{x}^*, \cdot)$ (Arora et al. [2019c]). Let $\text{NTK}_{\text{train}}$ denote the $n \times n$ kernel matrix for training data.

We have shown that our training set converges to a perfect data set that satisfies conditions of exact extrapolation. Moreover, note that our training set will only have a finite number of (not increase with n) $\boldsymbol{x}_i$ that are not precisely the same as those in a perfect dataset. This is because a perfect data only contains a finite number of key points and the other points can be replaced by any other points while still being a perfect data set. Thus, we have $\text{NTK}_{\text{train}} \to N^*$, where N^* is the $n \times n$ NTK matrix for some perfect data set.

Because neural tangent kernel is positive definite, we have $\text{NTK}_{\text{train}}^{-1} \to {N^*}^{-1}$. Recall that for any $\boldsymbol{x} \in \mathbb{R}^d$, the prediction of NTK is

$$f_{\text{NTK}}(\boldsymbol{x}) = (\text{NTK}(\boldsymbol{x}, \boldsymbol{x}_1), ..., \text{NTK}(\boldsymbol{x}, \boldsymbol{x}_n)) \cdot \text{NTK}_{\text{train}}^{-1} \boldsymbol{Y},$$

where $\text{NTK}_{\text{train}}$ is the $n \times n$ kernel for training data, $\text{NTK}(\boldsymbol{x}, \boldsymbol{x}_i)$ is the kernel value between test data $\boldsymbol{x}$ and training data $\boldsymbol{x}_i$, and $\boldsymbol{Y}$ is training labels.

Similarly, we have $(\text{NTK}(\boldsymbol{x}, \boldsymbol{x}_1), ..., \text{NTK}(\boldsymbol{x}, \boldsymbol{x}_n)) \to (\text{NTK}(\boldsymbol{x}, \boldsymbol{x}_1^*), ..., \text{NTK}(\boldsymbol{x}, \boldsymbol{x}_n^*))$, where x_i^* is a perfect data set that our training set converges to. Combining this with $\text{NTK}_{\text{train}}^{-1} \to {N^*}^{-1}$ gives

$$f_{\text{NTK}} \xrightarrow{p} f_{\text{NTK}}^* = g,$$

where f_{NTK} is the function learned using our training set, and f_{NTK}^* is that learned using a perfect data set, which is equal to the true underlying function g. This completes the proof.

E.2.4 Proof of Corollary 8.1

In order for GNN with linear aggregations

$$h_u^{(k)} = \sum_{v \in \mathcal{N}(u)} \text{MLP}^{(k)}\Big(h_u^{(k)}, h_v^{(k)}, \boldsymbol{x}_{(u,v)}\Big),$$
$$h_G = \text{MLP}^{(K+1)}\Big(\sum_{u \in G} h_u^{(K)}\Big),$$

to extrapolate in the maximum degree task, it must be able to simulate the underlying function

$$h_G = \max_{u \in G} \sum_{v \in \mathcal{N}(u)} 1$$

Because the max function cannot be decomposed as the composition of piece-wise linear functions, the $\text{MLP}^{(K+1)}$ module in GNN must learn a function that is not piece-wise linear over domains outside the training data range. Since Theorem 7.1 proves for two-layer overparameterized MLPs, here we also assume $\text{MLP}^{(K+1)}$ is a two-layer overparameterized MLP, although the result can be extended to more layers. It then follows from Theorem 7.1

that for any input and label (and thus gradient), MLP$^{(K+1)}$ will converge to linear functions along directions from the origin. Hence, there are always domains where the GNN cannot learn a correct target function.

E.2.5 Proof of Theorem 8.2

Our proof applies the similar proof techniques for Lemma 7.2 and 7.3 to Graph Neural Networks (GNNs). This is essentially an analysis of Graph Neural Tangent Kernel (GNTK), i.e., neural tangent kernel of GNNs.

We first define the simple GNN architecture we will be analyzing, and then present the GNTK for this architecture. Suppose $G = (V, E)$ is an input graph without edge feature, and $\boldsymbol{x}_u \in \mathbb{R}^d$ is the node feature of any node $u \in V$. Let us consider the simple one-layer GNN whose input is G and output is h_G

$$h_G = W^{(2)} \max_{u \in G} \sum_{v \in \mathcal{N}(u)} W^{(1)} \boldsymbol{x}_v \tag{E.59}$$

Note that our analysis can be extended to other variants of GNNs, e.g., with non-empty edge features, ReLU activation, different neighbor aggregation and graph-level pooling architectures. We analyze this GNN for simplicity of exposition.

Next, let us calculate the feature map of the neural tangent kernel for this GNN. Recall from Section E.1 that consider a graph neural network $f(\boldsymbol{\theta}, G) : \mathcal{G} \to \mathbb{R}$ where $\boldsymbol{\theta} \in \mathbb{R}^m$ is the parameters in the network and $G \in \mathcal{G}$ is the input graph. Then the neural tangent kernel is

$$\boldsymbol{H}_{ij} = \left\langle \frac{\partial f(\boldsymbol{\theta}, G_i)}{\partial \boldsymbol{\theta}}, \frac{\partial f(\boldsymbol{\theta}, G_j)}{\partial \boldsymbol{\theta}} \right\rangle,$$

where $\boldsymbol{\theta}$ are the infinite-dimensional parameters. Hence, the gradients with respect to all parameters give a natural feature map. Let us denote, for any node u, the degree of u by

$$\boldsymbol{h}_u = \sum_{v \in \mathcal{N}(u)} \boldsymbol{x}_v \tag{E.60}$$

It then follows from simple computation of derivative that the following is a feature map of the GNTK for (E.59)

$$\phi(G) = c \cdot \left(\max_{u \in G} \left(\boldsymbol{w}^{(k)\top} \boldsymbol{h}_u \right), \sum_{u \in G} \mathbb{I}\left(u = \arg\max_{v \in G} \boldsymbol{w}^{(k)\top} \boldsymbol{h}_v \right) \cdot \boldsymbol{h}_u, \ldots \right), \tag{E.61}$$

where $\boldsymbol{w}^{(k)} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, with k going to infinity. c is a constant, and $\mathbb{I}$ is the indicator function.

Next, given training data $\{(G_i, y_i\}_{i=1}^n$, let us analyze the function learned by GNN through the min-norm solution in the GNTK feature space. The same proof technique is also used in Lemma 7.2 and 7.3.

Recall the assumption that all graphs have uniform node feature, i.e., the learning task only considers graph structure, but not node feature. We assume $\boldsymbol{x}_v = 1$ without loss of generality. Observe that in this case, there are two directions, positive or negative, for one-dimensional Gaussian distribution. Hence, we can simplify our analysis by combining the effect of linear coefficients for $\boldsymbol{w}$ in the same direction as in Lemma 7.2 and 7.3.

Similarly, for any $\boldsymbol{w}$, let us define $\hat{\boldsymbol{\beta}}_{\boldsymbol{w}} \in \mathbb{R}$ as the linear coefficient corresponding to $\sum_{u \in G} \mathbb{I}\left(u = \arg\max_{v \in G} \boldsymbol{w}^\top \boldsymbol{h}_v \right) \cdot \boldsymbol{h}_u$ in RKHS space, and denote by $\hat{\boldsymbol{\beta}}'_{\boldsymbol{w}} \in \mathbb{R}$ the weight for $\max_{u \in G} \left(\boldsymbol{w}^\top \boldsymbol{h}_u \right)$. Similarly, we can combine the effect of all $\hat{\boldsymbol{\beta}}$ in the same direction as in Lemma 7.2 and 7.3. We define the combined effect with $\boldsymbol{\beta}_{\boldsymbol{w}}$ and $\boldsymbol{\beta}'_{\boldsymbol{w}}$. This allows us to reason about $\boldsymbol{w}$ with two directions, $+$ and $-$.

Recall that the underlying reasoning function, maximum degree, is

$$g(G) = \max_{u \in G} \boldsymbol{h}_u.$$

We formulate the constrained optimization problem, i.e., min-norm solution in GNTK feature space that fits all training data, as

$$\min_{\hat{\boldsymbol{\beta}}, \hat{\boldsymbol{\beta}}'} \int \hat{\boldsymbol{\beta}}_{\boldsymbol{w}}^2 + \hat{\boldsymbol{\beta}}_{\boldsymbol{w}}'^2 \mathrm{d}\mathbb{P}(\boldsymbol{w})$$

$$s.t. \int \sum_{u \in G_i} \mathbb{I}\left(u = \arg\max_{v \in G} \boldsymbol{w} \cdot \boldsymbol{h}_v \right) \cdot \hat{\boldsymbol{\beta}}_{\boldsymbol{w}} \cdot \boldsymbol{h}_u + \max_{u \in G_i} (\boldsymbol{w} \cdot \boldsymbol{h}_u) \cdot \hat{\boldsymbol{\beta}}'_{\boldsymbol{w}} \mathrm{d}\mathbb{P}(\boldsymbol{w}) = \max_{u \in G_i} \boldsymbol{h}_u \quad \forall i \in [n],$$

where G_i is the i-th training graph and $\boldsymbol{w} \sim \mathcal{N}(0,1)$. By combining the effect of $\hat{\boldsymbol{\beta}}$, and taking the derivative of the Lagrange for the constrained optimization problem and setting to zero, we get the global optimum solution satisfy the following constraints.

$$\boldsymbol{\beta}_+ = c \cdot \sum_{i=1}^{n} \lambda_i \cdot \sum_{u \in G_i} \boldsymbol{h}_u \cdot \mathbb{I}\left(u = \arg\max_{v \in G_i} \boldsymbol{h}_v\right) \tag{E.62}$$

$$\boldsymbol{\beta}_- = c \cdot \sum_{i=1}^{n} \lambda_i \cdot \sum_{u \in G_i} \boldsymbol{h}_u \cdot \mathbb{I}\left(u = \arg\min_{v \in G_i} \boldsymbol{h}_v\right) \tag{E.63}$$

$$\boldsymbol{\beta}'_+ = c \cdot \sum_{i=1}^{n} \lambda_i \cdot \max_{u \in G_i} \boldsymbol{h}_u \tag{E.64}$$

$$\boldsymbol{\beta}'_- = c \cdot \sum_{i=1}^{n} \lambda_i \cdot \min_{u \in G_i} \boldsymbol{h}_u \tag{E.65}$$

$$\max_{u \in G_i} \boldsymbol{h}_u = \boldsymbol{\beta}_+ \cdot \sum_{u \in G_i} \mathbb{I}\left(u = \arg\max_{v \in G_i} \boldsymbol{h}_v\right) \cdot \boldsymbol{h}_u + \boldsymbol{\beta}'_+ \cdot \max_{u \in G_i} \boldsymbol{h}_u \tag{E.66}$$

$$+ \boldsymbol{\beta}_- \cdot \sum_{u \in G_i} \mathbb{I}\left(u = \arg\min_{v \in G_i} \boldsymbol{h}_v\right) \cdot \boldsymbol{h}_u + \boldsymbol{\beta}'_- \cdot \min_{u \in G_i} \boldsymbol{h}_u \quad \forall i \in [n] \tag{E.67}$$

where c is some constant, λ_i are the Lagrange parameters. Note that here we used the fact that there are two directions $+1$ and -1. This enables the simplification of Lagrange derivative. For a similar step-by-step derivation of Lagrange, refer to the proof of Lemma 7.2.

Let us consider the solution $\boldsymbol{\beta}'_+ = 1$ and $\boldsymbol{\beta}_+ = \boldsymbol{\beta}_- = \boldsymbol{\beta}'_- = 0$. It is clear that this solution can fit the training data, and thus satisfies (E.66). Moreover, this solution is equivalent to the underlying reasoning function, maximum degree, $g(G) = \max_{u \in G} \boldsymbol{h}_u$.

Hence, it remains to show that, given our training data, there exist λ_i so that the remaining four constraints are satisfies for this solution. Let us rewrite these constraints as a linear systems where the variables are λ_i

$$\begin{pmatrix} \boldsymbol{\beta}_+ \\ \boldsymbol{\beta}_- \\ \boldsymbol{\beta}'_+ \\ \boldsymbol{\beta}'_- \end{pmatrix} = c \cdot \sum_{i=1}^{n} \lambda_i \cdot \begin{pmatrix} \sum_{u \in G_i} \boldsymbol{h}_u \cdot \mathbb{I}\left(u = \arg\max_{v \in G_i} \boldsymbol{h}_v\right) \\ \sum_{u \in G_i} \boldsymbol{h}_u \cdot \mathbb{I}\left(u = \arg\min_{v \in G_i} \boldsymbol{h}_v\right) \\ \max_{u \in G_i} \boldsymbol{h}_u \\ \min_{u \in G_i} \boldsymbol{h}_u \end{pmatrix} \tag{E.68}$$

By standard theory of linear systems, there exist λ_i to solve (E.68) if there are at least four training data G_i whose following vectors linear independent

$$\begin{pmatrix} \sum_{u \in G_i} \boldsymbol{h}_u \cdot \mathbb{I}\left(u = \arg\max\limits_{v \in G_i} \boldsymbol{h}_v\right) \\ \sum_{u \in G_i} \boldsymbol{h}_u \cdot \mathbb{I}\left(u = \arg\min\limits_{v \in G_i} \boldsymbol{h}_v\right) \\ \max\limits_{u \in G_i} \boldsymbol{h}_u \\ \min\limits_{u \in G_i} \boldsymbol{h}_u \end{pmatrix} = \begin{pmatrix} \max\limits_{u \in G_i} \boldsymbol{h}_u \cdot N_i^{\max} \\ \min\limits_{u \in G_i} \boldsymbol{h}_u \cdot N_i^{\min} \\ \max\limits_{u \in G_i} \boldsymbol{h}_u \\ \min\limits_{u \in G_i} \boldsymbol{h}_u \end{pmatrix} \tag{E.69}$$

Here, $N_i^{\max}$ denotes the number of nodes that achieve the maximum degree in the graph G_i, and $N_i^{\min}$ denotes the number of nodes that achieve the min degree in the graph G_i. By the assumption of our training data that there are at least four $G_i \sim \mathcal{G}$ with linearly independent (E.69). Hence, our simple GNN learns the underlying function as desired.

This completes the proof.

E.2.6 Proof of Lemma E.1

Let W denote the span of the feature maps of training data $\boldsymbol{x}_i$, i.e.

$$W = \text{span}\left(\phi\left(\boldsymbol{x}_1\right), \phi\left(\boldsymbol{x}_2\right), ..., \phi\left(\boldsymbol{x}_n\right)\right).$$

Then we can decompose the coordinates of f_{NTK} in the RKHS space, $\boldsymbol{\beta}_{\text{NTK}}$, into a vector $\boldsymbol{\beta}_0$ for the component of f_{NTK} in the span of training data features W, and a vector $\boldsymbol{\beta}_1$ for the component in the orthogonal complement $W^\top$, i.e.,

$$\boldsymbol{\beta}_{\text{NTK}} = \boldsymbol{\beta}_0 + \boldsymbol{\beta}_1.$$

First, note that since f_{NTK} must be able to fit the training data (NTK is a universal kernel as we will discuss next), i.e.,

$$\phi(\boldsymbol{x}_i)^\top \boldsymbol{\beta}_{\text{NTK}} = y_i.$$

Thus, we have $\phi(\boldsymbol{x}_i)^\top \boldsymbol{\beta}_0 = y_i$. Then, $\boldsymbol{\beta}_0$ is uniquely determined by the kernel regression solution with respect to the neural tangent kernel

$$f_{\text{NTK}}(\boldsymbol{x}) = \Big(\Big\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}_1)\Big\rangle, ..., \Big\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}_n)\Big\rangle\Big) \cdot \text{NTK}_{\text{train}}^{-1} \boldsymbol{Y},$$

where $\text{NTK}_{\text{train}}$ is the $n \times n$ kernel for training data, $\Big\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}_i)\Big\rangle$ is the kernel between test data $\boldsymbol{x}$ and training data $\boldsymbol{x}_i$, and $\boldsymbol{Y}$ is training labels.

The kernel regression solution f_{NTK} is uniquely determined because the neural tangent kernel $\text{NTK}_{\text{train}}$ is positive definite assuming no two training data are parallel, which can be enforced with a bias term [Du et al., 2019c]. In any case, the solution is a min-norm by pseudo-inverse.

Moreover, a unique kernel regression solution f_{NTK} that spans the training data features corresponds to a unique representation in the RKHS space $\boldsymbol{\beta}_0$.

Since $\boldsymbol{\beta}_0$ and $\boldsymbol{\beta}_1$ are orthogonal, we also have the following

$$\|\boldsymbol{\beta}_{\text{NTK}}\|_2^2 = \|\boldsymbol{\beta}_0 + \boldsymbol{\beta}_1\|_2^2 = \|\boldsymbol{\beta}_0\|_2^2 + \|\boldsymbol{\beta}_1\|_2^2.$$

This implies the norm of $\boldsymbol{\beta}_{\text{NTK}}$ is at least as large as the norm of any $\boldsymbol{\beta}$ such that $\phi(\boldsymbol{x}_i)^\top \boldsymbol{\beta}_{\text{NTK}} = y_i$. Moreover, observe that the solution to kernel regression (E.4) is in the feature span of training data, given the kernel matrix for training data is full rank.

$$f_{\text{NTK}}(\boldsymbol{x}) = \Big(\Big\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}_1)\Big\rangle, ..., \Big\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}_n)\Big\rangle\Big) \cdot \text{NTK}_{\text{train}}^{-1} \boldsymbol{Y}.$$

Since $\boldsymbol{\beta}_1$ is for the component of f_{NTK} in the orthogonal complement of training data feature span, we must have $\boldsymbol{\beta}_1 = \mathbf{0}$. It follows that $\boldsymbol{\beta}_{\text{NTK}}$ is equivalent to

$$\begin{aligned} &\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2 \\ \text{s.t.} \quad &\phi(\boldsymbol{x}_i)^\top \boldsymbol{\beta} = y_i, \quad \text{for } i = 1, ..., n. \end{aligned}$$

as desired.

E.2.7 Proof of Lemma E.2

We first compute the neural tangent kernel $\text{NTK}(\boldsymbol{x}, \boldsymbol{x}')$ for a two-layer multi-layer perceptron (MLP) with ReLU activation function, and then show that it can be induced by the feature space $\phi(\boldsymbol{x})$ specified in the lemma so that $\text{NTK}(\boldsymbol{x}, \boldsymbol{x}') = \left\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \right\rangle$.

Recall that Jacot et al. [2018] have derived the general framework for computing the neural tangent kernel of a neural network with general architecture and activation function. This framework is also described in Arora et al. [2019c], Du et al. [2019b], which, in addition, compute the exact kernel formula for convolutional networks and Graph Neural Networks, respectively. Following the framework in Jacot et al. [2018] and substituting the general activation function σ with ReLU gives the kernel formula for a two-layer MLP with ReLU activation. This has also been described in several previous works [Du et al., 2019c, Chizat et al., 2019, Bietti and Mairal, 2019].

Below we describe the general framework in Jacot et al. [2018] and Arora et al. [2019c]. Let σ denote the activation function. The neural tangent kernel for an h-layer multi-layer perceptron can be recursively defined via a dynamic programming process. Here, $\Sigma^{(i)} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ for $i = 0...h$ is the covariance for the i-th layer.

$$\begin{aligned}
\Sigma^{(0)}(\boldsymbol{x}, \boldsymbol{x}') &= \boldsymbol{x}^\top \boldsymbol{x}', \\
\wedge^{(i)}(\boldsymbol{x}, \boldsymbol{x}') &= \begin{pmatrix} \Sigma^{(i-1)}(\boldsymbol{x}, \boldsymbol{x}) & \Sigma^{(i-1)}(\boldsymbol{x}, \boldsymbol{x}') \\ \Sigma^{(i-1)}(\boldsymbol{x}', \boldsymbol{x}) & \Sigma^{(i-1)}(\boldsymbol{x}', \boldsymbol{x}') \end{pmatrix}, \\
\Sigma^{(i)}(\boldsymbol{x}, \boldsymbol{x}') &= c \cdot \mathop{\mathbb{E}}_{u,v \sim \mathcal{N}(\boldsymbol{0}, \wedge^{(i)})} \left[\sigma(u)\sigma(v)\right].
\end{aligned}$$

The derivative covariance is defined similarly:

$$\dot{\Sigma}^{(i)}(\boldsymbol{x}, \boldsymbol{x}') = c \cdot \mathop{\mathbb{E}}_{u,v \sim \mathcal{N}(\boldsymbol{0}, \wedge^{(i)})} \left[\dot{\sigma}(u)\dot{\sigma}(v)\right].$$

Then the neural tangent kernel for an h-layer network is defined as

$$\text{NTK}^{(h-1)}(\boldsymbol{x}, \boldsymbol{x}') = \sum_{i=1}^{h} \left(\Sigma^{(i-1)}(\boldsymbol{x}, \boldsymbol{x}') \cdot \prod_{k=i}^{h} \dot{\Sigma}^{(k)}(\boldsymbol{x}, \boldsymbol{x}') \right),$$

where we let $\dot{\Sigma}^{(h)}(\boldsymbol{x}, \boldsymbol{x}') = 1$ for the convenience of notations.

We compute the explict NTK formula for a two-layer MLP with ReLU activation function by following this framework and substituting the general activation function with ReLU, i.e. $\sigma(a) = \max(0, a) = a \cdot \mathbb{I}(a \geq 0)$ and $\dot{\sigma}(a) = \mathbb{I}(a \geq 0)$.

$$\begin{aligned} \text{NTK}^{(1)}(\boldsymbol{x}, \boldsymbol{x}') &= \sum_{i=1}^{2} \left(\Sigma^{(i-1)}(\boldsymbol{x}, \boldsymbol{x}') \cdot \prod_{k=i}^{h} \dot{\Sigma}^{(k)}(\boldsymbol{x}, \boldsymbol{x}') \right) \\ &= \Sigma^{(0)}(\boldsymbol{x}, \boldsymbol{x}') \cdot \dot{\Sigma}^{(1)}(\boldsymbol{x}, \boldsymbol{x}') + \Sigma^{(1)}(\boldsymbol{x}, \boldsymbol{x}') \end{aligned}$$

So we can get the NTK via $\Sigma^{(1)}(\boldsymbol{x}, \boldsymbol{x}')$ and $\dot{\Sigma}^{(1)}(\boldsymbol{x}, \boldsymbol{x}')$, $\Sigma^{(0)}(\boldsymbol{x}, \boldsymbol{x}')$. Precisely,

$$\begin{aligned} &\Sigma^{(0)}(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^\top \boldsymbol{x}', \\ &\wedge^{(1)}(\boldsymbol{x}, \boldsymbol{x}') = \begin{pmatrix} \boldsymbol{x}^\top \boldsymbol{x} & \boldsymbol{x}^\top \boldsymbol{x}' \\ \boldsymbol{x}'^\top \boldsymbol{x} & \boldsymbol{x}'^\top \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{x}' \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{x} & \boldsymbol{x}' \end{pmatrix}, \\ &\Sigma^{(1)}(\boldsymbol{x}, \boldsymbol{x}') = c \cdot \mathop{\mathbb{E}}_{u,v \sim \mathcal{N}(\mathbf{0}, \wedge^{(1)})} \left[u \cdot \mathbb{I}(u \geq 0) \cdot v \cdot \mathbb{I}(v \geq 0) \right]. \end{aligned}$$

To sample from $\mathcal{N}(\mathbf{0}, \wedge^{(1)})$, we let L be a decomposition of $\wedge^{(1)}$, such that $\wedge^{(1)} = LL^\top$. Here, we can see that $L = (\boldsymbol{x}, \boldsymbol{x}')^\top$. Thus, sampling from $\mathcal{N}(\mathbf{0}, \wedge^{(1)})$ is equivalent to first sampling $\boldsymbol{w} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, and output

$$L\boldsymbol{w} = \boldsymbol{w}^\top (\boldsymbol{x}, \boldsymbol{x}').$$

Then we have the equivalent sampling $(u, v) = (\boldsymbol{w}^\top \boldsymbol{x}, \boldsymbol{w}^\top \boldsymbol{x}')$. It follows that

$$\Sigma^{(1)}(\boldsymbol{x}, \boldsymbol{x}') = c \cdot \mathop{\mathbb{E}}_{\boldsymbol{w} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})} \left[\boldsymbol{w}^\top \boldsymbol{x} \cdot \mathbb{I}\left(\boldsymbol{w}^\top \boldsymbol{x} \geq 0 \right) \cdot \boldsymbol{w}^\top \boldsymbol{x}' \cdot \mathbb{I}\left(\boldsymbol{w}^\top \boldsymbol{x}' \geq 0 \right) \right]$$

It follows from the same reasoning that

$$\dot{\Sigma}^{(1)}(\boldsymbol{x}, \boldsymbol{x}') = c \cdot \mathop{\mathbb{E}}_{\boldsymbol{w} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})} \left[\mathbb{I}\left(\boldsymbol{w}^\top \boldsymbol{x} \geq 0 \right) \cdot \mathbb{I}\left(\boldsymbol{w}^\top \boldsymbol{x}' \geq 0 \right) \right].$$

The neural tangent kernel for a two-layer MLP with ReLU activation is then

$$\begin{aligned}\text{NTK}^{(1)}(\boldsymbol{x},\boldsymbol{x}') &= \Sigma^{(0)}(\boldsymbol{x},\boldsymbol{x}')\cdot\dot{\Sigma}^{(1)}(\boldsymbol{x},\boldsymbol{x}')+\Sigma^{(1)}(\boldsymbol{x},\boldsymbol{x}')\\ &= c\cdot\underset{\boldsymbol{w}\sim\mathcal{N}(\boldsymbol{0},\boldsymbol{I})}{\mathbb{E}}\left[\boldsymbol{x}^\top\boldsymbol{x}'\cdot\mathbb{I}\left(\boldsymbol{w}^\top\boldsymbol{x}\geq 0\right)\cdot\mathbb{I}\left(\boldsymbol{w}^\top\boldsymbol{x}'\geq 0\right)\right]\\ &+ c\cdot\underset{\boldsymbol{w}\sim\mathcal{N}(\boldsymbol{0},\boldsymbol{I})}{\mathbb{E}}\left[\boldsymbol{w}^\top\boldsymbol{x}\cdot\mathbb{I}\left(\boldsymbol{w}^\top\boldsymbol{x}\geq 0\right)\cdot\boldsymbol{w}^\top\boldsymbol{x}'\cdot\mathbb{I}\left(\boldsymbol{w}^\top\boldsymbol{x}'\geq 0\right)\right].\end{aligned}$$

Next, we use the kernel formula to compute a feature map for a two-layer MLP with ReLU activation function. Recall that by definition a valid feature map must satisfy the following condition

$$\text{NTK}^{(1)}(\boldsymbol{x},\boldsymbol{x}') = \left\langle\phi(\boldsymbol{x}),\phi(\boldsymbol{x}')\right\rangle$$

It is easy to see that the way we represent our NTK formula makes it easy to find such a decomposition. The following infinite-dimensional feature map would satisfy the requirement because the inner product of $\phi(\boldsymbol{x})$ and $\phi(\boldsymbol{x}')$ for any $\boldsymbol{x}$, $\boldsymbol{x}'$ would be equivalent to the expected value in NTK, after we integrate with respect to the density function of $\boldsymbol{w}$.

$$\phi\left(\boldsymbol{x}\right) = c'\left(\boldsymbol{x}\cdot\mathbb{I}\left(\boldsymbol{w}^{(k)^\top}\boldsymbol{x}\geq 0\right),\boldsymbol{w}^{(k)^\top}\boldsymbol{x}\cdot\mathbb{I}\left(\boldsymbol{w}^{(k)^\top}\boldsymbol{x}\geq 0\right),\ldots\right),$$

where $\boldsymbol{w}^{(k)}\sim\mathcal{N}(\boldsymbol{0},\boldsymbol{I})$, with k going to infinity. c' is a constant, and $\mathbb{I}$ is the indicator function. Note that here the density of features of $\phi(\boldsymbol{x})$ is determined by the density of $\boldsymbol{w}$, i.e. Gaussian.

E.3 Experimental Details

In this section, we describe the model, data and training details for reproducing our experiments. Our experiments support all of our theoretical claims and insights.

Overview. We classify our experiments into the following major categories, each of which includes several ablation studies:

1) Learning tasks where the target functions are **simple nonlinear functions** in various dimensions and training/test distributions: quadratic, cosine, square root, and l1 norm functions, with MLPs with a wide range of hyper-parameters.

   This validates our implications on MLPs generally cannot extrapolate in tasks with nonlinear target functions, unless the nonlinear function is directionally linear out-of-distribution. In the latter case, the extrapolation error is more sensitive to the hyper-parameters.

2) Computation of the **R-Squared** of MLP's learned functions along (thousands of) randomly sampled directions in out-of-distribution domain.

   This validates Theorem 7.1 and shows the convergence rate is very high in practice, and often happens immediately out of training range.

3) Learning tasks where the target functions are **linear functions** with MLPs. These validate Theorem 7.3 and Lemma 7.2, i.e., MLPs can extrapolate if the underlying function is linear under conditions on training distribution. This section includes four ablation studies:

   a) Training distribution satisfy the conditions in Theorem 7.3 and cover all directions, and hence, MLPs extrapolate.

   b) Training data distribution is **restricted** in some **directions**, e.g., restricted to be positive/negative/constant in some feature dimensions. This shows when training distribution is restrictive in directions, MLPs may fail to extrapolate.

   c) Exact extrapolation with **infinitely-wide neural networks**, i.e., exact computation with **neural tangent kernel** (NTK) on the data regime in Lemma 7.2. This is mainly for theoretical understanding.

4) MLPs with cosine, quadratic, and tanh activation functions.

5) Learning **maximum degree of graphs** with Graph Neural Networks. Extrapolation on graph structure, number of nodes, and node features. To show the role of architecture for extrapolation, we study the following GNN architecture regimes.

   a) GNN with graph-level max-pooling and neighbor-level sum-pooling. By Theorem 8.2, this GNN architecture extrapolates in max degree with appropriate training data.

   b) GNN with graph-level and neighbor-level sum-pooling. By Corollary 8.1, this default GNN architecture cannot extrapolate in max degree.

   To show the importance of training distribution, i.e., graph structure in training set, we study the following training data regimes.

   a) Node features are **identical**, e.g., 1. In such regimes, our learning tasks only consider graph structure. We consider training sets sampled from various graph structure, and find only those satisfy conditions in Theorem 8.2 enables GNNs with graph-level max-pooling to extrapolate.

   b) Node features are **spurious** and continuous. This also requires extrapolation on OOD node features. GNNs with graph-level max-pooling with appropriate training sets also extrapolate to OOD spurious node features.

6) Learning the length of the **shortest path** between given source and target nodes, with Graph Neural Networks. Extrapolation on graph structure, number of nodes, and edge weights. We study the following regimes.

   a) Continuous features. Edge and node features are real values. This regime requires extrapolating to graphs with edge weights out of training range.

Test graphs are all sampled from the "general graphs" family with a diverse range of structure. Regarding the type of training graph structure, we consider two schemes. Both schemes show a U-shape curve of extrapolation error with respect to the sparsity of training graphs.

a) Specific graph structure: path, cycle, tree, expander, ladder, complete graphs, general graphs, 4-regular graphs.

b) Random graphs with a range of probability p of an edge between any two nodes. Smaller p samples sparse graphs and large p samples dense graphs.

7) **Physical reasoning** of the **n-Body problem** in the orbit setting with Graph Neural Networks. We show that GNNs on the original features from previous works fail to extrapolate to unseen masses and distances. On the other hand, we show extrapolation can be achieved via an improved representation of the input edge features. We consider the following extrapolation regimes.

   a) Extrapolation on the masses of the objects.

   b) Extrapolation on the distances between objects.

   We consider the following two *input representation* schemes to compare the effects of how representation helps extrapolation.

   a) Original features. Following previous works on solving n-body problem with GNNs, the edge features are simply set to 0.

   b) Improved features. We show although our edge features do not bring in new information, it helps extrapolation.

E.3.1 Learning Simple Non-Linear Functions

Dataset details. We consider four tasks where the underlying functions are simple non-linear functions $g : \mathbb{R}^d \to \mathbb{R}$. Given an input $\boldsymbol{x} \in \mathbb{R}^d$, the label is computed by $y = g(\boldsymbol{x})$ for all $\boldsymbol{x}$. We consider the following four families of simple functions g.

a) Quadratic functions $g(\boldsymbol{x}) = \boldsymbol{x}^\top A \boldsymbol{x}$. In each dataset, we randomly sample A. In the simplest case where $A = I$, $g(\boldsymbol{x}) = \sum_{i=1}^d x_i^2$.

a) Cosine functions $g(\boldsymbol{x}) = \sum_{i=1}^d \cos(2\pi \cdot \boldsymbol{x}_i)$.

c) Square root functions $g(\boldsymbol{x}) = \sum_{i=1}^d \sqrt{\boldsymbol{x}_i}$. Here, the domain $\mathcal{X}$ of $\boldsymbol{x}$ is restricted to the space in $\mathbb{R}^d$ with non-negative value in each dimension.

d) L1 norm functions $g(\boldsymbol{x}) = |\boldsymbol{x}|_1 = \sum_{i=1}^d |\boldsymbol{x}_i|$.

We sample each dataset of a task by considering the following parameters

a) The shape and support of training, validation, and test data distributions.

   i) Training, validation, and test data are uniformly sampled from a hyper-cube. Training and validation data are sampled from $[-a, a]^d$ with $a \in \{0.5, 1.0\}$, i.e., each dimension of $\boldsymbol{x} \in \mathbb{R}^d$ is uniformly sampled from $[-a, a]$. Test data are sampled from $[-a, a]^d$ with $a \in \{2.0, 5.0, 10.0\}$.

   ii) Training and validation data are uniformly sampled from a sphere, where every point has $L2$ distance r from the origin. We sample r from $r \in \{0.5, 1.0\}$. Then, we sample a random Gaussian vector $\boldsymbol{q}$ in $\mathbb{R}^d$. We obtain the training or validation data $\boldsymbol{x} = \boldsymbol{q}/\|\boldsymbol{q}\|_2 \cdot r$. This corresponds to uniform sampling from the sphere.

   Test data are sampled (non-uniformly) from a hyper-ball. We first sample r uniformly from $[0.0, 2.0]$, $[0.0, 5.0]$, and $[0.0, 10.0]$. Then, we sample a random Gaussian vector $\boldsymbol{q}$ in $\mathbb{R}^d$. We obtain the test data $\boldsymbol{x} = \boldsymbol{q}/\|\boldsymbol{q}\|_2 \cdot r$. This corresponds to (non-uniform) sampling from a hyper-ball in $\mathbb{R}^d$.

b) We sample $20,000$ training data, $1,000$ validation data, and $20,000$ test data.

c) We sample input dimension d from $\{1, 2, 8\}$.

d) For quadratic functions, we sample the entries of A uniformly from $[-1, 1]$.

Model and hyperparameter settings. We consider the multi-layer perceptron (MLP) architecture.

$$\text{MLP}(\boldsymbol{x}) = \boldsymbol{W}^{(d)} \cdot \sigma\left(\boldsymbol{W}^{(d-1)}\sigma\left(...\sigma\left(\boldsymbol{W}^{(1)}\boldsymbol{x}\right)\right)\right)$$

We search the following hyper-parameters for MLPs

a) Number of layers d from $\{2, 4\}$.

b) Width of each $\boldsymbol{W}^{(k)}$ from $\{64, 128, 512\}$.

c) Initialization schemes.

   i) The default initialization in PyTorch.

   ii) The initialization scheme in neural tangent kernel theory, i.e., we sample entries of $\boldsymbol{W}^k$ from $\mathcal{N}(0, 1)$ and scale the output after each $\boldsymbol{W}^{(k)}$ by $\sqrt{\frac{2}{d_k}}$, where d_k is the output dimension of $\boldsymbol{W}^{(k)}$.

d) Activation function σ is set to ReLU.

We train the MLP with the mean squared error (MSE) loss, and Adam and SGD optimizer. We consider the following hyper-parameters for training

a) Initial learning rate from $\{5e-2, 1e-2, 5e-3, 1e-3\}$. Learning rate decays 0.5 for every 50 epochs

b) Batch size from $\{32, 64, 128\}$.

c) Weight decay is set to $1e-5$.

d) Number of epochs is set to 250.

Test error and model selection. For each dataset and architecture, training hyper-parameter setting, we perform model selection via validation set, i.e., we report the test error by selecting the epoch where the model achieves the best validation error. Note that our validation sets always have the same distribution as the training sets.

We train our models with the MSE loss. Because we sample test data from different ranges, the mean absolute percentage error (MAPE) loss, which scales the error by the actual value, better measures the extrapolation performance

$$\text{MAPE} = \frac{1}{n}\left|\frac{A_i - F_i}{A_i}\right|,$$

where A_i is the actual value and F_i is the predicted value. Hence, in our experiments, we also report the MAPE.

E.3.2 R-squared for Out-of-distribution Directions

We perform linear regression to fit the predictions of MLPs along randomly sampled directions in out-of-distribution regions, and compute the R-squared (or R^2) for these directions. This experiment is to validate Theorem 7.1 and show that the convergence rate (to a linear function) is very high in practice.

Definition. R-squared, also known as coefficient of determination, assesses how strong the linear relationship is between input and output variables. The closer R-squared is to 1, the stronger the linear relationship is, with 1 being perfectly linear.

Datasets and models. We perform the R-squared computation on over $2,000$ combinations of datasets, test/train distributions, and hyper-parameters, e.g., learning rate, batch size, MLP layer, width, initialization. These are described in Appendix E.3.1.

Computation. For each combination of dataset and model hyper-parameters as described in Section E.3.1, we save the trained MLP model $f : \mathbb{R}^d \to \mathbb{R}$. For each dataset and model combination, we then randomly sample $5,000$ directions via Gaussian vectors $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$. For each of these directions $\boldsymbol{w}$, we compute the intersection point $\boldsymbol{x}_{\boldsymbol{w}}$ of direction $\boldsymbol{w}$ and the training data distribution support (specified by a hyper-sphere or hyper-cube; see Section E.3.1 for details).

We then collect 100 predictions of the trained MLP f along direction $\boldsymbol{w}$ (assume $\boldsymbol{w}$ is normalized) with

$$\left\{\left(\boldsymbol{x}_{\boldsymbol{w}} + k \cdot \frac{r}{10} \cdot \boldsymbol{w}\right), f\left(\boldsymbol{x}_{\boldsymbol{w}} + k \cdot \frac{r}{10} \cdot \boldsymbol{w}\right)\right\}_{k=0}^{100}, \tag{E.70}$$

where r is the range of training data distribution support (see Section E.3.1). We perform linear regression on these predictions in (E.70), and obtain the R-squared.

Results. We obtain the R-squared for each combination of dataset, model and training setting, and randomly sampled direction. For the tasks of learning the simple non-linear functions, we confirm that more than 96% of the R-squared results are above 0.99. This empirically confirms Theorem 7.1 and shows that the convergence rate is in fact fast in practice. Along most directions, MLP's learned function becomes linear immediately out of the training data support.

E.3.3 Learning Linear Functions

Dataset details. We consider the tasks where the underlying functions are linear $g : \mathbb{R}^d \to \mathbb{R}$. Given an input $\boldsymbol{x} \in \mathbb{R}^d$, the label is computed by $y = g(\boldsymbol{x}) = A\boldsymbol{x}$ for all $\boldsymbol{x}$. For each dataset, we sample the following parameters

a) We sample $10,000$ training data, $1,000$ validation data, and $2,000$ test data.

b) We sample input dimension d from $\{1, 2, 32\}$.

c) We sample entries of A uniformly from $[-a, a]$, where we sample $a \in \{5.0, 10.0\}$.

d) The shape and support of training, validation, and test data distributions.

   i) Training, validation, and test data are uniformly sampled from a hyper-cube. Training and validation data are sampled from $[-a, a]^d$ with $a \in \{5.0, 10.0\}$, i.e., each dimension of $\boldsymbol{x} \in \mathbb{R}^d$ is uniformly sampled from $[-a, a]$. Test data are sampled from $[-a, a]^d$ with $a \in \{20.0, 50.0\}$.

   ii) Training and validation data are uniformly sampled from a sphere, where every point has $L2$ distance r from the origin. We sample r from $r \in \{5.0, 10.0\}$. Then, we sample a random Gaussian vector $\boldsymbol{q}$ in $\mathbb{R}^d$. We obtain the training or validation data $\boldsymbol{x} = \boldsymbol{q}/\|\boldsymbol{q}\|_2 \cdot r$. This corresponds to uniform sampling from the sphere.

Test data are sampled (non-uniformly) from a hyper-ball. We first sample r uniformly from $[0.0, 20.0]$ and $[0.0, 50.0]$,. Then, we sample a random Gaussian vector $\boldsymbol{q}$ in $\mathbb{R}^d$. We obtain the test data $\boldsymbol{x} = \boldsymbol{q}/\|\boldsymbol{q}\|_2 \cdot r$. This corresponds to (non-uniform) sampling from a hyper-ball in $\mathbb{R}^d$.

e) We perform ablation study on how the training distribution support misses directions. The test distributions remain the same as in d).

   i) We restrict the first dimension of any training data $\boldsymbol{x}_i$ to a fixed number 0.1, and randomly sample the remaining dimensions according to d).

   ii) We restrict the first k dimensions of any training data $\boldsymbol{x}_i$ to be positive. For input dimension 32, we only consider the hyper-cube training distribution, where we sample the first k dimensions from $[0, a]$ and sample the remaining dimensions from $[-a, a]$. For input dimensions 1 and 2, we consider both hyper-cube and hyper-sphere training distribution by performing rejection sampling. For input dimension 2, we consider k from $\{1, 2\}$. For input dimension 32, we consider k from $\{1, 16, 32\}$.

   iii) We restrict the first k dimensions of any training data $\boldsymbol{x}_i$ to be negative. For input dimension 32, we only consider the hyper-cube training distribution, where we sample the first k dimensions from $[-a, 0]$ and sample the remaining dimensions from $[-a, a]$. For input dimensions 1 and 2, we consider both hyper-cube and hyper-sphere training distribution by performing rejection sampling. For input dimension 2, we consider k from $\{1, 2\}$. For input dimension 32, we consider k from $\{1, 16, 32\}$.

Model and hyperparameter settings. For the regression task, we search the same set of hyper-parameters as those in simple non-linear functions (Section E.3.1).We report the test error with the same validation procedure as in Section E.3.1.

Exact computation with neural tangent kernel Our experiments with MLPs validate Theorem 7.3 asymptotic extrapolation for neural networks trained in regular regimes. Here, we also validate Lemma 7.2, exact extrapolation with finite data regime, by training an infinitely-wide neural network. That is, we directly perform the kernel regression with the neural tangent kernel (NTK). This experiment is mainly of theoretical interest.

We sample the same test set as in our experiments with MLPs. For training set, we sample $2d$ training examples according to the conditions in Lemma 7.2. Specifically, we first sample an orthogonal basis and their opposite vectors $\boldsymbol{X} = \{\boldsymbol{e}_i, -\boldsymbol{e}_i\}_{i=1}^d$. We then randomly sample 100 orthogonal transform matrices Q via the QR decomposition. Our training samples are $Q\boldsymbol{X}$, i.e., multiply each point in $\boldsymbol{X}$ by Q. This gives 100 training sets with $2d$ data points satisfying the condition in Lemma 7.2.

We perform kernel regression on these training sets using a two-layer neural tangent kernel (NTK). Our code for exact computation of NTK is adapted from Arora et al. [2020], Novak et al. [2020]. We verify that the test losses are all precisely 0, up to machine precision. This empirically confirms Lemma 7.2.

Note that due to the differences in hyper-parameter settings in different implementations of NTK, to reproduce our experiments and achieve zero test error, the implementation by Arora et al. [2020] is assumed.

E.3.4 MLPs with cosine, quadratic, and tanh Activation

This section describes the experimental settings for extrapolation experiments for MLPs with cosine, quadratic, and tanh activation functions. We train MLPs to learn the following functions:

a) Quadratic function $g(\boldsymbol{x}) = \boldsymbol{x}\top A\boldsymbol{x}$, where A is a randomly sampled matrix.

b) Cosine function $g(\boldsymbol{x}) = \sum_{i=1}^d \cos(2\pi \cdot \boldsymbol{x}_i)$.

c) Hyperbolic tangent function $g(\boldsymbol{x}) = \sum_{i=1}^d \tanh(\boldsymbol{x}_i)$.

d) Linear function $g(\boldsymbol{x}) = W\boldsymbol{x} + b$.

Dataset details. We use 20,000 training, 1,000 validation, and 20,000 test data. For quadratic, we sample input dimension d from $\{1, 8\}$, training and validation data from $[-1, 1]^d$, and test data from $[-5, 5]^d$. For cosine, we sample input dimension d from $\{1, 2\}$, training and validation data from $[-100, 100]^d$, and test data from $[-200, 200]^d$. For tanh, we sample input dimension d from $\{1, 8\}$, training and validation data from $[-100, 100]^d$, and test data from $[-200, 200]^d$. For linear, we use a subset of datasets from Appendix E.3.3: 1 and 8 input dimensions with hyper-cube training distributions.

Model and hyperparameter settings. We use the same hyperparameters from Appendix E.3.1, except we fix the batch size to 128, as the batch size has minimal impact on models. MLPs with cos activation is hard to optimize, so we only report models with training MAPE less than 1.

E.3.5 Max Degree

Dataset details. We consider the task of finding the maximum degree on a graph. Given any input graph $G = (V, E)$, the label is computed by the underlying function $y = g(G) = \max\limits_{u \in G} \sum_{v \in \mathcal{N}(u)} 1$. For each dataset, we sample the graphs and node features with the following parameters

a) Graph structure for training and validation sets. For each dataset, we consider one of the following graph structure: path graphs, cycles, ladder graphs, 4-regular random graphs, complete graphs, random trees, expanders (here we use random graphs with $p = 0.8$ as they are expanders with high probability), and general graphs (random graphs with $p = 0.1$ to 0.9 with equal probability for a broad range of graph structure). We use the networkx library for sampling graphs.

b) Graph structure for test set. We consider the general graphs (random graphs with $p = 0.1$ to 0.9 with equal probability).

c) The number of vertices of graphs $|V|$ for training and validation sets are sampled uniformly from $[20...30]$. The number of vertices of graphs $|V|$ for test set is sampled uniformly from $[50..100]$.

d) We consider two schemes for node features.

  i) Identical features. All nodes in training, validation and set sets have uniform feature 1.

  ii) Spurious (continuous) features. Node features in training and validation sets are sampled uniformly from $[-5.0, 5.0]^3$, i.e., a three-dimensional vector where each dimension is sampled from $[-5.0, 5.0]$. There are two schemes for test sets, in the first case we do not extrapolate node features, so we sample node features uniformly from $[-5.0, 5.0]^3$. In the second case we extrapolate node features, we sample node features uniformly from $[-10.0, 10.0]^3$.

e) We sample $5,000$ graphs for training, $1,000$ graphs for validation, and $2,500$ graphs for testing.

Model and hyperparameter settings. We consider the following Graph Neural Network (GNN) architecture. Given an input graph G, GNN learns the output h_G by first iteratively aggregating and transforming the neighbors of all node vectors $h_u^{(k)}$ (vector for node u in layer k), and perform a max or sum-pooling over all node features h_u to obtain h_G. Formally, we have

$$h_u^{(k)} = \sum_{v \in \mathcal{N}(u)} \text{MLP}^{(k)}\left(h_v^{(k-1)}, h_u^{(k-1)}\right), \quad h_G = \text{MLP}^{(K+1)}\left(\text{graph-pooling}\{h_u^{(K)} : u \in G\}\right). \tag{E.71}$$

Here, $\mathcal{N}(u)$ denotes the neighbors of u, K is the number of GNN iterations, and graph-pooling is a hyper-parameter with choices as max or sum. $h_u^{(0)}$ is the input node feature of node u. We search the following hyper-parameters for GNNs

a) Number of GNN iterations K is 1.

b) Graph pooling is from max or sum.

c) Width of all MLPs are set to 256.

d) The number of layers for MLP$^{(k)}$ with $k = 1..K$ are set to 2. The number of layers for MLP$^{(K+1)}$ is set to 1.

We train the GNNs with the mean squared error (MSE) loss, and Adam and SGD optimizer. We search the following hyper-parameters for training

a) Initial learning rate is set to 0.01.

b) Batch size is set to 64.

c) Weight decay is set to $1e-5$.

d) Number of epochs is set to 300 for graphs with continuous node features, and 100 for graphs with uniform node features.

Test error and model selection. For each dataset and architecture, training hyper-parameter setting, we perform model selection via validation set, i.e., we report the test error by selecting the epoch where the model achieves the best validation error. Note that our validation sets always have the same distribution as the training sets. Again, we report the MAPE for test error as in MLPs.

E.3.6 Shortest Path

Dataset details. We consider the task of finding the length of the shortest path on a graph, from a given source to target nodes. Given any graph $G = (V, E)$, the node features, besides regular node features, encode whether a node is source s, and whether a node is target t. The edge features are a scalar representing the edge weight. For unweighted graphs, all edge weights are 1. Then the label $y = g(G)$ is the length of the shortest path from s to t on G.

For each dataset, we sample the graphs and node, edge features with the following parameters

a) Graph structure for training and validation sets. For each dataset, we consider one of the following graph structure: path graphs, cycles, ladder graphs, 4-regular random graphs, complete graphs, random trees, expanders (here we use random graphs with

$p = 0.6$ which are expanders with high probability), and general graphs (random graphs with $p = 0.1$ to 0.9 with equal probability for a broad range of graph structure). We use the networkx library for sampling graphs.

b) Graph structure for test set. We consider the general graphs (random graphs with $p = 0.1$ to 0.9 with equal probability).

c) The number of vertices of graphs $|V|$ for training and validation sets are sampled uniformly from $[20...40]$. The number of vertices of graphs $|V|$ for test set is sampled uniformly from $[50..70]$.

d) We consider the following scheme for node and edge features. All edges have continuous weights. Edge weights for training and validation graphs are sampled from $[1.0, 5.0]$. There are two schemes for test sets, in the first case we do not extrapolate edge weights, so we sample edge weights uniformly from $[1.0, 5.0]$. In the second case we extrapolate edge weights, we sample edge weights uniformly from $[1.0, 10.0]$. All node features are $[h, \mathbb{I}(v = s), \mathbb{I}(v = t)]$ with h sampled from $[-5.0, 5.0]$.

e) After sampling a graph and edge weights, we sample source s and t by randomly sampling s, t and selecting the first pair s, s whose shortest path involves at most 3 hops. This enables us to solve the task using GNNs with 3 iterations.

f) We sample $10,000$ graphs for training, $1,000$ graphs for validation, and $2,500$ graphs for testing.

We also consider the ablation study of training on random graphs with different p. We consider $p = 0.05..1.0$ and report the test error curve. The other parameters are the same as described above.

Model and hyperparameter settings. We consider the following Graph Neural Network (GNN) architecture. Given an input graph G, GNN learns the output h_G by first iteratively aggregating and transforming the neighbors of all node vectors $h_u^{(k)}$ (vector for node u in layer k), and perform a max or sum-pooling over all node features h_u to obtain h_G. Formally,

we have

$$h_u^{(k)} = \min_{v \in \mathcal{N}(u)} \text{MLP}^{(k)} \left(h_v^{(k-1)}, h_u^{(k-1)}, w_{(u,v)} \right), \quad h_G = \text{MLP}^{(K+1)} \left(\min_{u \in G} h_u \right). \qquad \text{(E.72)}$$

Here, $\mathcal{N}(u)$ denotes the neighbors of u, K is the number of GNN iterations, and for neighbor aggregation we run both min and sum. $h_u^{(0)}$ is the input node feature of node u. $w_{(u,v)}$ is the input edge feature of edge (u, v). We search the following hyper-parameters for GNNs

a) Number of GNN iterations K is set to 3.

b) Graph pooling is set to min.

c) Neighobr aggregation is selected from min and sum.

d) Width of all MLPs are set to 256.

e) The number of layers for MLP$^{(k)}$ with $k = 1..K$ are set to 2. The number of layers for MLP$^{(K+1)}$ is set to 1.

We train the GNNs with the mean squared error (MSE) loss, and Adam and SGD optimizer. We consider the following hyper-parameters for training

a) Initial learning rate is set to 0.01.

b) Batch size is set to 64.

c) Weight decay is set to $1e-5$.

d) Number of epochs is set to 250.

We perform the same model selection and validation as in Section E.3.5.

E.3.7 N-Body Problem

Task description. The n-body problem asks a neural network to predict how n stars in a physical system evolves according to physics laws. That is, we train neural networks to predict properties of future states of each star in terms of next frames, e.g., 0.001 seconds.

Mathematically, in an n-body system $S = \{X_i\}_{i=1}^n$, such as solar systems, all n stars $\{X_i\}_{i=1}^n$ exert distance and mass-dependent gravitational forces on each other, so there were $n(n-1)$ relations or forces in the system. Suppose X_i at time t is at position $\boldsymbol{x}_i^t$ and has velocity $\boldsymbol{v}_i^t$. The overall forces a star X_i receives from other stars is determined by physics laws as the following

$$\boldsymbol{F}_i^t = G \cdot \sum_{j \neq i} \frac{m_i \times m_j}{\|\boldsymbol{x}_i^t - \boldsymbol{x}_j^t\|_2^3} \cdot \left(\boldsymbol{x}_j^t - \boldsymbol{x}_i^t\right), \tag{E.73}$$

where G is the gravitational constant, and m_i is the mass of star X_i. Then acceralation $\boldsymbol{a}_i^t$ is determined by the net force $\boldsymbol{F}_i^t$ and the mass of star m_i

$$\boldsymbol{a}_i^t = \boldsymbol{F}_i^t / m_i \tag{E.74}$$

Suppose the velocity of star X_i at time t is $\boldsymbol{v}_i^t$. Then assuming the time steps dt, i.e., difference between time frames, are sufficiently small, the velocity at the next time frame $t+1$ can be approximated by

$$\boldsymbol{v}_i^{t+1} = \boldsymbol{v}_i^t + \boldsymbol{a}_i^t \cdot dt. \tag{E.75}$$

Given m_i, $\boldsymbol{x}_i^t$, and $\boldsymbol{v}_i^t$, our task asks the neural network to predict $\boldsymbol{v}_i^{t+1}$ for all stars X_i. In our task, we consider two extrapolation schemes

a) The distances between stars $\|\boldsymbol{x}_i^t - \boldsymbol{x}_j^t\|_2$ are out-of-distribution for test set, i.e., different sampling ranges from the training set.

b) The masses of stars m_i are out-of-distribution for test set, i.e., different sampling ranges from the training set.

Here, we use a physics engine that we code in Python to simulate and sample the inputs and labels. We describe the dataset details next.

Dataset details. We first describe the simulation and sampling of our training set. We sample 100 videos of n-body system evolution, each with 500 rollout, i.e., time steps. We consider the orbit situation: there exists a huge center star and several other stars. We sample the initial states, i.e., position, velocity, masses, acceleration etc according to the following parameters.

a) The mass of the center star is $100kg$.

b) The masses of other stars are sampled from $[0.02, 9.0]kg$.

c) The number of stars is 3.

d) The initial position of the center star is $(0.0, 0.0)$.

d) The initial positions $\boldsymbol{x}_i^t$ of other objects are randomly sampled from all angles, with a distance in $[10.0, 100.0]m$.

e) The velocity of the center star is $\mathbf{0}$.

f) The velocities of other stars are perpendicular to the gravitational force between the center star and itself. The scale is precisely determined by physics laws to ensure the initial state is an orbit system.

For each video, after we get the initial states, we continue to rollout the next frames according the physics engine described above. We perform rejection sampling of the frames to ensure that all pairwise distances of stars in a frame are at least $30m$. We guarantee that there are $10,000$ data points in the training set.

The validation set has the same sampling and simultation parameters as the training set. We have $2,500$ data points in the validation set.

For test set, we consider two datasets, where we respectively have OOD distances and masses. We have $5,000$ data points for each dataset.

a) We sample the distance OOD test set to ensure all pairwise distances of stars in a frame are from $[1..20]m$, but have in-distribution masses.

b) We sample the mass OOD test set as follows

    i) The mass of the center star is $200kg$, i.e., twice of that in the training set.

ii) The masses of other stars are sampled from $[0.04, 18.0]kg$, compared to $[0.02, 9.0]kg$ in the training set.

iii) The distances are in-distribution, i.e., same sampling process as training set.

Model and hyperparameter settings. We consider the following one-iteration Graph Neural Network (GNN) architecture, a.k.a. Interaction Networks. Given a collection of stars $S = \{X_i\}_{i=1}^n$, our GNN runs on a complete graph with nodes being the stars X_i. GNN learns the star (node) representations by aggregating and transforming the interactions (forces) of all other node vectors

$$o_u = \text{MLP}^{(2)} \left(\sum_{v \in S \setminus \{u\}} \text{MLP}^{(1)} \left(h_v, h_u, w_{(u,v)} \right) \right). \tag{E.76}$$

Here, h_v is the input feature of node v, including mass, position and velocity

$$h_v = (m_v, \boldsymbol{x}_v, \boldsymbol{v}_v)$$

$w_{(u,v)}$ is the input edge feature of edge (u, v). The loss is computed and backpropagated via the MSE loss of

$$\|[o_1, ..., o_n] - [ans_1, .., ans_n]\|_2,$$

where o_i denotes the output of GNN for node i, and ans_i denotes the true label for node i in the next frame.

We search the following hyper-parameters for GNNs

a) Number of GNN iterations is set to 1.

b) Width of all MLPs are set to 128.

c) The number of layers for MLP$^{(1)}$ is set to 4. The number of layers for MLP$^{(2)}$ is set to 2.

d) We consider *two representations* of edge/relations $w_{(i,j)}$.

i) The first one is simply 0.

ii) The better representation, which makes the underlying target function more linear, is

$$w_{(i,j)} = \frac{m_j}{\|\boldsymbol{x}_i^t - \boldsymbol{x}_j^t\|_2^3} \cdot \left(\boldsymbol{x}_j^t - \boldsymbol{x}_i^t\right)$$

We train the GNN with the mean squared error (MSE) loss, and Adam optimizer. We search the following hyper-parameters for training

a) Initial learning rate is set to 0.005. learning rate decays 0.5 for every 50 epochs

b) Batch size is set to 32.

c) Weight decay is set to $1e-5$.

d) Number of epochs is set to $2,000$.

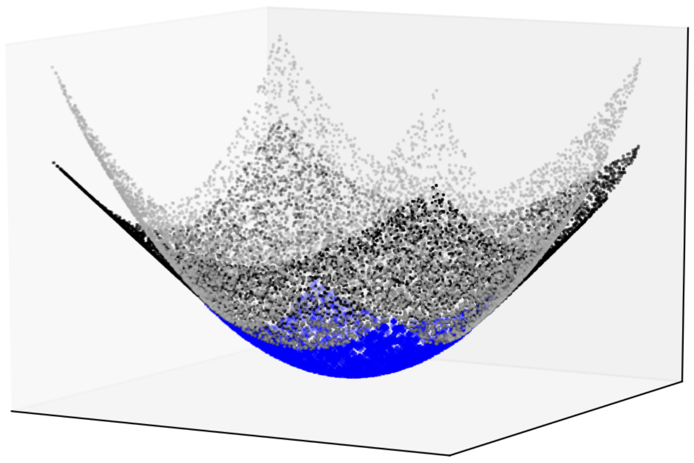
E.4 Visualization and Additional Experimental Results

E.4.1 Visualization Results

In this section, we show additional visualization results of the MLP's learned function out of training distribution (in **black color**) v.s. the underlying true function (in **grey color**). We color the predictions in training distribution in **blue color**.

In general, MLP's learned functions agree with the underlying true functions in training range (blue). This is explained by in-distribution generalization arguments. When out of distribution, the MLP's learned functions become linear along directions from the origin. We explain this OOD directional linearity behavior in Theorem 7.1.

Finally, we show additional experimental results for graph-based reasoning tasks.



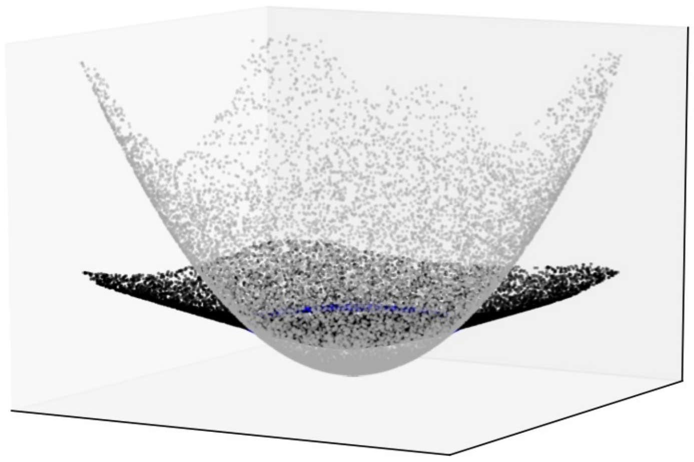
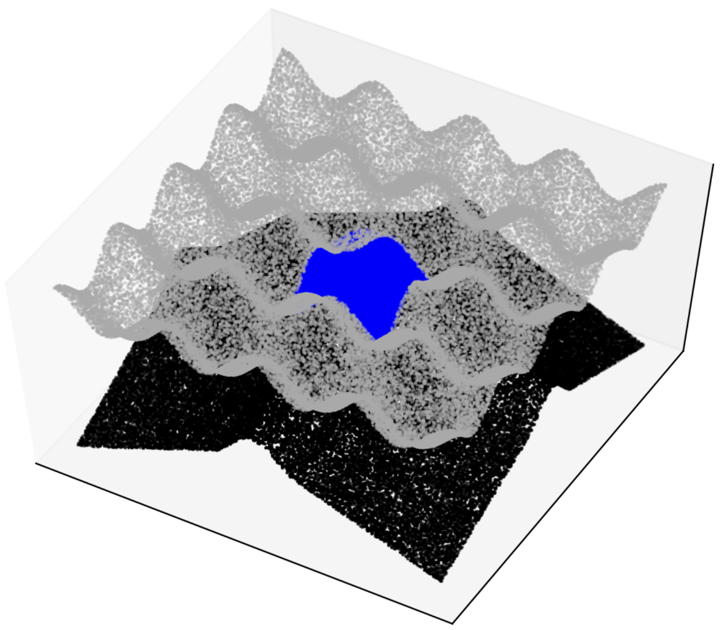


Figure E-1: **(Quadratic function).** Both panels show the learned v.s. true $y = x_1^2 + x_2^2$. In each figure, we color OOD predictions by MLPs in black, underlying function in grey, and in-distribution predictions in blue. The support of training distribution is a square (cube) for the top panel, and is a circle (sphere) for the bottom panel.



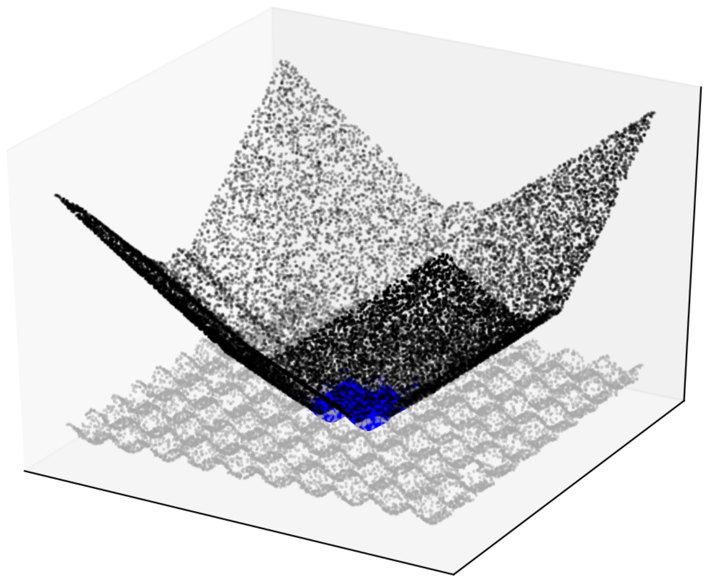
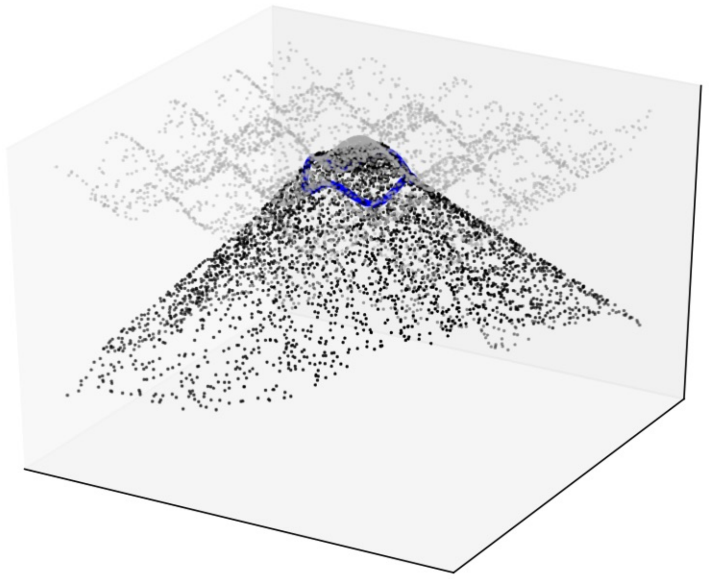


Figure E-2: **(Cos function).** Both panels show the learned v.s. true $y = \cos(2\pi \cdot x_1) + \cos(2\pi \cdot x_2)$. In each figure, we color OOD predictions by MLPs in black, underlying function in grey, and in-distribution predictions in blue. The support of training distribution is a square (cube) for both top and bottom panels, but with different ranges.



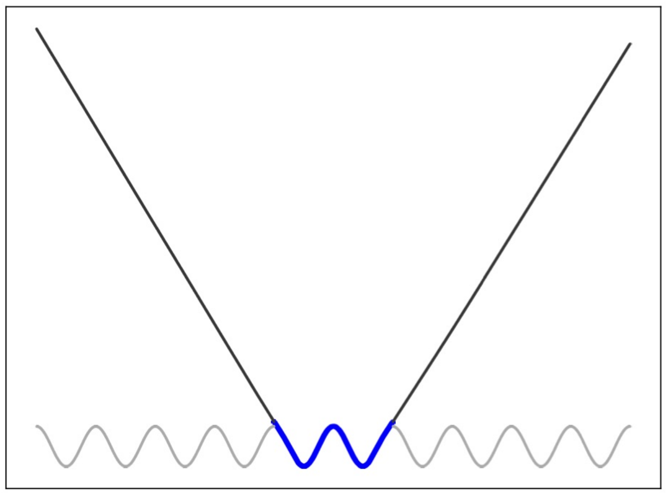
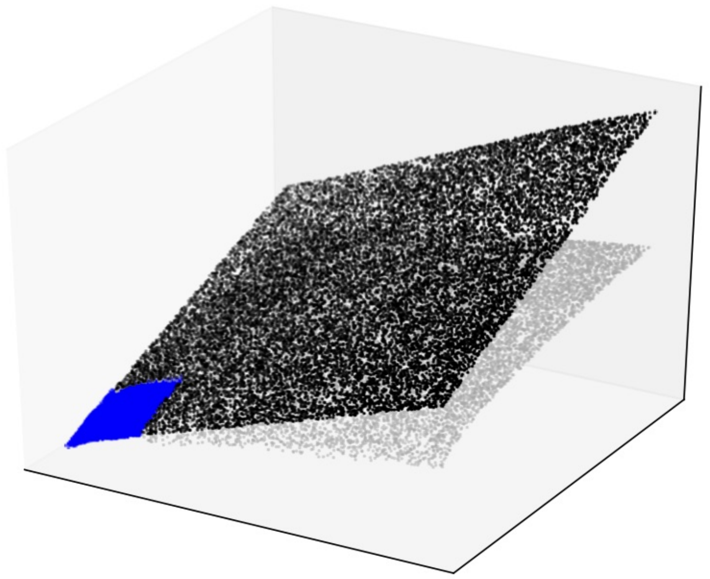


Figure E-3: **(Cos function).** Top panel shows the learned v.s. true $y = \cos(2\pi \cdot x_1) + \cos(2\pi \cdot x_2)$ where the support of training distribution is a circle (sphere). Bottom panel shows results for cosine in 1D, i.e. $y = \cos(2\pi \cdot x)$. In each figure, we color OOD predictions by MLPs in black, underlying function in grey, and in-distribution predictions in blue.



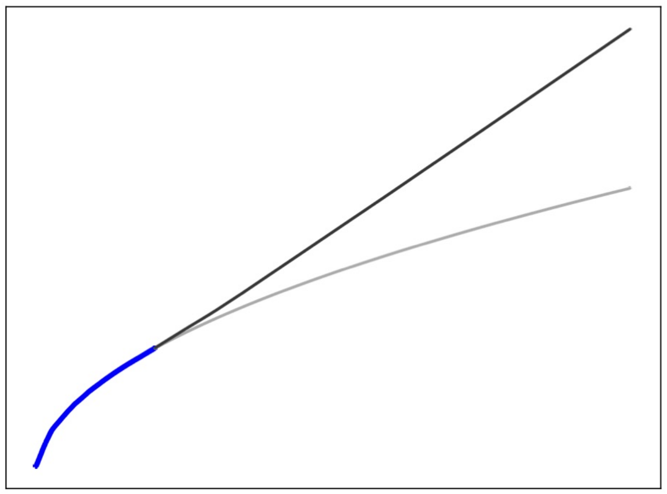


Figure E-4: **(Sqrt function).** Top panel shows the learned v.s. true $y = \sqrt{x_1} + \sqrt{x_2}$ where the support of training distribution is a square (cube). Bottom panel shows the results for the square root function in 1D, i.e. $y = \sqrt{x}$. In each figure, we color OOD predictions by MLPs in black, underlying function in grey, and in-distribution predictions in blue.

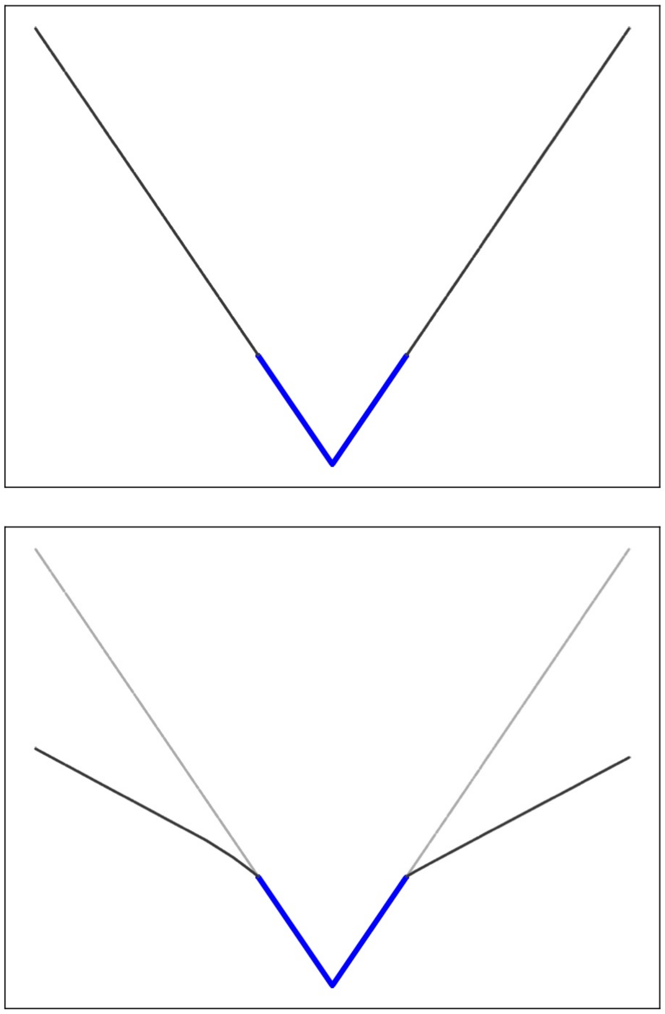


Figure E-5: **(L1 function).** Both panels show the learned v.s. true $y = |x|$. In the top panel, the MLP successfully learns to extrapolate the absolute function. In the bottom panel, an MLP with different hyper-parameters fails to extrapolate. In each figure, we color OOD predictions by MLPs in black, underlying function in grey, and in-distribution predictions in blue.

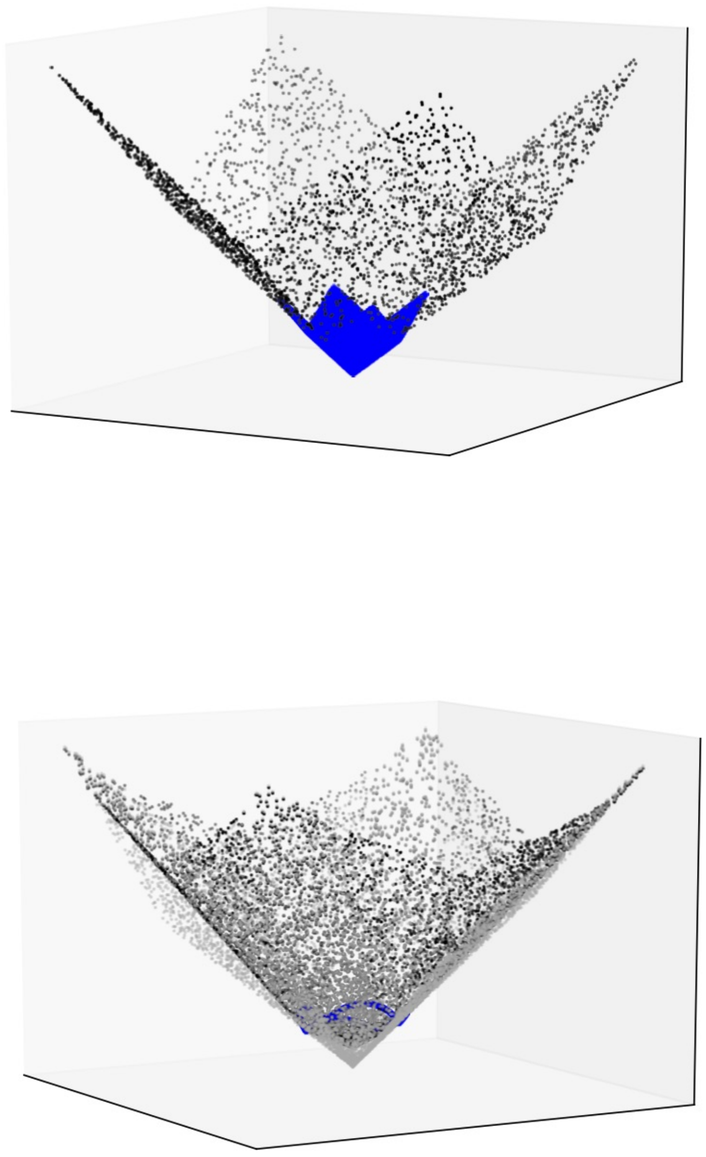


Figure E-6: **(L1 function).** Both panels show the learned v.s. true $y = |x_1| + |x_2|$. In the top panel, the MLP successfully learns to extrapolate the l1 norm function. In the bottom panel, an MLP with different hyper-parameters fails to extrapolate. In each figure, we color OOD predictions by MLPs in black, underlying function in grey, and in-distribution predictions in blue.

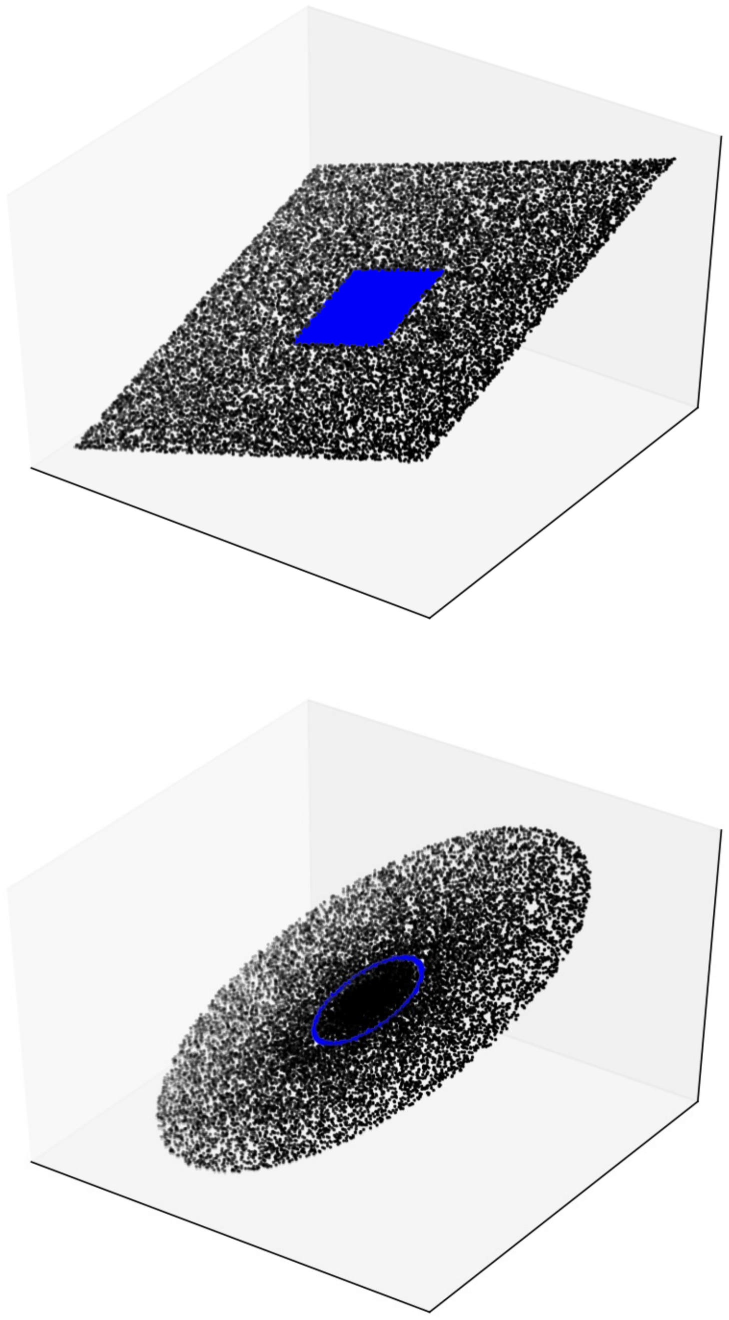


Figure E-7: **(Linear function).** Both panels show the learned v.s. true $y = x_1 + x_2$, with the support of training distributions being square (cube) for top panel, and circle (sphere) for bottom panel. MLPs successfully extrapolate the linear function with both training distributions. This is explained by Theorem 7.3: both sphere and cube intersect all directions. In each figure, we color OOD predictions by MLPs in black, underlying function in grey, and in-distribution predictions in blue.

E.4.2 Extra Experimental Results

In this section, we show additional experimental results.

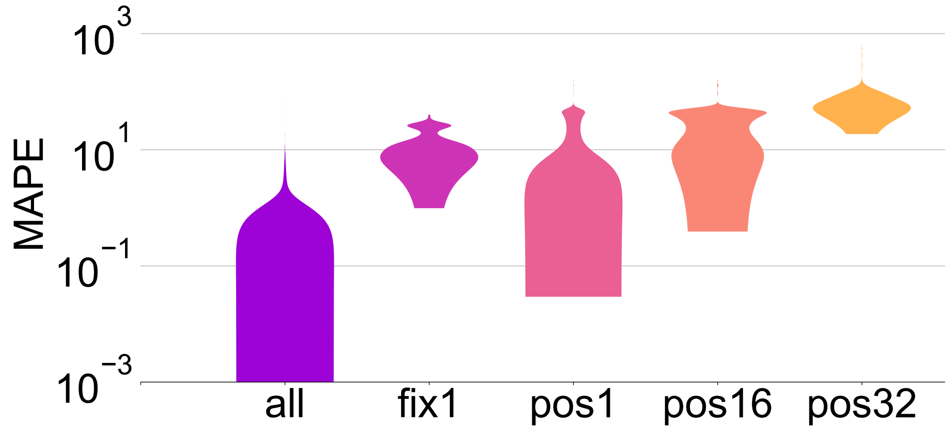


Figure E-8: **Density plot of the test errors in MAPE.** The underlying functions are linear, but we train MLPs on different distributions, whose support potentially miss some directions. The training support for "all" are hyper-cubes that intersect all directions. In "fix1", we set the first dimension of training data to a fixed number. In "posX", we restrict the first X dimensions of training data to be positive. We can see that MLPs trained on "all" extrapolate the underlying linear functions, but MLPs trained on datasets with missing directions, i.e., "fix1" and "posX", often cannot extrapolate well.

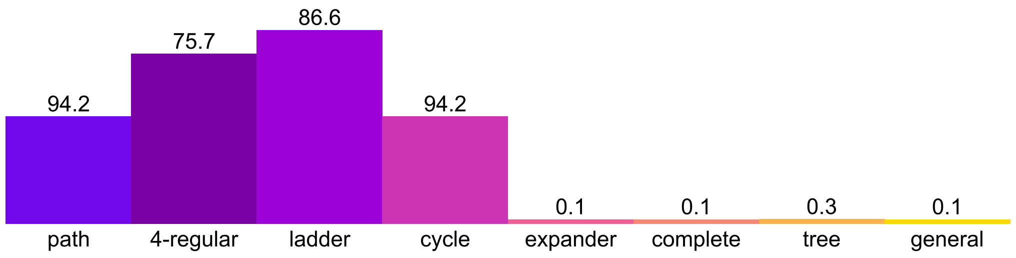


Figure E-9: **Maximum degree: continuous and "spurious" node features.** Here, each node has a node feature in $\mathbb{R}^3$ that shall not contribute to the answer of maximum degree. GNNs with graph-level max-pooling extrapolate to graphs with OOD node features and graph structure, graph sizes, if trained on graphs that satisfy the condition in Theorem 8.2.

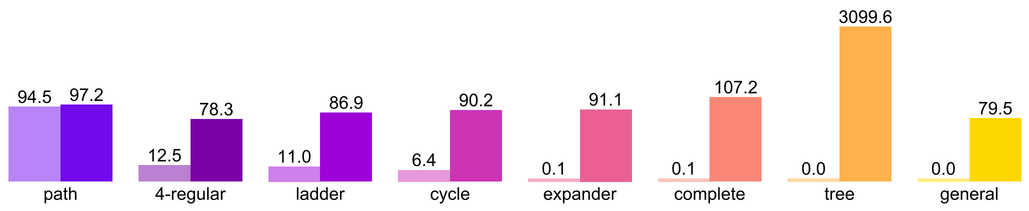


Figure E-10: **Maximum degree: max-pooling v.s. sum-pooling.** In each sub-figure, left column shows test errors for GNNs with graph-level max-pooling; right column shows test errors for GNNs with graph-level sum-pooling. x-axis shows the graph structure covered in training set. GNNs with sum-pooling fail to extrapolate, validating Corollary 8.1. GNNs with max-pooling encodes appropriate non-linear operations, and thus extrapolates under appropriate training sets (Theorem 8.2).

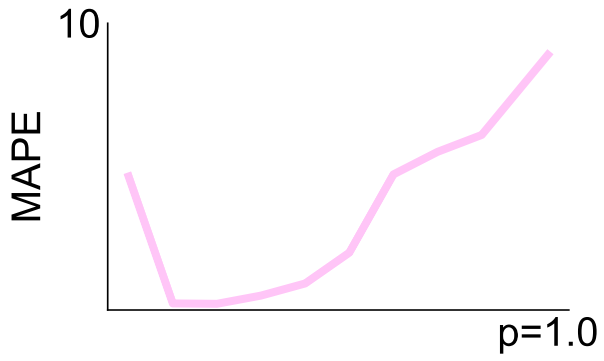


Figure E-11: **Shortest path: random graphs.** We train GNNs with neighbor and graph-level min-pooling on training sets whose graphs are random graphs with probability p of an edge between any two vertices. x-axis denotes the p for the training set, and y-axis denotes the test/extrapolation error on unseen graphs. The test errors follow a U-shape: errors are high if the training graphs are very sparse (small p) or dense (large p). The same pattern is obtained if we train on specific graph structure.

Appendix F

Convergence and Implicit Acceleration

F.1 Proofs

In this section, we complete the proofs of our theoretical results. We show the proofs of Theorem 9.1 in Appendix F.1.1, Proposition 9.2 in Appendix F.1.2, Proposition 9.3 in Appendix F.1.3, Theorem 9.4 in Appendix F.1.4, Proposition 9.5 in Appendix F.1.5, and Theorem 9.6 in Appendix F.1.6.

Before starting our proofs, we first introduce additional notation used in the proofs. We define the corner cases on the products of B as:

$$B_{(H)}B_{(H-1)}\cdots B_{(l+1)} := I_{m_l} \quad \text{if } H = l \tag{F.1}$$

$$B_{(H)}B_{(H-1)}\dots B_{(1)} := I_{m_x} \quad \text{if } H = 0 \tag{F.2}$$

$$B_{(l-1)}B_{(l-2)}\dots B_{(1)} := I_{m_x} \quad \text{if } l = 1 \tag{F.3}$$

Similarly, for any matrices $M_{(l)}$, we define $M_{(l)}M_{(l-1)}\cdots M_{(k)} := I_m$ if $l < k$, and $M_{(l)}M_{(l-1)}\cdots M_{(k)} := M_{(k)} = M_{(l)}$ if $l = k$. Given a scalar-valued variable $a \in \mathbb{R}$ and a matrix $M \in \mathbb{R}^{d\times d'}$, we define

$$\frac{\partial a}{\partial M} = \begin{bmatrix} \frac{\partial a}{\partial M_{11}} & \cdots & \frac{\partial a}{\partial M_{1d'}} \\ \vdots & \ddots & \vdots \\ \frac{\partial a}{\partial M_{d1}} & \cdots & \frac{\partial a}{\partial M_{dd'}} \end{bmatrix} \in \mathbb{R}^{d\times d'}, \tag{F.4}$$

where M_{ij} represents the (i, j)-th entry of the matrix M. Given a vector-valued variable $a \in \mathbb{R}^d$ and a column vector $b \in \mathbb{R}^{d'}$, we let

$$\frac{\partial a}{\partial b} = \begin{bmatrix} \frac{\partial a_1}{\partial b_1} & \cdots & \frac{\partial a_1}{\partial b_{d'}} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_d}{\partial b_1} & \cdots & \frac{\partial a_d}{\partial b_{d'}} \end{bmatrix} \in \mathbb{R}^{d \times d'}, \tag{F.5}$$

where b_i represents the i-th entry of the column vector b. Similarly, given a vector-valued variable $a \in \mathbb{R}^d$ and a row vector $b \in \mathbb{R}^{1 \times d'}$, we write

$$\frac{\partial a}{\partial b} = \begin{bmatrix} \frac{\partial a_1}{\partial b_{11}} & \cdots & \frac{\partial a_1}{\partial b_{1d'}} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_d}{\partial b_{11}} & \cdots & \frac{\partial a_d}{\partial b_{1d'}} \end{bmatrix} \in \mathbb{R}^{d \times d'}, \tag{F.6}$$

where b_{1i} represents the i-th entry of the row vector b. Finally, we recall the definition of the Kronecker product product of two matrices: for matrices $M \in \mathbb{R}^{d_M \times d'_M}$ and $\bar{M} \in \mathbb{R}^{d_{\bar{M}} \times d'_{\bar{M}}}$,

$$M \otimes \bar{M} = \begin{bmatrix} M_{11}\bar{M} & \cdots & M_{1d'_M}\bar{M} \\ \vdots & \ddots & \vdots \\ M_{d_M 1}\bar{M} & \cdots & M_{d_M d'_M}\bar{M} \end{bmatrix} \in \mathbb{R}^{d_M d_{\bar{M}} \times d'_M d'_{\bar{M}}}. \tag{F.7}$$

F.1.1 Proof of Theorem 9.1

We begin with a proof overview of Theorem 9.1. We first relate the gradients $\nabla_{W_{(H)}} L$ and $\nabla_{B_{(l)}} L$ to the gradient $\nabla_{(H)} L$, which is defined by

$$\nabla_{(H)} L(W, B) := \frac{\partial L(W, B)}{\partial \hat{Y}} (X(S^H)_{*\mathcal{I}})^\top \in \mathbb{R}^{m_y \times m_x}.$$

Using the proven relation of $(\nabla_{W_{(H)}} L, \nabla_{B_{(l)}} L)$ and $\nabla_{(H)} L$, we first analyze the dynamics induced in the space of $W_{(l)} B_{(l)} B_{(l-1)} \cdots B_{(1)}$ in Appendix F.1.1, and then the dynamics induced int the space of loss value $L(W, B)$ in Appendix F.1.1. Finally, we complete the proof by using the assumption of employing the square loss in Appendix F.1.1.

Let $W_{(H)} = W$ (during the proof of Theorem 9.1). We first prove the relationship of the gradients $\nabla_{W_{(H)}} L$, $\nabla_{B_{(l)}} L$ and $\nabla_{(H)} L$ in the following lemma:

Lemma F.1. *Let f be an H-layer linear GNN and $\ell(q, Y) = \|q - Y\|_F^2$ where $q, Y \in \mathbb{R}^{m_y \times \bar{n}}$. Then, for any (W, B),*

$$\nabla_{W_{(H)}} L(W, B) = \nabla_{(H)} L(W, B)(B_{(H)} B_{(H-1)} \dots B_{(1)})^\top \in \mathbb{R}^{m_y \times m_l}, \tag{F.8}$$

and

$$\nabla_{B_{(l)}} L(W, B) = (W_{(H)} B_{(H)} B_{(H-1)} \cdots B_{(l+1)})^\top \nabla_{(H)} L(W, B)(B_{(l-1)} B_{(l-2)} \dots B_{(1)})^\top \in \mathbb{R}^{m_l \times m_{l-1}}, \tag{F.9}$$

Proof of Lemma F.1. From Definition 9.1, we have $\hat{Y} = f(X, W, B)_{*\mathcal{I}} = W_{(H)}(X_{(H)})_{*\mathcal{I}}$ where $X_{(l)} = B_{(l)} X_{(l-1)} S$. Using this definition, we can derive the formula of $\frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[W_{(H)}]} \in \mathbb{R}^{m_y n \times m_y m_{\bar{H}}}$ as:

$$\begin{aligned}\frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[W_{(H)}]} &= \frac{\partial}{\partial \operatorname{vec}[W_{(H)}]} \operatorname{vec}[W_{(H)}(X_{(H)})_{*\mathcal{I}}] \\ &= \frac{\partial}{\partial \operatorname{vec}[W_{(H)}]} [((X_{(H)})_{*\mathcal{I}})^\top \otimes I_{m_y}] \operatorname{vec}[W_{(H)}] = [((X_{(H)})_{*\mathcal{I}})^\top \otimes I_{m_y}] \in \mathbb{R}^{m_y n \times m_y m_{\bar{H}}}\end{aligned} \tag{F.10}$$

We will now derive the formula of $\frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[B_{(l)}]} \in \mathbb{R}^{m_y n \times m_l m_{l-1}}$:

$$\begin{aligned}
\frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[B_{(l)}]} &= \frac{\partial}{\partial \operatorname{vec}[B_{(l)}]} \operatorname{vec}[W_{(H)}(X_{(H)})_{*\mathcal{I}}] \\
&= \frac{\partial}{\partial \operatorname{vec}[B_{(l)}]} [I_n \otimes W_{(H)}] \operatorname{vec}[(X_{(H)})_{*\mathcal{I}}] \\
&= [I_n \otimes W_{(H)}] \frac{\partial \operatorname{vec}[(X_{(H)})_{*\mathcal{I}}]}{\partial \operatorname{vec}[B_{(l)}]} \\
&= [I_n \otimes W_{(H)}] \frac{\partial \operatorname{vec}[(X_{(H)})_{*\mathcal{I}}]}{\partial \operatorname{vec}[X_{(l)}]} \frac{\partial \operatorname{vec}[X_{(l)}]}{\partial \operatorname{vec}[B_{(l)}]} \\
&= [I_n \otimes W_{(H)}] \frac{\partial \operatorname{vec}[(X_{(H)})_{*\mathcal{I}}]}{\partial \operatorname{vec}[X_{(l)}]} \frac{\partial \operatorname{vec}[B_{(l)} X_{(l-1)} S]}{\partial \operatorname{vec}[B_{(l)}]} \\
&= [I_n \otimes W_{(H)}] \frac{\partial \operatorname{vec}[(X_{(H)})_{*\mathcal{I}}]}{\partial \operatorname{vec}[X_{(l)}]} \frac{\partial [(X_{(l-1)} S)^\top \otimes I_{m_l}] \operatorname{vec}[B_{(l)}]}{\partial \operatorname{vec}[B_{(l)}]} \\
&= [I_n \otimes W_{(H)}] \frac{\partial \operatorname{vec}[(X_{(H)})_{*\mathcal{I}}]}{\partial \operatorname{vec}[X_{(l)}]} [(X_{(l-1)} S)^\top \otimes I_{m_l}] \qquad \text{(F.11)}
\end{aligned}$$

Here, we have that

$$\operatorname{vec}[(X_{(H)})_{*\mathcal{I}}] = \operatorname{vec}[B_{(H)} X_{(H-1)} S_{*\mathcal{I}}] = \operatorname{vec}[(S^\top)_{\mathcal{I}*} \otimes B_{(H)}] \operatorname{vec}[X_{(H-1)}]. \qquad \text{(F.12)}$$

and

$$\operatorname{vec}[X_{(H)}] = \operatorname{vec}[B_{(H)} X_{(H-1)} S_{*\mathcal{I}}] = \operatorname{vec}[S \otimes B_{(H)}] \operatorname{vec}[X_{(H-1)}]. \qquad \text{(F.13)}$$

By recursively applying (F.13), we have that

$$\begin{aligned}
\operatorname{vec}[(X_{(H)})_{*\mathcal{I}}] &= \operatorname{vec}[(S^\top)_{\mathcal{I}*} \otimes B_{(H)}] \operatorname{vec}[S^\top \otimes B_{(H-1)}] \cdots \operatorname{vec}[S^\top \otimes B_{(l+1)}] \operatorname{vec}[X_{(l)}] \\
&= \operatorname{vec}[((S^{H-l})^\top)_{\mathcal{I}*} \otimes B_{(H)} B_{(H-1)} \cdots B_{(l+1)}] \operatorname{vec}[X_{(l)}],
\end{aligned}$$

where

$$B_{(H)} B_{(H-1)} \cdots B_{(l+1)} := I_{m_l} \quad \text{if } H = l.$$

Therefore,

$$\frac{\partial \operatorname{vec}[(X_{(H)})_{*\mathcal{I}}]}{\partial \operatorname{vec}[X_{(l)}]} = \operatorname{vec}[((S^{H-l})^\top)_{\mathcal{I}*} \otimes B_{(H)}B_{(H-1)} \cdots B_{(l+1)}]. \tag{F.14}$$

Combining (F.11) and (F.14) yields

$$\begin{aligned}
\frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[B_{(l)}]} &= [I_n \otimes W_{(H)}] \frac{\partial \operatorname{vec}[(X_{(H)})_{*\mathcal{I}}]}{\partial \operatorname{vec}[X_{(l)}]} [(X_{(l-1)}S)^\top \otimes I_{m_l}] \\
&= [I_n \otimes W_{(H)}] \operatorname{vec}[((S^{H-l})^\top)_{\mathcal{I}*} \otimes B_{(H)}B_{(H-1)} \cdots B_{(l+1)}][(X_{(l-1)}S)^\top \otimes I_{m_l}] \\
&= [(X_{(l-1)}(S^{H-l+1})_{*\mathcal{I}})^\top \otimes W_{(H)}B_{(H)}B_{(H-1)} \cdots B_{(l+1)}] \in \mathbb{R}^{m_y n \times m_l m_{l-1}}.
\end{aligned} \tag{F.15}$$

Using (F.10), we will now derive the formula of $\nabla_{W_{(H)}} L(W,B) \in \mathbb{R}^{m_y \times m_H}$:

$$\frac{\partial L(W,B)}{\partial \operatorname{vec}[W_{(H)}]} = \frac{\partial L(W,B)}{\partial \operatorname{vec}[\hat{Y}]} \frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[W_{(H)}]} = \frac{\partial L(W,B)}{\partial \operatorname{vec}[\hat{Y}]} [(X_{(H)})_{*\mathcal{I}}^\top \otimes I_{m_y}]$$

Thus, with $\frac{\partial L(W,B)}{\partial \hat{Y}} \in \mathbb{R}^{m_y \times n}$,

$$\begin{aligned}
\nabla_{\operatorname{vec}[W_{(H)}]} L(W,B) &= \left(\frac{\partial L(W,B)}{\partial \operatorname{vec}[W_{(H)}]}\right)^\top \\
&= [(X_{(H)})_{*\mathcal{I}} \otimes I_{m_y}] \left(\frac{\partial L(W,B)}{\partial \operatorname{vec}[\hat{Y}]}\right)^\top \\
&= [(X_{(H)})_{*\mathcal{I}} \otimes I_{m_y}] \operatorname{vec}\left[\frac{\partial L(W,B)}{\partial \hat{Y}}\right] \\
&= \operatorname{vec}\left[\frac{\partial L(W,B)}{\partial \hat{Y}} (X_{(H)})_{*\mathcal{I}}^\top\right] \in \mathbb{R}^{m_y m_H}.
\end{aligned}$$

Therefore,

$$\nabla_{W_{(H)}} L(W,B) = \frac{\partial L(W,B)}{\partial \hat{Y}} (X_{(H)})_{*\mathcal{I}}^\top \in \mathbb{R}^{m_y \times m_H}. \tag{F.16}$$

Using (F.15), we will now derive the formula of $\nabla_{B_{(l)}} L(W, B) \in \mathbb{R}^{m_l \times m_{l-1}}$:

$$\frac{\partial L(W,B)}{\partial \operatorname{vec}[B_{(l)}]} = \frac{\partial L(W,B)}{\partial \operatorname{vec}[\hat{Y}]} \frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[B_{(l)}]} = \frac{\partial L(W,B)}{\partial \operatorname{vec}[\hat{Y}]} [(X_{(l-1)}(S^{H-l+1})_{*\mathcal{I}})^\top \otimes W_{(H)} B_{(H)} B_{(H-1)} \cdots B_{(l+1)}].$$

Thus, with $\frac{\partial L(W,B)}{\partial \hat{Y}} \in \mathbb{R}^{m_y \times n}$,

$$\begin{aligned}
\nabla_{\operatorname{vec}[B_{(l)}]} L(W,B) &= \left(\frac{\partial L(W,B)}{\partial \operatorname{vec}[B_{(l)}]}\right)^\top \\
&= [X_{(l-1)}(S^{H-l+1})_{*\mathcal{I}} \otimes (W_{(H)} B_{(H)} B_{(H-1)} \cdots B_{(l+1)})^\top] \left(\frac{\partial L(W,B)}{\partial \operatorname{vec}[\hat{Y}]}\right)^\top \\
&= [X_{(l-1)}(S^{H-l+1})_{*\mathcal{I}} \otimes (W_{(H)} B_{(H)} B_{(H-1)} \cdots B_{(l+1)})^\top] \operatorname{vec}\left[\frac{\partial L(W,B)}{\partial \hat{Y}}\right] \\
&= \operatorname{vec}\left[(W_{(H)} B_{(H)} B_{(H-1)} \cdots B_{(l+1)})^\top \frac{\partial L(W,B)}{\partial \hat{Y}} (X_{(l-1)}(S^{H-l+1})_{*\mathcal{I}})^\top\right] \in \mathbb{R}^{m_l m_{l-1}}.
\end{aligned}$$

Therefore,

$$\nabla_{B_{(l)}} L(W,B) = (W_{(H)} B_{(H)} B_{(H-1)} \cdots B_{(l+1)})^\top \frac{\partial L(W,B)}{\partial \hat{Y}} (X_{(l-1)}(S^{H-l+1})_{*\mathcal{I}})^\top \in \mathbb{R}^{m_l \times m_{l-1}}. \tag{F.17}$$

With (F.16) and (F.17), we are now ready to prove the statement of this lemma by introducing the following notation:

$$\nabla_{(l)} L(W,B) := \frac{\partial L(W,B)}{\partial \hat{Y}} (X(S^l)_{*\mathcal{I}})^\top \in \mathbb{R}^{m_y \times m_x}.$$

Using this notation along with (F.16)

$$\begin{aligned}
\nabla_{W_{(H)}} L(W,B) &= \frac{\partial L(W,B)}{\partial \hat{Y}} (X_{(H)})_{*\mathcal{I}}^\top \\
&= \frac{\partial L(W,B)}{\partial \hat{Y}} (B_{(H)} X_{(H-1)} (S)_{*\mathcal{I}})^\top \\
&= \frac{\partial L(W,B)}{\partial \hat{Y}} (B_{(H)} B_{(H-1)} \dots B_{(1)} X (S^H)_{*\mathcal{I}})^\top \\
&= \nabla_{(H)} L(W,B) (B_{(H)} B_{(H-1)} \dots B_{(1)})^\top,
\end{aligned}$$

Similarly, using (F.17),

$$
\begin{aligned}
\nabla_{B_{(l)}} L(W,B) &= (W_{(H)}B_{(H)}B_{(H-1)}\cdots B_{(l+1)})^\top \frac{\partial L(W,B)}{\partial \hat{Y}} (X_{(l-1)}(S^{H-l+1})_{*\mathcal{I}})^\top \\
&= (W_{(H)}B_{(H)}B_{(H-1)}\cdots B_{(l+1)})^\top \frac{\partial L(W,B)}{\partial \hat{Y}} (B_{(l-1)}B_{(l-2)}\dots B_{(1)}X(S^{l-1}S^{H-l+1})_{*\mathcal{I}})^\top \\
&= (W_{(H)}B_{(H)}B_{(H-1)}\cdots B_{(l+1)})^\top \frac{\partial L(W,B)}{\partial \hat{Y}} (B_{(l-1)}B_{(l-2)}\dots B_{(1)}X(S^{H})_{*\mathcal{I}})^\top \\
&= (W_{(H)}B_{(H)}B_{(H-1)}\cdots B_{(l+1)})^\top \nabla_{(H)} L(W,B)(B_{(l-1)}B_{(l-2)}\dots B_{(1)})^\top
\end{aligned}
$$

where $B_{(l-1)}B_{(l-2)}\dots B_{(1)} := I_{m_x}$ if $l=1$.

$\square$

By using Lemma F.1, we complete the proof of Theorem 9.1 in the following.

Dynamics induced in the space of $W_{(l)}B_{(l)}B_{(l-1)}\cdots B_{(1)}$

We now consider the dynamics induced in the space of $W_{(l)}B_{(l)}B_{(l-1)}\cdots B_{(1)}$. We first consider the following discrete version of the dynamics:

$$W'_{(H)} = W_{(H)} - \alpha \nabla_{W_{(H)}} L(W,B)$$

$$B'_{(l)} = B_{(l)} - \alpha \nabla_{B_{(l)}} L(W,B).$$

This dynamics induces the following dynamics:

$$W'_{(H)}B'_{(H)}B'_{(H-1)}\cdots B'_{(1)} = (W_{(H)} - \alpha \nabla_{W_{(H)}} L(W,B))(B_{(H)} - \alpha \nabla_{B_{(H)}} L(W,B))\cdots(B_{(1)} - \alpha \nabla_{B_{(1)}} L(W,B)).$$

Define

$$Z_{(H)} := W_{(H)}B_{(H)}B_{(H-1)}\cdots B_{(1)},$$

and

$$Z'_{(H)} := W'_{(H)}B'_{(H)}B'_{(H-1)}\cdots B'_{(1)}.$$

Then, we can rewrite

$$Z'_{(H)} = (W_{(H)} - \alpha \nabla_{W_{(H)}} L(W,B))(B_{(H)} - \alpha \nabla_{B_{(H)}} L(W,B)) \cdots (B_{(1)} - \alpha \nabla_{B_{(1)}} L(W,B)).$$

By expanding the multiplications, this can be written as:

$$\begin{aligned} Z'_{(H)} =& Z_{(H)} - \alpha \nabla_{W_{(H)}} L(W,B) B_{(H)} \cdots B_{(1)} - \\ & \alpha \sum_{i=1}^{H} W_{(H)} B_{(H)} \cdots B_{(i+1)} \nabla_{B_{(i)}} L(W,B) B_{(i-1)} \cdots B_{(1)} + O(\alpha^2). \end{aligned}$$

By vectorizing both sides,

$$\begin{aligned} & \operatorname{vec}[Z'_{(H)}] - \operatorname{vec}[Z_{(H)}] \\ =& - \alpha \operatorname{vec}[\nabla_{W_{(H)}} L(W,B) B_{(H)} \cdots B_{(1)}] \\ & - \alpha \sum_{i=1}^{H} \operatorname{vec}[W_{(H)} B_{(H)} \cdots B_{(i+1)} \nabla_{B_{(i)}} L(W,B) B_{(i-1)} \cdots B_{(1)}] + O(\alpha^2). \end{aligned}$$

Here, using the formula of $\nabla_{W_{(H)}} L(W,B)$ and $\nabla_{B_{(H)}} L(W,B)$, we have that

$$\begin{aligned} \operatorname{vec}[\nabla_{W_{(H)}} L(W,B) B_{(H)} \cdots B_{(1)}] &= \operatorname{vec}[\nabla_{(H)} L(W,B) (B_{(H)} \ldots B_{(1)})^\top B_{(H)} \cdots B_{(1)}] \\ &= [(B_{(H)} \ldots B_{(1)})^\top B_{(H)} \cdots B_{(1)} \otimes I_{m_y}] \operatorname{vec}[\nabla_{(H)} L(W,B)], \end{aligned}$$

and

$$\begin{aligned} & \sum_{i=1}^{H} \operatorname{vec}[W_{(H)} B_{(H)} \cdots B_{(i+1)} \nabla_{B_{(i)}} L(W,B) B_{(i-1)} \cdots B_{(1)}] \\ &= \sum_{i=1}^{H} \operatorname{vec}\left[W_{(H)} B_{(H)} \cdots B_{(i+1)} (W_{(H)} B_{(H)} \cdots B_{(i+1)})^\top \nabla_{(H)} L(W,B) (B_{(i-1)} \ldots B_{(1)})^\top B_{(i-1)} \cdots B_{(1)}\right] \\ &= \sum_{i=1}^{H} [(B_{(i-1)} \ldots B_{(1)})^\top B_{(i-1)} \cdots B_{(1)} \otimes W_{(H)} B_{(H)} \cdots B_{(i+1)} (W_{(H)} B_{(H)} \cdots B_{(i+1)})^\top] \operatorname{vec}\left[\nabla_{(H)} L(W,B)\right]. \end{aligned}$$

Summarizing above,

$$\begin{aligned}
&\operatorname{vec}[Z'_{(H)}] - \operatorname{vec}[Z_{(H)}] \\
&= -\alpha[(B_{(H)} \dots B_{(1)})^\top B_{(H)} \cdots B_{(1)} \otimes I_{m_y}] \operatorname{vec}[\nabla_{(H)} L(W,B)] \\
&\quad - \alpha \sum_{i=1}^{H} [(B_{(i-1)} \dots B_{(1)})^\top B_{(i-1)} \cdots B_{(1)} \otimes W_{(H)} B_{(H)} \cdots B_{(i+1)} (W_{(H)} B_{(H)} \cdots B_{(i+1)})^\top] \operatorname{vec}\left[\nabla_{(H)} L(W,B\right. \\
&\quad + O(\alpha^2)
\end{aligned}$$

Therefore, the induced continuous dynamics of $Z_{(H)} = W_{(H)} B_{(H)} B_{(H-1)} \cdots B_{(1)}$ is

$$\frac{d}{dt} \operatorname{vec}[Z_{(H)}] = -F_{(H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] - \left(\sum_{i=1}^{H} J_{(i,H)}^\top J_{(i,H)} \right) \operatorname{vec}\left[\nabla_{(H)} L(W,B)\right],$$

where

$$F_{(H)} = [(B_{(H)} \dots B_{(1)})^\top B_{(H)} \cdots B_{(1)} \otimes I_{m_y}],$$

and

$$J_{(i,H)} = [B_{(i-1)} \dots B_{(1)} \otimes (W_{(H)} B_{(H)} \cdots B_{(i+1)})^\top].$$

This is because

$$\begin{aligned}
J_{(i,H)}^\top J_{(i,H)} &= [(B_{(i-1)} \dots B_{(1)})^\top \otimes W_{(H)} B_{(H)} \cdots B_{(i+1)}][B_{(i-1)} \dots B_{(1)} \otimes (W_{(H)} B_{(H)} \cdots B_{(i+1)})^\top] \\
&= [(B_{(i-1)} \dots B_{(1)})^\top B_{(i-1)} \dots B_{(1)} \otimes W_{(H)} B_{(H)} \cdots B_{(i+1)} (W_{(H)} B_{(H)} \cdots B_{(i+1)})^\top].
\end{aligned}$$

Dynamics induced int the space of loss value $L(W,B)$

We now analyze the dynamics induced int the space of loss value $L(W,B)$. Using chain rule,

$$\begin{aligned}
\frac{d}{dt} L(W,B) &= \frac{d}{dt} L_0(Z_{(H)}) \\
&= \frac{\partial L_0(Z_{(H)})}{\partial \operatorname{vec}[Z_{(H)}]} \frac{d \operatorname{vec}[Z_{(H)}]}{dt},
\end{aligned}$$

where

$$L_0(Z_{(H)}) = \ell(f_0(X, Z_{(H)})_{*\mathcal{I}}, Y), \;\; f_0(X, Z_{(H)}) = Z_{(H)} X S^H, \;\text{ and } Z_{(H)} = W_{(H)} B_{(H)} B_{(H-1)} \cdots B_{(1)}.$$

Since $f_0(X, Z_{(H)}) = f(X, W, B) = \hat{Y}$ and $L_0(Z_{(H)}) = L(W, B)$, we have that

$$\begin{aligned}
\left(\frac{\partial L_0(Z_{(H)})}{\partial \operatorname{vec}[Z_{(H)}]}\right)^\top &= \left(\frac{\partial L(W,B)}{\partial \operatorname{vec}[\hat{Y}]} \frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[Z_{(H)}]}\right)^\top \\
&= \left(\frac{\partial L(W,B)}{\partial \operatorname{vec}[\hat{Y}]} \left(\frac{\partial}{\partial \operatorname{vec}[Z_{(H)}]} [(X(S^H)_{*\mathcal{I}})^\top \otimes I_{m_y}] \operatorname{vec}[Z_{(H)}]\right)\right)^\top \\
&= [X(S^H)_{*\mathcal{I}} \otimes I_{m_y}] \operatorname{vec}\left[\frac{\partial L(W,B)}{\partial \hat{Y}}\right] \\
&= \operatorname{vec}\left[\frac{\partial L(W,B)}{\partial \hat{Y}} (X(S^H)_{*\mathcal{I}})^\top\right] \\
&= \operatorname{vec}[\nabla_{(H)} L(W,B)]
\end{aligned}$$

Combining these,

$$\begin{aligned}
&\frac{d}{dt} L(W,B) \\
&= \operatorname{vec}[\nabla_{(H)} L(W,B)]^\top \frac{d \operatorname{vec}[Z_{(H)}]}{dt} \\
&= -\operatorname{vec}[\nabla_{(H)} L(W,B)]^\top F_{(H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] - \sum_{i=1}^{H} \operatorname{vec}[\nabla_{(H)} L(W,B)]^\top J_{(i,H)}^\top J_{(i,H)} \operatorname{vec}\left[\nabla_{(H)} L(W,B)\right] \\
&= -\operatorname{vec}[\nabla_{(H)} L(W,B)]^\top F_{(H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] - \sum_{i=1}^{H} \| J_{(i,H)} \operatorname{vec}\left[\nabla_{(H)} L(W,B)\right] \|_2^2
\end{aligned}$$

Therefore,

$$\frac{d}{dt} L(W,B) = -\operatorname{vec}[\nabla_{(H)} L(W,B)]^\top F_{(H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] - \sum_{i=1}^{H} \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] \right\|_2^2 \tag{F.18}$$

Since $F_{(H)}$ is real symmetric and positive semidefinite,

$$\frac{d}{dt} L(W,B) \leq -\lambda_{\min}(F_{(H)}) \| \operatorname{vec}[\nabla_{(H)} L(W,B)] \|_2^2 - \sum_{i=1}^{H} \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] \right\|_2^2.$$

With $\lambda_{W,B} = \lambda_{\min}(F_{(H)})$,

$$\frac{d}{dt}L(W,B) \le -\lambda_{W,B} \| \operatorname{vec}[\nabla_{(H)} L(W,B)]\|_2^2 - \sum_{i=1}^{H} \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] \right\|_2^2 \quad \text{(F.19)}$$

Completing the proof by using the assumption of the square loss

Using the assumption that $L(W,B) = \ell(f(X,W,B)_{*\mathcal{I}}, Y) = \|f(X,W,B)_{*\mathcal{I}} - Y\|_F^2$ with $\hat{Y} = f(X,W,B)_{*\mathcal{I}}$, we have

$$\frac{\partial L(W,B)}{\partial \hat{Y}} = \frac{\partial}{\partial \hat{Y}} \|\hat{Y} - Y\|_F^2 = 2(\hat{Y} - Y) \in \mathbb{R}^{m_y \times n},$$

and

$$\begin{aligned} \operatorname{vec}[\nabla_{(H)} L(W,B)] = \operatorname{vec}\left[\frac{\partial L(W,B)}{\partial \hat{Y}} (X(S^H)_{*\mathcal{I}})^\top \right] &= 2 \operatorname{vec}\left[(\hat{Y} - Y)(X(S^H)_{*\mathcal{I}})^\top \right] \\ &= 2[X(S^H)_{*\mathcal{I}} \otimes I_{m_y}] \operatorname{vec}[\hat{Y} - Y]. \end{aligned}$$

Therefore,

$$\| \operatorname{vec}[\nabla_{(H)} L(W,B)]\|_2^2 = 4 \operatorname{vec}[\hat{Y} - Y]^\top [(X(S^H)_{*\mathcal{I}})^\top X(S^H)_{*\mathcal{I}} \otimes I_{m_y}] \operatorname{vec}[\hat{Y} - Y] \quad \text{(F.20)}$$

Using (F.19) and (F.20),

$$\begin{aligned} \frac{d}{dt}L(W,B) \le &- \lambda_{W,B} \| \operatorname{vec}[\nabla_{(H)} L(W,B)]\|_2^2 - \sum_{i=1}^{H} \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] \right\|_2^2 \\ \le &- 4\lambda_{W,B} \operatorname{vec}[\hat{Y} - Y]^\top [(X(S^H)_{*\mathcal{I}})^\top X(S^H)_{*\mathcal{I}} \otimes I_{m_y}] \operatorname{vec}[\hat{Y} - Y] \\ &- \sum_{i=1}^{H} \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] \right\|_2^2 \\ = &- 4\lambda_{W,B} \operatorname{vec}[\hat{Y} - Y]^\top \left[\tilde{G}_H^\top \tilde{G}_H \otimes I_{m_y} \right] \operatorname{vec}[\hat{Y} - Y] - \sum_{i=1}^{H} \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] \right\|_2^2 \end{aligned}$$

where the last line follows from the following definition:

$$\tilde{G}_H := X(S^H)_{*\mathcal{I}}.$$

Decompose $\operatorname{vec}[\hat{Y}-Y]$ as $\operatorname{vec}[\hat{Y}-Y] = v + v^{\perp}$, where $v = \mathbf{P}_{\tilde{G}_H^\top \otimes I_{m_y}} \operatorname{vec}[\hat{Y}-Y]$, $v^{\perp} = (I_{m_y n} - \mathbf{P}_{\tilde{G}_H^\top \otimes I_{m_y}}) \operatorname{vec}[\hat{Y}-Y]$, and $\mathbf{P}_{\tilde{G}_H^\top \otimes I_{m_y}} \in \mathbb{R}^{m_y n \times m_y n}$ represents the orthogonal projection onto the column space of $\tilde{G}_H^\top \otimes I_{m_y} \in \mathbb{R}^{m_y n \times m_y m_x}$. Then,

$$\begin{aligned}
\operatorname{vec}[\hat{Y}-Y]^\top \left[\tilde{G}_H^\top \tilde{G}_H \otimes I_{m_y}\right] \operatorname{vec}[\hat{Y}-Y] &= (v+v^\perp)^\top \left[\tilde{G}_H^\top \otimes I_{m_y}\right]\left[\tilde{G}_H \otimes I_{m_y}\right](v+v^\perp) \\
&= v^\top \left[\tilde{G}_H^\top \otimes I_{m_y}\right]\left[\tilde{G}_H \otimes I_{m_y}\right] v \\
&\geq \sigma_{\min}^2(\tilde{G}_H) \|\mathbf{P}_{\tilde{G}_H^\top \otimes I_{m_y}} \operatorname{vec}[\hat{Y}-Y]\|_2^2 \\
&= \sigma_{\min}^2(\tilde{G}_H) \|\mathbf{P}_{\tilde{G}_H^\top \otimes I_{m_y}} \operatorname{vec}[\hat{Y}] - \mathbf{P}_{\tilde{G}_H^\top \otimes I_{m_y}} \operatorname{vec}[Y]\|_2^2 \\
&= \sigma_{\min}^2(\tilde{G}_H) \|\operatorname{vec}[\hat{Y}] - \mathbf{P}_{\tilde{G}_H^\top \otimes I_{m_y}} \operatorname{vec}[Y] \pm \operatorname{vec}[Y]\|_2^2 \\
&= \sigma_{\min}^2(\tilde{G}_H) \|\operatorname{vec}[\hat{Y}] - \operatorname{vec}[Y] + (I_{m_y n} - \mathbf{P}_{\tilde{G}_H^\top \otimes I_{m_y}}) \operatorname{vec}[Y]\|_2^2 \\
&\geq \sigma_{\min}^2(\tilde{G}_H) (\|\operatorname{vec}[\hat{Y}-Y]\|_2 - \|(I_{m_y n} - \mathbf{P}_{\tilde{G}_H^\top \otimes I_{m_y}}) \operatorname{vec}[Y]\|_2)^2 \\
&\geq \sigma_{\min}^2(\tilde{G}_H) (\|\operatorname{vec}[\hat{Y}-Y]\|_2^2 - \|(I_{m_y n} - \mathbf{P}_{\tilde{G}_H^\top \otimes I_{m_y}}) \operatorname{vec}[Y]\|_2^2,
\end{aligned}$$

where we used the fact that the singular values of $\left[\tilde{G}_H^\top \otimes I_{m_y}\right]$ are products of singular values of $\tilde{G}_H$ and I_{m_y}.

By noticing that $L(W,B) = \|\operatorname{vec}[\hat{Y}-Y]\|_2^2$ and $L_H^* = \|(I_{m_y n} - \mathbf{P}_{\tilde{G}_H^\top \otimes I_{m_y}}) \operatorname{vec}[Y]\|_2^2$,

$$\operatorname{vec}[\hat{Y}-Y]^\top \left[\tilde{G}_H^\top \tilde{G}_H \otimes I_{m_y}\right] \operatorname{vec}[\hat{Y}-Y] \geq \sigma_{\min}^2(\tilde{G}_H)(L(W,B) - L_H^*).$$

Therefore,

$$\begin{aligned}
\frac{d}{dt} L(W,B) &\leq -4\lambda_{W,B} \operatorname{vec}[\hat{Y}-Y]^\top \left[\tilde{G}_H^\top \tilde{G}_H \otimes I_{m_y}\right] \operatorname{vec}[\hat{Y}-Y] - \sum_{i=1}^{H} \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] \right\|_2^2 \\
&\leq -4\lambda_{W,B} \sigma_{\min}^2(\tilde{G}_H)(L(W,B) - L_H^*) - \sum_{i=1}^{H} \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] \right\|_2^2
\end{aligned}$$

Since $\frac{d}{dt} L_H^* = 0$,

$$\frac{d}{dt}(L(W,B) - L_H^*) \leq -4\lambda_{W,B} \sigma_{\min}^2(\tilde{G}_H)(L(W,B) - L_H^*) - \sum_{i=1}^{H} \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] \right\|_2^2$$

By defining $\mathbf{L} = L(W,B) - L^*_H$,

$$\frac{d\mathbf{L}}{dt} \leq -4\lambda_{W,B}\sigma^2_{\min}(\tilde{G}_H)\mathbf{L} - \sum_{i=1}^{H} \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] \right\|_2^2 \tag{F.21}$$

Since $\frac{d}{dt}\mathbf{L} \leq 0$ and $\mathbf{L} \geq 0$, if $\mathbf{L} = 0$ at some time $\bar{t}$, then $\mathbf{L} = 0$ for any time $t \geq \bar{t}$. Therefore, if $\mathbf{L} = 0$ at some time $\bar{t}$, then we have the desired statement of this theorem for any time $t \geq \bar{t}$. Thus, we can focus on the time interval $[0, \bar{t}]$ such that $\mathbf{L} > 0$ for any time $t \in [0, \bar{t}]$ (here, it is allowed to have $\bar{t} = \infty$). Thus, focusing on the time interval with $\mathbf{L} > 0$, equation (F.21) implies that

$$\frac{1}{\mathbf{L}}\frac{d\mathbf{L}}{dt} \leq -4\lambda_{W,B}\sigma^2_{\min}(\tilde{G}_H) - \frac{1}{\mathbf{L}}\sum_{i=1}^{H} \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] \right\|_2^2$$

By taking integral over time

$$\int_0^T \frac{1}{\mathbf{L}}\frac{d\mathbf{L}}{dt}dt \leq -\int_0^T 4\lambda_{W,B}\sigma^2_{\min}(\tilde{G}_H)dt - \int_0^T \frac{1}{\mathbf{L}}\sum_{i=1}^{H} \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] \right\|_2^2 dt$$

By using the substitution rule for integrals, $\int_0^T \frac{1}{\mathbf{L}}\frac{d\mathbf{L}}{dt}dt = \int_{\mathbf{L}_0}^{\mathbf{L}_T} \frac{1}{\mathbf{L}}d\mathbf{L} = \log(\mathbf{L}_T) - \log(\mathbf{L}_0)$, where $\mathbf{L}_0 = L(W_0, B_0) - L^*$ and $\mathbf{L}_T = L(W_T, B_T) - L^*_H$. Thus,

$$\log(\mathbf{L}_T) - \log(\mathbf{L}_0) \leq -4\sigma^2_{\min}(\tilde{G}_H)\int_0^T \lambda_{W,B}dt - \int_0^T \frac{1}{\mathbf{L}}\sum_{i=1}^{H} \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] \right\|_2^2 dt$$

which implies that

$$\begin{aligned}\mathbf{L}_T &\leq e^{\log(\mathbf{L}_0) - 4\sigma^2_{\min}(\tilde{G}_H)\int_0^T \lambda_{W,B}dt - \int_0^T \frac{1}{\mathbf{L}}\sum_{i=1}^{H} \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] \right\|_2^2 dt} \\ &= \mathbf{L}_0 e^{-4\sigma^2_{\min}(\tilde{G}_H)\int_0^T \lambda_{W,B}dt - \int_0^T \frac{1}{\mathbf{L}}\sum_{i=1}^{H} \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W,B)] \right\|_2^2 dt}\end{aligned}$$

By recalling the definition of $\mathbf{L} = L(W,B) - L^*_H$ and that $\frac{d}{dt}\mathbf{L} \leq 0$, we have that if $L(W_T, B_T) - L^*_H > 0$, then $L(W_t, B_t) - L^*_H > 0$ for all $t \in [0, T]$, and

$$L(W_T, B_T) - L^*_H \leq (L(W_0, B_0) - L^*_H)e^{-4\sigma^2_{\min}(\tilde{G}_H)\int_0^T \lambda_{W_t,B_t}dt - \int_0^T \frac{1}{L(W_t,B_t) - L^*_H}\sum_{i=1}^{H} \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W_t,B_t)] \right\|_2^2 dt}. \tag{F.22}$$

Using the property of Kronecker product,

$$\lambda_{\min}([(B_{(H),t} \dots B_{(1),t})^\top B_{(H),t} \cdots B_{(1),t} \otimes I_{m_y}]) = \lambda_{\min}((B_{(H),t} \dots B_{(1),t})^\top B_{(H),t} \cdots B_{(1),t}),$$

which implies that $\lambda_T^{(H)} = \inf_{t \in [0,T]} \lambda_{W_t,B_t}$. Thus, by noticing that

$$\int_0^T \frac{1}{L(W_t, B_t) - L_H^*} \sum_{i=1}^H \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W_t, B_t)] \right\|_2^2 dt \geq 0$$

Equation (F.22) implies that

$$\begin{aligned} L(W_T, B_T) - L_H^* &\leq (L(W_0, B_0) - L_H^*) e^{-4\lambda_T^{(H)} \sigma_{\min}^2(\tilde{G}_H) T - \int_0^T \frac{1}{L(W_t,B_t) - L_H^*} \sum_{i=1}^H \left\| J_{(i,H)} \operatorname{vec}[\nabla_{(H)} L(W_t,B_t)] \right\|_2^2 dt} \\ &\leq (L(W_0, B_0) - L_H^*) e^{-4\lambda_T^{(H)} \sigma_{\min}^2(\tilde{G}_H) T} \\ &= (L(W_0, B_0) - L_H^*) e^{-4\lambda_T^{(H)} \sigma_{\min}^2(X(S^H)_{*\mathcal{I}}) T} \end{aligned}$$

□

F.1.2 Proof of Proposition 9.2

From Definition 9.4, we have that $\sigma_{\min}(\bar{B}^{(1:H)}) = \sigma_{\min}(B_{(H)} B_{(H-1)} \cdots B_{(1)}) \geq \gamma$ for all (W, B) such that $L(W, B) \leq L(W_0, B_0)$. From equation (F.19) in the proof of Theorem 9.1, it holds that $\frac{d}{dt} L(W_t, B_t) \leq 0$ for all t. Thus, we have that $L(W_t, B_t) \leq L(W_0, B_0)$ and hence $\sigma_{\min}(\bar{B}_t^{(1:H)}) \geq \gamma$ for all t. Under this problem setting ($m_H \geq m_x$), this implies that $\lambda_{\min}((\bar{B}_t^{(1:H)})^\top \bar{B}_t^{(1:H)}) \geq \gamma^2$ for all t and thus $\lambda_T^{(H)} \geq \gamma^2$.

F.1.3 Proof of Proposition 9.3

We first give the complete version of Proposition 9.3. Proposition F.2 is the formal version of Proposition 9.3 and shows that our singular margin generalizes deficiency margin proposed in Arora et al. [2019a]. Using the deficiency margin assumption, Arora et al. [2019a]

analyzed the following optimization problem:

$$\underset{\tilde{W}}{\text{minimize}}\, \tilde{L}(\tilde{W}_{(1)}, \dots, \tilde{W}_{(H+1)}) := \frac{1}{2}\|\tilde{W}_{(H+1)}\tilde{W}_{(H)} \cdots \tilde{W}_{(1)} - \tilde{\Phi}\|_F^2 \tag{F.23}$$

$$= \frac{1}{2}\|\tilde{W}_{(1)}^\top \tilde{W}_{(2)}^\top \cdots \tilde{W}_{(H+1)}^\top - \tilde{\Phi}^\top\|_F^2, \tag{F.24}$$

where $\tilde{\Phi} \in \mathbb{R}^{\tilde{m}_y \times \tilde{m}_x}$ is a target matrix and the last equality follows from $\|M\|_F = \|M^\top\|_F$ for any matrix M by the definition of the Frobenius norm. Therefore, this optimization problem (F.23) from the previous work is equivalent to the following optimization problem in our notation:

$$\underset{W,B}{\text{minimize}}\, L(W, B) := \frac{1}{2}\|WB_{(H)}B_{(H-1)} \cdots B_{(1)} - \Phi\|_F^2, \tag{F.25}$$

where $WB_{(H)}B_{(H-1)} \cdots B_{(1)} = \tilde{W}_{(H+1)}\tilde{W}_{(H)} \cdots \tilde{W}_{(1)}$ (i.e., $W = \tilde{W}_{(H+1)}$ with $B_{(l)} = \tilde{W}_{(l)}$) and $\Phi = \tilde{\Phi}$ if $\tilde{m}_y \geq \tilde{m}_x$, and $WB_{(H)}B_{(H-1)} \cdots B_{(1)} = \tilde{W}_{(1)}^\top \tilde{W}_{(2)}^\top \cdots \tilde{W}_{(H+1)}^\top$ (i.e., $W = \tilde{W}_{(1)}^\top$ with $B_{(l)} = \tilde{W}_{(H+2-l)}^\top$) and $\Phi = \tilde{\Phi}^\top$ if $\tilde{m}_y < \tilde{m}_x$. That is, we have $\Phi \in \mathbb{R}^{m_y \times m_x}$ where $m_y = \tilde{m}_y$ with $m_x = \tilde{m}_x$ if $\tilde{m}_y \geq \tilde{m}_x$, and $m_y = \tilde{m}_x$ with $m_x = \tilde{m}_y$ if $\tilde{m}_y < \tilde{m}_x$. Therefore, our general problem framework with graph structures can be reduced and applicable to the previous optimization problem without graph structures by setting $\frac{1}{n}XX^\top = I$, $S = I$, $\mathcal{I} = [n]$, $f(X, W, B) = WB_{(H)}B_{(H-1)} \cdots B_{(1)}$, and $\ell(q, \Phi) = \frac{1}{2}\|q - \Phi\|_F^2$ where $\Phi \in \mathbb{R}^{m_y \times m_x}$ is a target matrix with $m_y \geq m_x$ without loss of generality. An initialization (W_0, B_0) is said to have deficiency margin $c > 0$ if the end-to-end matrix $W_0\bar{B}_0^{(1:H)}$ of the initialization (W_0, B_0) has deficiency margin $c > 0$ with respect to the target Φ [Arora et al., 2019a, Definition 2]: i.e., Arora et al. [2019a] assumed that the initialization (W_0, B_0) has deficiency margin $c > 0$ (as it is also invariant to the transpose of $\tilde{W}_{(H+1)}\tilde{W}_{(H)} \cdots \tilde{W}_{(1)} - \tilde{\Phi}$).

Proposition F.2. *Consider the optimization problem in [Arora et al., 2019a] by setting* $\frac{1}{n}XX^\top = I$, $S = I$, $\mathcal{I} = [n]$, $f(X, W, B) = WB_{(H)}B_{(H-1)} \cdots B_{(1)}$, *and* $\ell(q, \Phi) = \frac{1}{2}\|q - \Phi\|_F^2$ *where* $\Phi \in \mathbb{R}^{m_y \times m_x}$ *is a target matrix with* $m_y \geq m_x$ *without loss of generality (since the transpose of these two dimensions leads to the equivalent optimization problem under this setting: see above). Then, if an initialization* (W_0, B_0) *has deficiency margin*

$c > 0$*, it has singular margin* $\gamma > 0$.

Proof of Proposition F.2. By the definition of the deficiency margin [Arora et al., 2019a, Definition 2] and its consequence [Arora et al., 2019a, Claim 1], if an initialization (W_0, B_0) has deficiency margin $c > 0$, then any pair (W, B) for which $L(W, B) \leq L(W_0, B_0)$ satisfies $\sigma_{\min}(WB_{(H)}B_{(H-1)} \cdots B_{(1)}) \geq c > 0$. Since the number of nonzero singular values is equal to the matrix rank, this implies that $\operatorname{rank}(WB_{(H)}B_{(H-1)} \cdots B_{(1)}) \geq \min(m_y, m_x)$ for any pair (W, B) for which $L(W, B) \leq L(W_0, B_0)$. Since $\operatorname{rank}(MM') \leq \min(\operatorname{rank}(M), \operatorname{rank}(M'))$, this implies that

$$m_H \geq \min(m_y, m_x) = m_x, \tag{F.26}$$

(as well as $m_l \geq \min(m_y, m_x)$ for all l), and that for any pair (W, B) for which $L(W, B) \leq L(W_0, B_0)$,

$$\begin{aligned} m_x = \min(m_y, m_x) &\leq \operatorname{rank}(WB_{(H)}B_{(H-1)} \cdots B_{(1)}) && \text{(F.27)} \\ &\leq \min(\operatorname{rank}(W), \operatorname{rank}(B_{(H)}B_{(H-1)} \cdots B_{(1)})) && \text{(F.28)} \\ &\leq \operatorname{rank}(B_{(H)}B_{(H-1)} \cdots B_{(1)}) \leq m_x. && \text{(F.29)} \end{aligned}$$

This shows that $\operatorname{rank}(B_{(H)}B_{(H-1)} \cdots B_{(1)}) = m_x$ for any pair (W, B) for which $L(W, B) \leq L(W_0, B_0)$. Since $m_H \geq m_x$ from (F.26) and the number of nonzero singular values is equal to the matrix rank, this implies that $\sigma_{\min}(B_{(H)}B_{(H-1)} \cdots B_{(1)}) \geq \gamma$ for some $\gamma > 0$ for any pair (W, B) for which $L(W, B) \leq L(W_0, B_0)$. Thus, if an initialization (W_0, B_0) has deficiency margin $c > 0$, then it has singular margin $\gamma > 0$.

$\square$

F.1.4 Proof of Theorem 9.4

This section completes the proof of Theorem 9.4. We compute the derivatives of the output of multiscale linear GNN with respect to the parameters $W_{(l)}$ and $B_{(l)}$ in Appendix F.1.4. Then using these derivatives, we compute the gradient of the loss with respect to $W_{(l)}$ in Appendix F.1.4 and $B_{(l)}$ in Appendix F.1.4. We then rearrange the formula of the gradients such that they are related to the formula of $\nabla_{(l)} L(W, B)$ in Appendices F.1.4. Using the proven relation, we first analyze the dynamics induced in the space of $W_{(l)} B_{(l)} B_{(l-1)} \cdots B_{(1)}$ in Appendix F.1.4, and then the dynamics induced int the space of loss value $L(W, B)$ in Appendix F.1.4. Finally, we complete the proof by using the assumption of using the square loss in Appendices F.1.4–F.1.4. In the following, we first prove the statement for the case of $\mathcal{I} = [n]$ for the simplicity of notation and then prove the statement for the general case afterwards.

Derivation of formula for $\frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[W_{(l)}]} \in \mathbb{R}^{m_y n \times m_y m_l}$ and $\frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[B_{(l)}]} \in \mathbb{R}^{m_y n \times m_l m_{l-1}}$

We can easily compute $\frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[W_{(l)}]}$ by using the property of the Kronecker product as follows:

$$\begin{aligned} \frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[W_{(l)}]} = \frac{\partial}{\partial \operatorname{vec}[W_{(l)}]} \sum_{k=0}^{H} \operatorname{vec}[W_{(k)} X_{(k)}] &= \frac{\partial}{\partial \operatorname{vec}[W_{(l)}]} \sum_{k=0}^{H} [X_{(k)}^\top \otimes I_{m_y}] \operatorname{vec}[W_{(k)}] \\ &= [X_{(l)}^\top \otimes I_{m_y}] \in \mathbb{R}^{m_y n \times m_y m_l} \end{aligned} \tag{F.30}$$

We now compute $\frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[B_{(l)}]}$ by using the chain rule and the property of the Kronecker product as follows:

$$\begin{aligned} \frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[B_{(l)}]} &= \frac{\partial}{\partial \operatorname{vec}[B_{(l)}]} \sum_{k=0}^{H} \operatorname{vec}[W_{(k)} X_{(k)}] \\ &= \frac{\partial}{\partial \operatorname{vec}[B_{(l)}]} \sum_{k=0}^{H} [I_n \otimes W_{(k)}] \operatorname{vec}[X_{(k)}] \\ &= \sum_{k=0}^{H} [I_n \otimes W_{(k)}] \frac{\partial \operatorname{vec}[X_{(k)}]}{\partial \operatorname{vec}[B_{(l)}]} \\ &= \sum_{k=l}^{H} [I_n \otimes W_{(k)}] \frac{\partial \operatorname{vec}[X_{(k)}]}{\partial \operatorname{vec}[X_{(l)}]} \frac{\partial \operatorname{vec}[X_{(l)}]}{\partial \operatorname{vec}[B_{(l)}]} \\ &= \sum_{k=l}^{H} [I_n \otimes W_{(k)}] \frac{\partial \operatorname{vec}[X_{(k)}]}{\partial \operatorname{vec}[X_{(l)}]} \frac{\partial \operatorname{vec}[B_{(l)} X_{(l-1)} S]}{\partial \operatorname{vec}[B_{(l)}]} \\ &= \sum_{k=l}^{H} [I_n \otimes W_{(k)}] \frac{\partial \operatorname{vec}[X_{(k)}]}{\partial \operatorname{vec}[X_{(l)}]} \frac{\partial [(X_{(l-1)} S)^\top \otimes I_{m_l}] \operatorname{vec}[B_{(l)}]}{\partial \operatorname{vec}[B_{(l)}]} \\ &= \sum_{k=l}^{H} [I_n \otimes W_{(k)}] \frac{\partial \operatorname{vec}[X_{(k)}]}{\partial \operatorname{vec}[X_{(l)}]} [(X_{(l-1)} S)^\top \otimes I_{m_l}] \end{aligned}$$

Here, for any $k \geq 1$,

$$\operatorname{vec}[X_{(k)}] = \operatorname{vec}[B_{(k)}X_{(k-1)}S] = \operatorname{vec}[S^\top \otimes B_{(k)}] \operatorname{vec}[X_{(k-1)}].$$

By recursively applying this, we have that for any $k \geq l$,

$$\begin{aligned}\operatorname{vec}[X_{(k)}] &= \operatorname{vec}[S^\top \otimes B_{(k)}] \operatorname{vec}[S^\top \otimes B_{(k-1)}] \cdots \operatorname{vec}[S^\top \otimes B_{(l+1)}] \operatorname{vec}[X_{(l)}] \\ &= \operatorname{vec}[(S^{k-l})^\top \otimes B_{(k)}B_{(k-1)} \cdots B_{(l+1)}] \operatorname{vec}[X_{(l)}],\end{aligned}$$

where $S^0 := I_n$ and

$$B_{(k)}B_{(k-1)} \cdots B_{(l+1)} := I_{m_l} \quad \text{if } k = l.$$

Therefore,

$$\frac{\partial \operatorname{vec}[X_{(k)}]}{\partial \operatorname{vec}[X_{(l)}]} = \operatorname{vec}[(S^{k-l})^\top \otimes B_{(k)}B_{(k-1)} \cdots B_{(l+1)}].$$

Combining the above equations yields

$$\begin{aligned}\frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[B_{(l)}]} &= \sum_{k=l}^{H} [I_n \otimes W_{(k)}] \frac{\partial \operatorname{vec}[X_{(k)}]}{\partial \operatorname{vec}[X_{(l)}]} [(X_{(l-1)}S)^\top \otimes I_{m_l}] \\ &= \sum_{k=l}^{H} [I_n \otimes W_{(k)}] \operatorname{vec}[(S^{k-l})^\top \otimes B_{(k)}B_{(k-1)} \cdots B_{(l+1)}][(X_{(l-1)}S)^\top \otimes I_{m_l}] \\ &= \sum_{k=l}^{H} [(X_{(l-1)}S^{k-l+1})^\top \otimes W_{(k)}B_{(k)}B_{(k-1)} \cdots B_{(l+1)}] \in \mathbb{R}^{m_y n \times m_l m_{l-1}}. \quad \text{(F.31)}\end{aligned}$$

Derivation of a formula of $\nabla_{W_{(l)}} L(W, B) \in \mathbb{R}^{m_y \times m_l}$

Using the chain rule and (F.30), we have that

$$\frac{\partial L(W,B)}{\partial \operatorname{vec}[W_{(l)}]} = \frac{\partial L(W,B)}{\partial \operatorname{vec}[\hat{Y}]} \frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[W_{(l)}]} = \frac{\partial L(W,B)}{\partial \operatorname{vec}[\hat{Y}]} [X_{(l)}^\top \otimes I_{m_y}].$$

Thus, with $\frac{\partial L(W,B)}{\partial \hat{Y}} \in \mathbb{R}^{m_y \times n}$, by using

$$\begin{aligned}
\nabla_{\mathrm{vec}[W_{(l)}]} L(W,B) &= \left(\frac{\partial L(W,B)}{\partial \,\mathrm{vec}[W_{(l)}]}\right)^\top \\
&= [X_{(l)} \otimes I_{m_y}] \left(\frac{\partial L(W,B)}{\partial \,\mathrm{vec}[\hat{Y}]}\right)^\top \\
&= [X_{(l)} \otimes I_{m_y}] \,\mathrm{vec}\left[\frac{\partial L(W,B)}{\partial \hat{Y}}\right] \\
&= \mathrm{vec}\left[\frac{\partial L(W,B)}{\partial \hat{Y}} X_{(l)}^\top\right] \in \mathbb{R}^{m_y m_l}.
\end{aligned}$$

Therefore,

$$\nabla_{W_{(l)}} L(W,B) = \frac{\partial L(W,B)}{\partial \hat{Y}} X_{(l)}^\top \in \mathbb{R}^{m_y \times m_l}. \tag{F.32}$$

Derivation of a formula of $\nabla_{B_{(l)}} L(W,B) \in \mathbb{R}^{m_l \times m_{l-1}}$

Using the chain rule and (F.31), we have that

$$\frac{\partial L(W,B)}{\partial \,\mathrm{vec}[B_{(l)}]} = \frac{\partial L(W,B)}{\partial \,\mathrm{vec}[\hat{Y}]} \frac{\partial \,\mathrm{vec}[\hat{Y}]}{\partial \,\mathrm{vec}[B_{(l)}]} = \frac{\partial L(W,B)}{\partial \,\mathrm{vec}[\hat{Y}]} \sum_{k=l}^{H} [(X_{(l-1)} S^{k-l+1})^\top \otimes W_{(k)} B_{(k)} B_{(k-1)} \cdots B_{(l+1)}].$$

Thus, with $\frac{\partial L(W,B)}{\partial \hat{Y}} \in \mathbb{R}^{m_y \times n}$,

$$\begin{aligned}
\nabla_{\mathrm{vec}[B_{(l)}]} L(W,B) &= \left(\frac{\partial L(W,B)}{\partial \,\mathrm{vec}[B_{(l)}]}\right)^\top \\
&= \sum_{k=l}^{H} [X_{(l-1)} S^{k-l+1} \otimes (W_{(k)} B_{(k)} B_{(k-1)} \cdots B_{(l+1)})^\top] \left(\frac{\partial L(W,B)}{\partial \,\mathrm{vec}[\hat{Y}]}\right)^\top \\
&= \sum_{k=l}^{H} [X_{(l-1)} S^{k-l+1} \otimes (W_{(k)} B_{(k)} B_{(k-1)} \cdots B_{(l+1)})^\top] \,\mathrm{vec}\left[\frac{\partial L(W,B)}{\partial \hat{Y}}\right] \\
&= \sum_{k=l}^{H} \mathrm{vec}\left[(W_{(k)} B_{(k)} B_{(k-1)} \cdots B_{(l+1)})^\top \frac{\partial L(W,B)}{\partial \hat{Y}} (X_{(l-1)} S^{k-l+1})^\top\right] \in \mathbb{R}^{m_l m_{l-1}}.
\end{aligned}$$

Therefore,

$$\nabla_{B_{(l)}} L(W,B) = \sum_{k=l}^{H} (W_{(k)} B_{(k)} B_{(k-1)} \cdots B_{(l+1)})^\top \frac{\partial L(W,B)}{\partial \hat{Y}} (X_{(l-1)} S^{k-l+1})^\top \in \mathbb{R}^{m_l \times m_{l-1}}. \tag{F.33}$$

Relating gradients to $\nabla_{(l)} L$

We now relate the gradients of the loss to $\nabla_{(l)} L$, which is defined by

$$\nabla_{(l)} L(W,B) := \frac{\partial L(W,B)}{\partial \hat{Y}} (X S^l)^\top \in \mathbb{R}^{m_y \times m_x}.$$

By using this definition and (F.32), we have that

$$\begin{aligned}
\nabla_{W_{(l)}} L(W,B) &= \frac{\partial L(W,B)}{\partial \hat{Y}} X_{(l)}^\top \\
&= \frac{\partial L(W,B)}{\partial \hat{Y}} (B_{(l)} X_{(l-1)} S)^\top \\
&= \frac{\partial L(W,B)}{\partial \hat{Y}} (B_{(l)} B_{(l-1)} \dots B_{(1)} X S^l)^\top \\
&= \nabla_{(l)} L(W,B) (B_{(l)} B_{(l-1)} \dots B_{(1)})^\top,
\end{aligned}$$

where $B_{(l)} B_{(l-1)} \dots B_{(1)} := I_{m_x}$ if $l = 0$. Similarly, by using the definition and (F.33),

$$\begin{aligned}
\nabla_{B_{(l)}} L(W,B) &= \sum_{k=l}^{H} (W_{(k)} B_{(k)} B_{(k-1)} \cdots B_{(l+1)})^\top \frac{\partial L(W,B)}{\partial \hat{Y}} (X_{(l-1)} S^{k-l+1})^\top \\
&= \sum_{k=l}^{H} (W_{(k)} B_{(k)} B_{(k-1)} \cdots B_{(l+1)})^\top \frac{\partial L(W,B)}{\partial \hat{Y}} (B_{(l-1)} B_{(l-2)} \dots B_{(1)} X S^{l-1} S^{k-l+1})^\top \\
&= \sum_{k=l}^{H} (W_{(k)} B_{(k)} B_{(k-1)} \cdots B_{(l+1)})^\top \frac{\partial L(W,B)}{\partial \hat{Y}} (B_{(l-1)} B_{(l-2)} \dots B_{(1)} X S^{k})^\top \\
&= \sum_{k=l}^{H} (W_{(k)} B_{(k)} B_{(k-1)} \cdots B_{(l+1)})^\top \nabla_{(k)} L(W,B) (B_{(l-1)} B_{(l-2)} \dots B_{(1)})^\top
\end{aligned}$$

where $B_{(l-1)} B_{(l-2)} \dots B_{(1)} := I_{m_x}$ if $l = 1$. In summary thus far, we have that

$$\nabla_{W_{(l)}} L(W,B) = \nabla_{(l)} L(W,B) (B_{(l)} B_{(l-1)} \dots B_{(1)})^\top \in \mathbb{R}^{m_y \times m_l}, \tag{F.34}$$

and

$$\nabla_{B_{(l)}} L(W,B) = \sum_{k=l}^{H} (W_{(k)} B_{(k)} B_{(k-1)} \cdots B_{(l+1)})^\top \nabla_{(k)} L(W,B) (B_{(l-1)} B_{(l-2)} \dots B_{(1)})^\top \in \mathbb{R}^{m_l \times m_{l-1}}, \tag{F.35}$$

where $\nabla_{(l)} L(W,B) := \frac{\partial L(W,B)}{\partial \hat{Y}} (XS^l)^\top \in \mathbb{R}^{m_y \times m_x}$, $B_{(k)} B_{(k-1)} \cdots B_{(l+1)} := I_{m_l}$ if $k = l$, $B_{(l)} B_{(l-1)} \dots B_{(1)} := I_{m_x}$ if $l = 0$, and $B_{(l-1)} B_{(l-2)} \dots B_{(1)} := I_{m_x}$ if $l = 1$.

Dynamics induced in the space of $W_{(l)} B_{(l)} B_{(l-1)} \cdots B_{(1)}$

We now consider the Dynamics induced in the space of $W_{(l)} B_{(l)} B_{(l-1)} \cdots B_{(1)}$. We first consider the following discrete version of the dynamics:

$$W'_{(l)} = W_{(l)} - \alpha \nabla_{W_{(l)}} L(W,B)$$

$$B'_{(l)} = B_{(l)} - \alpha \nabla_{B_{(l)}} L(W,B).$$

This dynamics induces the following dynamics:

$$W'_{(l)} B'_{(l)} B'_{(l-1)} \cdots B'_{(1)} = (W_{(l)} - \alpha \nabla_{W_{(l)}} L(W,B))(B_{(l)} - \alpha \nabla_{B_{(l)}} L(W,B)) \cdots (B_{(1)} - \alpha \nabla_{B_{(1)}} L(W,B)).$$

Define

$$Z_{(l)} := W_{(l)} B_{(l)} B_{(l-1)} \cdots B_{(1)}$$

and

$$Z'_{(l)} := W'_{(l)} B'_{(l)} B'_{(l-1)} \cdots B'_{(1)}.$$

Then, we can rewrite

$$Z'_{(l)} = (W_{(l)} - \alpha \nabla_{W_{(l)}} L(W,B))(B_{(l)} - \alpha \nabla_{B_{(l)}} L(W,B)) \cdots (B_{(1)} - \alpha \nabla_{B_{(1)}} L(W,B)).$$

By expanding the multiplications, this can be written as:

$$\begin{aligned}Z'_{(l)} =& Z_{(l)} - \alpha \nabla_{W_{(l)}} L(W,B) B_{(l)} \cdots B_{(1)} \\ & - \alpha \sum_{i=1}^{l} W_{(l)} B_{(l)} \cdots B_{(i+1)} \nabla_{B_{(i)}} L(W,B) B_{(i-1)} \cdots B_{(1)} + O(\alpha^2)\end{aligned}$$

By vectorizing both sides,

$$\begin{aligned}&\mathrm{vec}[Z'_{(l)}] - \mathrm{vec}[Z_{(l)}] \\ &= -\alpha\, \mathrm{vec}[\nabla_{W_{(l)}} L(W,B) B_{(l)} \cdots B_{(1)}] - \alpha \sum_{i=1}^{l} \mathrm{vec}[W_{(l)} B_{(l)} \cdots B_{(i+1)} \nabla_{B_{(i)}} L(W,B) B_{(i-1)} \cdots B_{(1)}] + O(\alpha^2)\end{aligned}$$

Here, using the formula of $\nabla_{W_{(l)}} L(W,B)$ and $\nabla_{B_{(l)}} L(W,B)$, we have that

$$\begin{aligned}\mathrm{vec}[\nabla_{W_{(l)}} L(W,B) B_{(l)} \cdots B_{(1)}] &= \mathrm{vec}[\nabla_{(l)} L(W,B) (B_{(l)} \ldots B_{(1)})^\top B_{(l)} \cdots B_{(1)}] \\ &= [(B_{(l)} \ldots B_{(1)})^\top B_{(l)} \cdots B_{(1)} \otimes I_{m_y}]\, \mathrm{vec}[\nabla_{(l)} L(W,B)],\end{aligned}$$

and

$$\begin{aligned}&\sum_{i=1}^{l} \mathrm{vec}[W_{(l)} B_{(l)} \cdots B_{(i+1)} \nabla_{B_{(i)}} L(W,B) B_{(i-1)} \cdots B_{(1)}] \\ &= \sum_{i=1}^{l} \mathrm{vec}\left[W_{(l)} B_{(l)} \cdots B_{(i+1)} \sum_{k=i}^{H} (W_{(k)} B_{(k)} \cdots B_{(i+1)})^\top \nabla_{(k)} L(W,B) (B_{(i-1)} \ldots B_{(1)})^\top B_{(i-1)} \cdots B_{(1)}\right] \\ &= \sum_{i=1}^{l} \sum_{k=i}^{H} \mathrm{vec}\left[W_{(l)} B_{(l)} \cdots B_{(i+1)} (W_{(k)} B_{(k)} \cdots B_{(i+1)})^\top \nabla_{(k)} L(W,B) (B_{(i-1)} \ldots B_{(1)})^\top B_{(i-1)} \cdots B_{(1)}\right] \\ &= \sum_{i=1}^{l} \sum_{k=i}^{H} [(B_{(i-1)} \ldots B_{(1)})^\top B_{(i-1)} \cdots B_{(1)} \otimes W_{(l)} B_{(l)} \cdots B_{(i+1)} (W_{(k)} B_{(k)} \cdots B_{(i+1)})^\top]\, \mathrm{vec}\left[\nabla_{(k)} L(W,B)\right].\end{aligned}$$

Summarizing above,

$$
\begin{aligned}
&\operatorname{vec}[Z'_{(l)}] - \operatorname{vec}[Z_{(l)}] \\
&= -\alpha[(B_{(l)} \dots B_{(1)})^\top B_{(l)} \cdots B_{(1)} \otimes I_{m_y}] \operatorname{vec}[\nabla_{(l)} L(W, B)] \\
&\quad - \alpha \sum_{i=1}^{l} \sum_{k=i}^{H} [(B_{(i-1)} \dots B_{(1)})^\top B_{(i-1)} \cdots B_{(1)} \otimes W_{(l)} B_{(l)} \cdots B_{(i+1)} (W_{(k)} B_{(k)} \cdots B_{(i+1)})^\top] \operatorname{vec} \Big[\nabla_{(k)} L(W, B) \\
&\quad + O(\alpha^2)
\end{aligned}
$$

Therefore, the induced continuous dynamics of $Z_{(l)} = W_{(l)} B_{(l)} B_{(l-1)} \cdots B_{(1)}$ is

$$
\frac{d}{dt} \operatorname{vec}[Z_{(l)}] = -F_{(l)} \operatorname{vec}[\nabla_{(l)} L(W, B)] - \sum_{i=1}^{l} \sum_{k=i}^{H} J_{(i,l)}^\top J_{(i,k)} \operatorname{vec} \Big[\nabla_{(k)} L(W, B)\Big]
$$

where

$$
F_{(l)} = [(B_{(l)} \dots B_{(1)})^\top B_{(l)} \cdots B_{(1)} \otimes I_{m_y}],
$$

and

$$
J_{(i,l)} = [B_{(i-1)} \dots B_{(1)} \otimes (W_{(l)} B_{(l)} \cdots B_{(i+1)})^\top].
$$

This is because

$$
\begin{aligned}
J_{(i,k)}^\top J_{(i,k)} &= [(B_{(i-1)} \dots B_{(1)})^\top \otimes W_{(l)} B_{(l)} \cdots B_{(i+1)}][B_{(i-1)} \dots B_{(1)} \otimes (W_{(k)} B_{(k)} \cdots B_{(i+1)})^\top] \\
&= [(B_{(i-1)} \dots B_{(1)})^\top B_{(i-1)} \dots B_{(1)} \otimes W_{(l)} B_{(l)} \cdots B_{(i+1)} (W_{(k)} B_{(k)} \cdots B_{(i+1)})^\top].
\end{aligned}
$$

Dynamics induced int the space of loss value $L(W, B)$

We now analyze the dynamics induced int the space of loss value $L(W, B)$. Define

$$
L(W, B) := \ell(f(X, W, B), Y),
$$

where ℓ is chosen later. Using chain rule,

$$\begin{aligned}\frac{d}{dt}L(W,B) &= \frac{d}{dt}L_0(Z_{(H)},\dots,Z_{(0)}) \\ &= \sum_{l=0}^{H} \frac{\partial L_0(Z_{(l)},\dots,Z_{(0)})}{\partial \operatorname{vec}[Z_{(l)}]} \frac{d \operatorname{vec}[Z_{(l)}]}{dt},\end{aligned}$$

where

$$L_0(Z_{(H)},\dots,Z_{(0)}) = \ell(f_0(X,Z),Y),\ \ f_0(X,Z) = \sum_{l=0}^{H} Z_{(l)}XS^l,\ \text{ and } Z_{(l)} = W_{(l)}B_{(l)}B_{(l-1)}\cdots B_{(1)}.$$

Since $f_0(X,Z) = f(X,W,B) = \hat{Y}$ and $L_0(Z_{(H)},\dots,Z_{(0)}) = L(W,B)$,

$$\begin{aligned}\left(\frac{\partial L_0(Z_{(l)},\dots,Z_{(0)})}{\partial \operatorname{vec}[Z_{(l)}]}\right)^{\top} &= \left(\frac{\partial L(W,B)}{\partial \operatorname{vec}[\hat{Y}]}\frac{\partial \operatorname{vec}[\hat{Y}]}{\partial \operatorname{vec}[Z_{(l)}]}\right)^{\top} \\ &= \left(\frac{\partial L(W,B)}{\partial \operatorname{vec}[\hat{Y}]}\left(\frac{\partial}{\partial \operatorname{vec}[Z_{(l)}]}\sum_{k=0}^{H}[(XS^k)^{\top}\otimes I_{m_y}]\operatorname{vec}[Z_{(k)}]\right)\right)^{\top} \\ &= [XS^l \otimes I_{m_y}]\operatorname{vec}\left[\frac{\partial L(W,B)}{\partial \hat{Y}}\right] \\ &= \operatorname{vec}\left[\frac{\partial L(W,B)}{\partial \hat{Y}}(XS^l)^{\top}\right] \\ &= \operatorname{vec}[\nabla_{(l)}L(W,B)]\end{aligned}$$

Therefore,

$$\begin{aligned}&\frac{d}{dt}L(W,B) \\ &= \sum_{l=0}^{H}\operatorname{vec}[\nabla_{(l)}L(W,B)]^{\top}\frac{d \operatorname{vec}[Z_{(l)}]}{dt} \\ &= -\sum_{l=0}^{H}\operatorname{vec}[\nabla_{(l)}L(W,B)]^{\top}F_{(l)}\operatorname{vec}[\nabla_{(l)}L(W,B)] \\ &\quad -\sum_{l=1}^{H}\sum_{i=1}^{l}\sum_{k=i}^{H}\operatorname{vec}[\nabla_{(l)}L(W,B)]^{\top}J_{(i,l)}^{\top}J_{(i,k)}\operatorname{vec}\left[\nabla_{(k)}L(W,B)\right]\end{aligned}$$

To simplify the second term, define $M_{(l,i)} = \sum_{k=i}^{H} \operatorname{vec}[\nabla_{(l)} L(W,B)]^\top J_{(i,l)}^\top J_{(i,k)} \operatorname{vec}\left[\nabla_{(k)} L(W,B)\right]$ and note that we can expand the double sums and regroup terms as follows:

$$\sum_{l=1}^{H}\sum_{i=1}^{l} M_{(l,i)} = \sum_{l=1}^{H} M_{(l,1)} + \sum_{l=2}^{H} M_{(l,2)} + \cdots + \sum_{l=H}^{H} M_{(l,H)} = \sum_{i=1}^{H}\sum_{l=i}^{H} M_{(l,i)}.$$

Moreover, for each $i \in \{1,\ldots,H\}$,

$$\begin{aligned}
\sum_{l=i}^{H} M_{(l,i)} &= \sum_{l=i}^{H}\sum_{k=i}^{H} \operatorname{vec}[\nabla_{(l)} L(W,B)]^\top J_{(i,l)}^\top J_{(i,k)} \operatorname{vec}\left[\nabla_{(k)} L(W,B)\right] \\
&= \left(\sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W,B)]\right)^\top \left(\sum_{k=i}^{H} J_{(i,k)} \operatorname{vec}\left[\nabla_{(k)} L(W,B)\right]\right) \\
&= \left\|\sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W,B)]\right\|_2^2
\end{aligned}$$

Using these facts, the second term can be simplified as

$$\begin{aligned}
&\sum_{l=1}^{H}\sum_{i=1}^{l}\sum_{k=i}^{H} \operatorname{vec}[\nabla_{(l)} L(W,B)]^\top J_{(i,l)}^\top J_{(i,k)} \operatorname{vec}\left[\nabla_{(k)} L(W,B)\right] \\
&= \sum_{l=1}^{H}\sum_{i=1}^{l} M_{(l,i)} \\
&= \sum_{i=1}^{H}\sum_{l=i}^{H} M_{(l,i)} \\
&= \sum_{i=1}^{H} \left\|\sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W,B)]\right\|_2^2
\end{aligned}$$

Combining these,

$$\frac{d}{dt} L(W,B) = -\sum_{l=0}^{H} \operatorname{vec}[\nabla_{(l)} L(W,B)]^\top F_{(l)} \operatorname{vec}[\nabla_{(l)} L(W,B)] - \sum_{i=1}^{H} \left\|\sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W,B)]\right\|_2^2 \tag{F.36}$$

Since $F_{(l)}$ is real symmetric and positive semidefinite,

$$\frac{d}{dt}L(W,B) \le -\sum_{l=0}^{H} \lambda_{\min}(F_{(l)}) \| \operatorname{vec}[\nabla_{(l)} L(W,B)]\|_2^2 - \sum_{i=1}^{H} \left\| \sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W,B)] \right\|_2^2 \tag{F.37}$$

Completing the proof by using the assumption of the square loss

Using the assumption that $L(W,B) = \ell(f(X,W,B),Y) = \|f(X,W,B) - Y\|_F^2$ with $\hat{Y} = f(X,W,B)$, we have

$$\frac{\partial L(W,B)}{\partial \hat{Y}} = \frac{\partial}{\partial \hat{Y}} \|\hat{Y} - Y\|_F^2 = 2(\hat{Y} - Y) \in \mathbb{R}^{m_y \times n},$$

and hence

$$\operatorname{vec}[\nabla_{(l)} L(W,B)] = \operatorname{vec}\left[\frac{\partial L(W,B)}{\partial \hat{Y}} (XS^l)^\top\right] = 2 \operatorname{vec}\left[(\hat{Y} - Y)(XS^l)^\top\right] = 2[XS^l \otimes I_{m_y}] \operatorname{vec}[\hat{Y} - Y].$$

Therefore,

$$\| \operatorname{vec}[\nabla_{(l)} L(W,B)]\|_2^2 = 4 \operatorname{vec}[\hat{Y} - Y]^\top [(XS^l)^\top XS^l \otimes I_{m_y}] \operatorname{vec}[\hat{Y} - Y]. \tag{F.38}$$

We are now ready to complete the proof of Theorem 9.4 for each cases (i), (ii) and (iii).

Case (I): Completing The Proof of Theorem 9.4 (i)

Using equation (F.37) and (F.38) with $\lambda_{W,B} = \min_{0 \le l \le H} \lambda_{\min}(F_{(l)})$, we have that

$$\begin{aligned}
\frac{d}{dt}L(W,B) &\le -\lambda_{W,B} \sum_{l=0}^{H} \| \operatorname{vec}[\nabla_{(l)} L(W,B)] \|_2^2 - \sum_{i=1}^{H} \left\| \sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W,B)] \right\|_2^2 \\
&\le -4\lambda_{W,B} \sum_{l=0}^{H} \operatorname{vec}[\hat{Y} - Y]^\top [(XS^l)^\top XS^l \otimes I_{m_y}] \operatorname{vec}[\hat{Y} - Y] \\
&\quad - \sum_{i=1}^{H} \left\| \sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W,B)] \right\|_2^2 \\
&\le -4\lambda_{W,B} \operatorname{vec}[\hat{Y} - Y]^\top \left[\left(\sum_{l=0}^{H} (XS^l)^\top XS^l \right) \otimes I_{m_y} \right] \operatorname{vec}[\hat{Y} - Y] \\
&\quad - \sum_{i=1}^{H} \left\| \sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W,B)] \right\|_2^2 \\
&= -4\lambda_{W,B} \operatorname{vec}[\hat{Y} - Y]^\top \left[G_H^\top G_H \otimes I_{m_y} \right] \operatorname{vec}[\hat{Y} - Y] \\
&\quad - \sum_{i=1}^{H} \left\| \sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W,B)] \right\|_2^2
\end{aligned}$$

where the last line follows from the following fact:

$$G_H^\top G_H = \begin{bmatrix} X \\ XS \\ \vdots \\ XS^H \end{bmatrix}^\top \begin{bmatrix} X \\ XS \\ \vdots \\ XS^H \end{bmatrix} = \sum_{l=0}^{H} (XS^l)^\top XS^l.$$

Decompose $\operatorname{vec}[\hat{Y} - Y]$ as $\operatorname{vec}[\hat{Y} - Y] = v + v^\perp$, where $v = \mathbf{P}_{G_H^\top \otimes I_{m_y}} \operatorname{vec}[\hat{Y} - Y]$, $v^\perp = (I_{m_y n} - \mathbf{P}_{G_H^\top \otimes I_{m_y}}) \operatorname{vec}[\hat{Y} - Y]$, and $\mathbf{P}_{G_H^\top \otimes I_{m_y}} \in \mathbb{R}^{m_y n \times m_y n}$ represents the orthogonal projection onto the column space of $G_H^\top \otimes I_{m_y} \in \mathbb{R}^{m_y n \times (H+1) m_y m_x}$. Then,

$$\begin{aligned}
\mathrm{vec}[\hat{Y} - Y]^\top \left[G_H^\top G_H \otimes I_{m_y}\right] \mathrm{vec}[\hat{Y} - Y] &= (v + v^\perp)^\top \left[G_H^\top \otimes I_{m_y}\right] \left[G_H \otimes I_{m_y}\right] (v + v^\perp) \\
&= v^\top \left[G_H^\top \otimes I_{m_y}\right] \left[G_H \otimes I_{m_y}\right] v \\
&\geq \sigma^2_{\min}(G_H) \|\mathbf{P}_{G_H^\top \otimes I_{m_y}} \mathrm{vec}[\hat{Y} - Y]\|_2^2 \\
&= \sigma^2_{\min}(G_H) \|\mathbf{P}_{G_H^\top \otimes I_{m_y}} \mathrm{vec}[\hat{Y}] - \mathbf{P}_{G_H^\top \otimes I_{m_y}} \mathrm{vec}[Y]\|_2^2 \\
&= \sigma^2_{\min}(G_H) \| \mathrm{vec}[\hat{Y}] - \mathbf{P}_{G_H^\top \otimes I_{m_y}} \mathrm{vec}[Y] \pm \mathrm{vec}[Y]\|_2^2 \\
&= \sigma^2_{\min}(G_H) \| \mathrm{vec}[\hat{Y}] - \mathrm{vec}[Y] + (I_{m_y n} - \mathbf{P}_{G_H^\top \otimes I_{m_y}}) \mathrm{vec}[Y]\|_2^2 \\
&\geq \sigma^2_{\min}(G_H) (\| \mathrm{vec}[\hat{Y} - Y]\|_2 - \|(I_{m_y n} - \mathbf{P}_{G_H^\top \otimes I_{m_y}}) \mathrm{vec}[Y]\|_2)^2 \\
&\geq \sigma^2_{\min}(G_H) (\| \mathrm{vec}[\hat{Y} - Y]\|_2^2 - \|(I_{m_y n} - \mathbf{P}_{G_H^\top \otimes I_{m_y}}) \mathrm{vec}[Y]\|_2^2,
\end{aligned}$$

where we used the fact that the singular values of $\left[G_H^\top \otimes I_{m_y}\right]$ are products of singular values of G_H and I_{m_y}.

By noticing that $L(W, B) = \| \mathrm{vec}[\hat{Y} - Y]\|_2^2$ and $L^*_{1:H} = \|(I_{m_y n} - \mathbf{P}_{G_H^\top \otimes I_{m_y}}) \mathrm{vec}[Y]\|_2^2$

,

$$\mathrm{vec}[\hat{Y} - Y]^\top \left[G_H^\top G_H \otimes I_{m_y}\right] \mathrm{vec}[\hat{Y} - Y] \geq \sigma^2_{\min}(G_H)(L(W, B) - L^*_{1:H}).$$

Therefore,

$$\begin{aligned}
\frac{d}{dt} L(W, B) &\leq -4\lambda_{W,B} \mathrm{vec}[\hat{Y} - Y]^\top \left[G_H^\top G_H \otimes I_{m_y}\right] \mathrm{vec}[\hat{Y} - Y] - \sum_{i=1}^{H} \left\| \sum_{l=i}^{H} J_{(i,l)} \mathrm{vec}[\nabla_{(l)} L(W, B)] \right\|_2^2 \\
&\leq -4\lambda_{W,B} \sigma^2_{\min}(G_H)(L(W, B) - L^*_{1:H}) - \sum_{i=1}^{H} \left\| \sum_{l=i}^{H} J_{(i,l)} \mathrm{vec}[\nabla_{(l)} L(W, B)] \right\|_2^2
\end{aligned}$$

Since $\frac{d}{dt} L^*_{1:H} = 0$,

$$\frac{d}{dt}(L(W, B) - L^*_{1:H}) \leq -4\lambda_{W,B} \sigma^2_{\min}(G_H)(L(W, B) - L^*_{1:H}) - \sum_{i=1}^{H} \left\| \sum_{l=i}^{H} J_{(i,l)} \mathrm{vec}[\nabla_{(l)} L(W, B)] \right\|_2^2$$

By defining $\mathbf{L} = L(W, B) - L^*_{1:H}$,

$$\frac{d\mathbf{L}}{dt} \leq -4\lambda_{W,B}\sigma^2_{\min}(G_H)\mathbf{L} - \sum_{i=1}^{H} \left\| \sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W, B)] \right\|_2^2 \tag{F.39}$$

Since $\frac{d}{dt}\mathbf{L} \leq 0$ and $\mathbf{L} \geq 0$, if $\mathbf{L} = 0$ at some time $\bar{t}$, then $\mathbf{L} = 0$ for any time $t \geq \bar{t}$. Therefore, if $\mathbf{L} = 0$ at some time $\bar{t}$, then we have the desired statement of this theorem for any time $t \geq \bar{t}$. Thus, we can focus on the time interval $[0, \bar{t}]$ such that $\mathbf{L} > 0$ for any time $t \in [0, \bar{t}]$ (here, it is allowed to have $\bar{t} = \infty$). Thus, focusing on the time interval with $\mathbf{L} > 0$, equation (F.39) implies that

$$\frac{1}{\mathbf{L}}\frac{d\mathbf{L}}{dt} \leq -4\lambda_{W,B}\sigma^2_{\min}(G_H) - \frac{1}{\mathbf{L}}\sum_{i=1}^{H} \left\| \sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W, B)] \right\|_2^2$$

By taking integral over time

$$\int_0^T \frac{1}{\mathbf{L}}\frac{d\mathbf{L}}{dt}dt \leq -\int_0^T 4\lambda_{W,B}\sigma^2_{\min}(G_H)dt - \int_0^T \frac{1}{\mathbf{L}}\sum_{i=1}^{H} \left\| \sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W, B)] \right\|_2^2 dt$$

By using the substitution rule for integrals, $\int_0^T \frac{1}{\mathbf{L}}\frac{d\mathbf{L}}{dt}dt = \int_{\mathbf{L}_0}^{\mathbf{L}_T} \frac{1}{\mathbf{L}}d\mathbf{L} = \log(\mathbf{L}_T) - \log(\mathbf{L}_0)$, where $\mathbf{L}_0 = L(W_0, B_0) - L^*_{1:H}$ and $\mathbf{L}_T = L(W_T, B_T) - L^*_{1:H}$. Thus,

$$\log(\mathbf{L}_T) - \log(\mathbf{L}_0) \leq -4\sigma^2_{\min}(G_H)\int_0^T \lambda_{W,B}dt - \int_0^T \frac{1}{\mathbf{L}}\sum_{i=1}^{H} \left\| \sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W, B)] \right\|_2^2 dt$$

which implies that

$$\begin{aligned} \mathbf{L}_T &\leq e^{\log(\mathbf{L}_0) - 4\sigma^2_{\min}(G_H)\int_0^T \lambda_{W,B}dt - \int_0^T \frac{1}{\mathbf{L}}\sum_{i=1}^{H}\left\|\sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W,B)]\right\|_2^2 dt} \\ &= \mathbf{L}_0 e^{-4\sigma^2_{\min}(G_H)\int_0^T \lambda_{W,B}dt - \int_0^T \frac{1}{\mathbf{L}}\sum_{i=1}^{H}\left\|\sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W,B)]\right\|_2^2 dt} \end{aligned}$$

By recalling the definition of $\mathbf{L} = L(W, B) - L^*_{1:H}$ and that $\frac{d}{dt}\mathbf{L} \leq 0$, we have that if $L(W_T, B_T) - L^*_{1:H} > 0$, then $L(W_t, B_t) - L^*_{1:H} > 0$ for all $t \in [0, T]$, and

$$L(W_T, B_T) - L^*_{1:H} \leq (L(W_0, B_0) - L^*_{1:H})e^{-4\sigma^2_{\min}(G_H)\int_0^T \lambda_{W_t,B_t}dt - \int_0^T \frac{1}{L(W_t,B_t)-L^*}\sum_{i=1}^{H}\left\|\sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W_t,B_t)]}$$

By noticing that $\lambda_T^{(1:H)} = \inf_{t\in[0,T]} \lambda_{W_t,B_t}$ and that

$$\int_0^T \frac{1}{L(W_t,B_t)-L^*} \sum_{i=1}^H \left\| \sum_{l=i}^H J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W_t,B_t)] \right\|_2^2 dt \ge 0$$

, this implies that

$$\begin{aligned} L(W_T,B_T)-L^*_{1:H} &\le (L(W_0,B_0)-L^*_{1:H}) e^{-4\lambda_T^{(1:H)}\sigma^2_{\min}(G_H)T - \int_0^T \frac{1}{L(W_t,B_t)-L^*} \sum_{i=1}^H \left\| \sum_{l=i}^H J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W_t,B_t)] \right\|_2^2 dt} \\ &\le (L(W_0,B_0)-L^*_{1:H}) e^{-4\lambda_T^{(1:H)}\sigma^2_{\min}(G_H)T}. \end{aligned}$$

This completes the proof of Theorem 9.4 (i) for the case of $\mathcal{I}=[n]$. Since every step in this proof is valid when we replace $f(X,W,B)$ by $f(X,W,B)_{*\mathcal{I}}$ and XS^l by $X(S^l)_{*\mathcal{I}}$ without using any assumption on S or the relation between S^{l-1} and S, our proof also yields for the general case of $\mathcal{I}$ that

$$L(W_T,B_T)-L^*_{1:H} \le (L(W_0,B_0)-L^*_{1:H}) e^{-4\lambda_T^{(1:H)}\sigma^2_{\min}((G_H)_{*\mathcal{I}})T}.$$

$\square$

Case (ii): Completing The Proof of Theorem 9.4 (ii)

Using equation (F.37) and (F.38) , we have that for any $H' \in \{0,1,\dots,H\}$,

$$\begin{aligned} \frac{d}{dt} L(W,B) &\le -\lambda_{\min}(F_{(H')}) \| \operatorname{vec}[\nabla_{(H')} L(W,B)] \|_2^2 \\ &\le -4\lambda_{\min}(F_{(H')}) \operatorname{vec}[\hat{Y}-Y]^\top [(XS^{H'})^\top XS^{H'} \otimes I_{m_y}] \operatorname{vec}[\hat{Y}-Y] \\ &= -4\lambda_{W,B} \operatorname{vec}[\hat{Y}-Y]^\top \left[\tilde{G}_{H'}^\top \tilde{G}_{H'} \otimes I_{m_y}\right] \operatorname{vec}[\hat{Y}-Y], \end{aligned}$$

where

$$\lambda_{W,B} := \lambda_{\min}(F_{(H')}),$$

and

$$\tilde{G}_{H'} := XS^{H'}.$$

Decompose $\mathrm{vec}[\hat{Y}-Y]$ as $\mathrm{vec}[\hat{Y}-Y] = v + v^{\perp}$, where $v = \mathbf{P}_{\tilde{G}_{H'}^{\top}\otimes I_{m_y}} \mathrm{vec}[\hat{Y}-Y]$, $v^{\perp} = (I_{m_y n} - \mathbf{P}_{\tilde{G}_{H'}^{\top}\otimes I_{m_y}}) \mathrm{vec}[\hat{Y}-Y]$, and $\mathbf{P}_{\tilde{G}_{H'}^{\top}\otimes I_{m_y}} \in \mathbb{R}^{m_y n \times m_y n}$ represents the orthogonal projection onto the column space of $\tilde{G}_{H'}^{\top} \otimes I_{m_y} \in \mathbb{R}^{m_y n \times m_y m_x}$. Then,

$$\begin{aligned}
\mathrm{vec}[\hat{Y}-Y]^{\top}\left[\tilde{G}_{H'}^{\top}\tilde{G}_{H'} \otimes I_{m_y}\right]\mathrm{vec}[\hat{Y}-Y] &= (v+v^{\perp})^{\top}\left[\tilde{G}_{H'}^{\top} \otimes I_{m_y}\right]\left[\tilde{G}_{H'} \otimes I_{m_y}\right](v+v^{\perp}) \\
&= v^{\top}\left[\tilde{G}_{H'}^{\top} \otimes I_{m_y}\right]\left[\tilde{G}_{H'} \otimes I_{m_y}\right]v \\
&\geq \sigma_{\min}^2(\tilde{G}_{H'})\|\mathbf{P}_{\tilde{G}_{H'}^{\top}\otimes I_{m_y}} \mathrm{vec}[\hat{Y}-Y]\|_2^2 \\
&= \sigma_{\min}^2(\tilde{G}_{H'})\|\mathbf{P}_{\tilde{G}_{H'}^{\top}\otimes I_{m_y}} \mathrm{vec}[\hat{Y}] - \mathbf{P}_{\tilde{G}_{H'}^{\top}\otimes I_{m_y}} \mathrm{vec}[Y]\|_2^2 \\
&= \sigma_{\min}^2(\tilde{G}_{H'})\| \mathrm{vec}[\hat{Y}] - \mathbf{P}_{\tilde{G}_{H'}^{\top}\otimes I_{m_y}} \mathrm{vec}[Y] \pm \mathrm{vec}[Y]\|_2^2 \\
&= \sigma_{\min}^2(\tilde{G}_{H'})\| \mathrm{vec}[\hat{Y}] - \mathrm{vec}[Y] + (I_{m_y n} - \mathbf{P}_{\tilde{G}_{H'}^{\top}\otimes I_{m_y}}) \mathrm{vec}[Y]\|_2^2 \\
&\geq \sigma_{\min}^2(\tilde{G}_{H'})(\| \mathrm{vec}[\hat{Y}-Y]\|_2 - \|(I_{m_y n} - \mathbf{P}_{\tilde{G}_{H'}^{\top}\otimes I_{m_y}}) \mathrm{vec}[Y]\|_2)^2 \\
&\geq \sigma_{\min}^2(\tilde{G}_{H'})(\| \mathrm{vec}[\hat{Y}-Y]\|_2^2 - \|(I_{m_y n} - \mathbf{P}_{\tilde{G}_{H'}^{\top}\otimes I_{m_y}}) \mathrm{vec}[Y]\|_2^2,
\end{aligned}$$

where we used the fact that the singular values of $\left[\tilde{G}_{H'}^{\top} \otimes I_{m_y}\right]$ are products of singular values of $\tilde{G}_{H'}$ and I_{m_y}.

By noticing that $L(W,B) = \| \mathrm{vec}[\hat{Y}-Y]\|_2^2$ and $L_{H'}^{*} = \|(I_{m_y n} - \mathbf{P}_{\tilde{G}_{H'}^{\top}\otimes I_{m_y}}) \mathrm{vec}[Y]\|_2^2$, we have that for any $H' \in \{0,1,\ldots,H\}$,

$$\mathrm{vec}[\hat{Y}-Y]^{\top}\left[\tilde{G}_{H'}^{\top}\tilde{G}_{H'} \otimes I_{m_y}\right]\mathrm{vec}[\hat{Y}-Y] \geq \sigma_{\min}^2(\tilde{G}_{H'})(L(W,B) - L_{H'}^{*}). \tag{F.40}$$

Therefore,

$$\begin{aligned}
\frac{d}{dt}L(W,B) &\leq -4\lambda_{W,B}\,\mathrm{vec}[\hat{Y}-Y]^{\top}\left[\tilde{G}_{H'}^{\top}\tilde{G}_{H'} \otimes I_{m_y}\right]\mathrm{vec}[\hat{Y}-Y] \\
&\leq -4\lambda_{W,B}\sigma_{\min}^2(\tilde{G}_{H'})(L(W,B) - L_{H'}^{*})
\end{aligned}$$

Since $\frac{d}{dt}L_{H'}^{*} = 0$,

$$\frac{d}{dt}(L(W,B) - L_{H'}^{*}) \leq -4\lambda_{W,B}\sigma_{\min}^2(\tilde{G}_{H'})(L(W,B) - L_{H'}^{*})$$

By defining $\mathbf{L} = L(W, B) - L^*_{H'}$,

$$\frac{d\mathbf{L}}{dt} \leq -4\lambda_{W,B}\sigma^2_{\min}(\tilde{G}_{H'})\mathbf{L} \tag{F.41}$$

Since $\frac{d}{dt}\mathbf{L} \leq 0$ and $\mathbf{L} \geq 0$, if $\mathbf{L} = 0$ at some time $\bar{t}$, then $\mathbf{L} = 0$ for any time $t \geq \bar{t}$. Therefore, if $\mathbf{L} = 0$ at some time $\bar{t}$, then we have the desired statement of this theorem for any time $t \geq \bar{t}$. Thus, we can focus on the time interval $[0, \bar{t}]$ such that $\mathbf{L} > 0$ for any time $t \in [0, \bar{t}]$ (here, it is allowed to have $\bar{t} = \infty$). Thus, focusing on the time interval with $\mathbf{L} > 0$, equation (F.41) implies that

$$\frac{1}{\mathbf{L}}\frac{d\mathbf{L}}{dt} \leq -4\lambda_{W,B}\sigma^2_{\min}(\tilde{G}_{H'})$$

By taking integral over time

$$\int_0^T \frac{1}{\mathbf{L}}\frac{d\mathbf{L}}{dt}dt \leq -\int_0^T 4\lambda_{W,B}\sigma^2_{\min}(\tilde{G}_{H'})dt$$

By using the substitution rule for integrals, $\int_0^T \frac{1}{\mathbf{L}}\frac{d\mathbf{L}}{dt}dt = \int_{\mathbf{L}_0}^{\mathbf{L}_T} \frac{1}{\mathbf{L}}d\mathbf{L} = \log(\mathbf{L}_T) - \log(\mathbf{L}_0)$, where $\mathbf{L}_0 = L(W_0, B_0) - L^*$ and $\mathbf{L}_T = L(W_T, B_T) - L^*_{H'}$. Thus,

$$\log(\mathbf{L}_T) - \log(\mathbf{L}_0) \leq -4\sigma^2_{\min}(\tilde{G}_{H'})\int_0^T \lambda_{W,B}dt$$

which implies that

$$\begin{aligned}\mathbf{L}_T &\leq e^{\log(\mathbf{L}_0) - 4\sigma^2_{\min}(\tilde{G}_{H'})\int_0^T \lambda_{W,B}dt} \\ &= \mathbf{L}_0 e^{-4\sigma^2_{\min}(\tilde{G}_{H'})\int_0^T \lambda_{W,B}dt}\end{aligned}$$

By recalling the definition of $\mathbf{L} = L(W, B) - L^*_{H'}$ and that $\frac{d}{dt}\mathbf{L} \leq 0$, we have that if $L(W_T, B_T) - L^*_{H'} > 0$, then $L(W_t, B_t) - L^*_{H'} > 0$ for all $t \in [0, T]$, and

$$L(W_T, B_T) - L^*_{H'} \leq (L(W_0, B_0) - L^*_{H'})e^{-4\sigma^2_{\min}(\tilde{G}_{H'})\int_0^T \lambda_{W_t,B_t}dt}.$$

By noticing that $\lambda_T^{(H')} = \inf_{t\in[0,T]} \lambda_{W_t,B_t}$, this implies that for any $H' \in \{0, 1, \dots, H\}$,

$$\begin{aligned} L(W_T, B_T) - L^*_{H'} &\leq (L(W_0, B_0) - L^*_{H'})e^{-4\lambda_T^{(H')}\sigma^2_{\min}(\tilde{G}_{H'})T} \\ &= (L(W_0, B_0) - L^*_{H'})e^{-4\lambda_T^{(H)}\sigma^2_{\min}(XS^{H'})T} \end{aligned}$$

This completes the proof of Theorem 9.4 (ii) for the case of $\mathcal{I} = [n]$. Since every step in this proof is valid when we replace $f(X, W, B)$ by $f(X, W, B)_{*\mathcal{I}}$ and XS^l by $X(S^l)_{*\mathcal{I}}$ without using any assumption on S or the relation between S^{l-1} and S, our proof also yields for the general case of $\mathcal{I}$ that

$$L(W_T, B_T) - L^*_{H'} \leq (L(W_0, B_0) - L^*_{H'})e^{-4\lambda_T^{(H)}\sigma^2_{\min}(X(S^{H'})_{*\mathcal{I}})T}$$

$\square$

Case (iii): Completing The Proof of Theorem 9.4 (iii)

In this case, we have the following assumption: there exist $l, l' \in \{0, \dots, H\}$ with $l < l'$ such that $L^*_l \geq L^*_{l+1} \geq \cdots \geq L^*_{l'}$ or $L^*_l \leq L^*_{l+1} \leq \cdots \leq L^*_{l'}$. Using equation (F.37) and (F.38) with $\tilde{G}_l = XS^l$, we have that

$$\begin{aligned} \frac{d}{dt}L(W, B) &\leq -\sum_{l=0}^{H} \lambda_{\min}(F_{(l)}) \| \operatorname{vec}[\nabla_{(l)} L(W, B)]\|_2^2 \\ &\leq -4\sum_{l=0}^{H} \lambda_{\min}(F_{(l)}) \operatorname{vec}[\hat{Y} - Y]^\top [(XS^l)^\top XS^l \otimes I_{m_y}] \operatorname{vec}[\hat{Y} - Y] \\ &= -4\sum_{l=0}^{H} \lambda_{\min}(F_{(l)}) \operatorname{vec}[\hat{Y} - Y]^\top [\tilde{G}_l^\top \tilde{G}_l \otimes I_{m_y}] \operatorname{vec}[\hat{Y} - Y] \end{aligned}$$

Using (F.40), since $\operatorname{vec}[\hat{Y} - Y]^\top \left[\tilde{G}_l^\top \tilde{G}_l \otimes I_{m_y}\right] \operatorname{vec}[\hat{Y} - Y] \geq \sigma^2_{\min}(\tilde{G}_l)(L(W, B) - L^*_l)$ for any $l \in \{0, 1, \dots, H\}$,

$$\frac{d}{dt}L(W, B) \leq -4\sum_{l=0}^{H} \lambda_{\min}(F_{(l)})\sigma^2_{\min}(\tilde{G}_l)(L(W, B) - L^*_l). \tag{F.42}$$

Let $l'' = l$ if $L^*_l \geq L^*_{l+1} \geq \cdots \geq L^*_{l'}$, and $l'' = l'$ if $L^*_l \leq L^*_{l+1} \leq \cdots \leq L^*_{l'}$. Then, using

(F.42) and the assumption of $L_l^* \geq L_{l+1}^* \geq \cdots \geq L_{l'}^*$ or $L_l^* \leq L_{l+1}^* \leq \cdots \leq L_{l'}^*$ for some $l, l' \in \{0, \dots, H\}$, we have that

$$\frac{d}{dt} L(W,B) \leq -4(L(W,B) - L_{l''}^*) \sum_{k=l}^{l'} \lambda_{\min}(F_{(k)}) \sigma_{\min}^2(\tilde{G}_k). \tag{F.43}$$

Since $\frac{d}{dt} L_{l''}^* = 0$,

$$\frac{d}{dt}(L(W,B) - L_{l''}^*) \leq -4(L(W,B) - L_{l''}^*) \sum_{k=l}^{l'} \lambda_{\min}(F_{(k)}) \sigma_{\min}^2(\tilde{G}_k).$$

By taking integral over time in the same way as that in the proof for the case of (i) and (ii), we have that

$$L(W_T, B_T) - L_{l''}^* \leq (L(W_0, B_0) - L_{l''}^*) e^{-4 \sum_{k=l}^{l'} \sigma_{\min}^2(\tilde{G}_k) \int_0^T \lambda_{\min}(F_{(k),t}) dt} \tag{F.44}$$

Using the property of Kronecker product,

$$\lambda_{\min}(F_{(l),t}) = \lambda_{\min}([(B_{(l),t} \dots B_{(1),t})^\top B_{(l),t} \cdots B_{(1),t} \otimes I_{m_y}]) = \lambda_{\min}((B_{(l),t} \dots B_{(1),t})^\top B_{(l),t} \cdots B_{(1),t}),$$

which implies that $\lambda_T^{(k)} = \inf_{t \in [0,T]} \lambda_{\min}(F_{(k),t})$. Therefore, equation (F.44) with $\lambda_T^{(k)} = \inf_{t \in [0,T]} \lambda_{\min}(F_{(k),t})$ yields that

$$\begin{aligned} L(W_T, B_T) - L_{l''}^* &\leq (L(W_0, B_0) - L_{l''}^*) e^{-4 \sum_{k=l}^{l'} \lambda_T^{(k)} \sigma_{\min}^2(\tilde{G}_k) T} \\ &= (L(W_0, B_0) - L_{l''}^*) e^{-4 \sum_{k=l}^{l'} \lambda_T^{(k)} \sigma_{\min}^2(X S^k) T} \end{aligned} \tag{F.45}$$

This completes the proof of Theorem 9.4 (iii) for the case of $\mathcal{I} = [n]$. Since every step in this proof is valid when we replace $f(X,W,B)$ by $f(X,W,B)_{*\mathcal{I}}$ and XS^l by $X(S^l)_{*\mathcal{I}}$ without using any assumption on S or the relation between S^{l-1} and S, our proof also yields for the general case of $\mathcal{I}$ that

$$L(W_T, B_T) - L_{l''}^* \leq (L(W_0, B_0) - L_{l''}^*) e^{-4 \sum_{k=l}^{l'} \lambda_T^{(k)} \sigma_{\min}^2(X(S^k)_{*\mathcal{I}}) T}.$$

□

F.1.5 Proof of Proposition 9.5

From Definition 9.4, for any $l \in \{1, 2, \dots, H\}$, we have that $\sigma_{\min}(\bar{B}^{(1:l)}) = \sigma_{\min}(B_{(l)}B_{(l-1)}\cdots B_{(1)}) \geq \gamma$ for all (W, B) such that $L(W, B) \leq L(W_0, B_0)$. From equation (F.37) in the proof of Theorem 9.4, it holds that $\frac{d}{dt}L(W_t, B_t) \leq 0$ for all t. Thus, we have that $L(W_t, B_t) \leq L(W_0, B_0)$ and hence $\sigma_{\min}(\bar{B}_t^{(1:l)}) \geq \gamma$ for all t. Under this problem setting ($m_l \geq m_x$), this implies that $\lambda_{\min}((\bar{B}_t^{(1:l)})^\top \bar{B}_t^{(1:l)}) \geq \gamma^2$ for all t and thus $\lambda_T^{(1:H)} \geq \gamma^2$.

F.1.6 Proof of Theorem 9.6

The proof of Theorem 9.6 follows from the intermediate results of the proofs of Theorem 9.1 and Theorem 9.4 as we show in the following. For the non-multiscale case, from equation (F.18) in the proof of Theorem 9.1, we have that

$$\frac{d}{dt}L_1(W, B) = -\|\operatorname{vec}[\nabla_{(H)}L(W, B)]\|_{F_{(H)}}^2 - \sum_{i=1}^{H}\left\|J_{(i,H)}\operatorname{vec}[\nabla_{(H)}L(W, B)]\right\|_2^2$$

where

$$\|\operatorname{vec}[\nabla_{(H)}L(W, B)]\|_{F_{(H)}}^2 := \operatorname{vec}[\nabla_{(H)}L(W, B)]^\top F_{(H)}\operatorname{vec}[\nabla_{(H)}L(W, B)].$$

Since equation (F.18) in the proof of Theorem 9.4 is derived without the assumption on the square loss, this holds for any differentiable loss ℓ. By noticing that $\nabla_{(H)}L(W, B) = V(X(S^H)_{*\mathcal{I}})^\top$, we have that

$$\frac{d}{dt}L_1(W, B) = -\|\operatorname{vec}[V(X(S^H)_{*\mathcal{I}})^\top]\|_{F_{(H)}}^2 - \sum_{i=1}^{H}\left\|J_{(i,H)}\operatorname{vec}[V(X(S^H)_{*\mathcal{I}})^\top]\right\|_2^2.$$

This proves the statement of Theorem 9.6 (i).

For the multiscale case, from equation (F.36) in the proof of Theorem 9.4, we have that

$$\frac{d}{dt}L_2(W,B) = -\sum_{l=0}^{H} \| \operatorname{vec}[\nabla_{(l)} L(W,B)] \|^2_{F_{(l)}} - \sum_{i=1}^{H} \left\| \sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[\nabla_{(l)} L(W,B)] \right\|_2^2 \tag{F.46}$$

where

$$\| \operatorname{vec}[\nabla_{(l)} L(W,B)] \|^2_{F_{(l)}} := \operatorname{vec}[\nabla_{(l)} L(W,B)]^\top F_{(l)} \operatorname{vec}[\nabla_{(l)} L(W,B)].$$

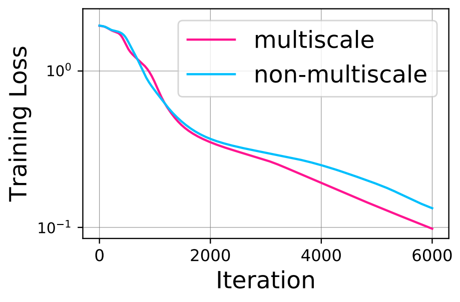
Since equation (F.36) in the proof of Theorem 9.4 is derived without the assumption on the square loss, this holds for any differentiable loss ℓ. Since every step to derive equation (F.36) is valid when we replace $f(X,W,B)$ by $f(X,W,B)_{*\mathcal{I}}$ and XS^l by $X(S^l)_{*\mathcal{I}}$ without using any assumption on S or the relation between S^{l-1} and S, the steps to derive equation (F.36) also yields this for the general case of $\mathcal{I}$: i.e., $\nabla_{(l)} L(W,B) = V(X(S^l)_{*\mathcal{I}})^\top$. Thus, we have that

$$\frac{d}{dt}L_1(W,B) = -\sum_{l=0}^{H} \| \operatorname{vec}[V(X(S^l)_{*\mathcal{I}})^\top] \|^2_{F_{(l)}} - \sum_{i=1}^{H} \left\| \sum_{l=i}^{H} J_{(i,l)} \operatorname{vec}[V(X(S^l)_{*\mathcal{I}})^\top] \right\|_2^2$$

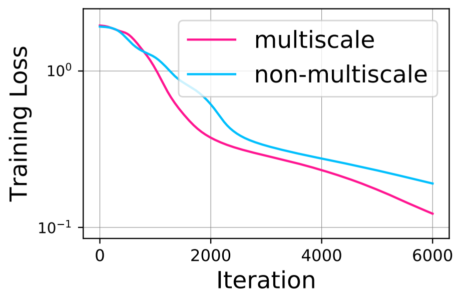
This completes the proof of Theorem 9.6 (ii).

F.2 Additional Experimental Results

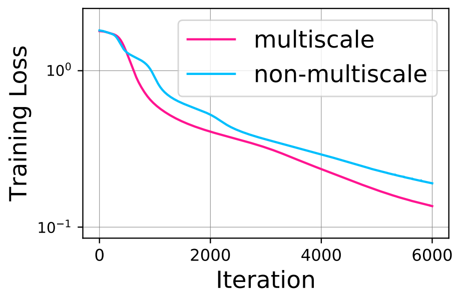
In this section, we present additional experimental results.



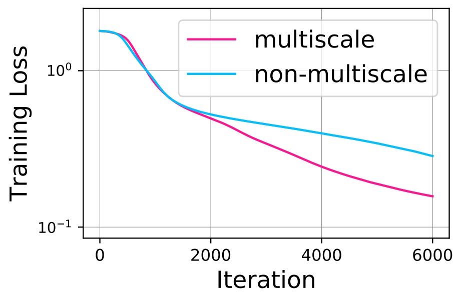
(a) Linear and Cora.



(b) ReLU and Cora.

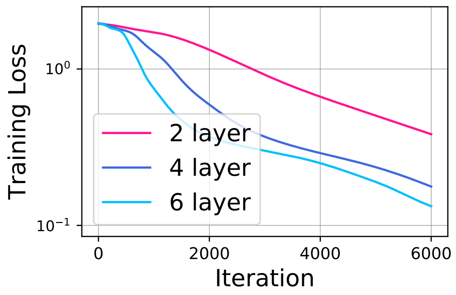


(c) Linear and Citeseer.

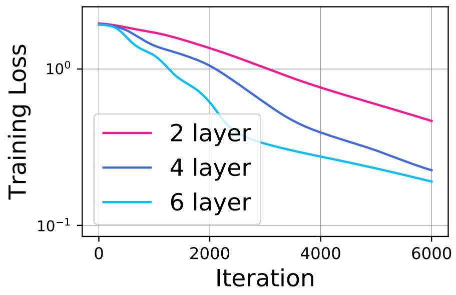


(d) ReLU and Citeseer.

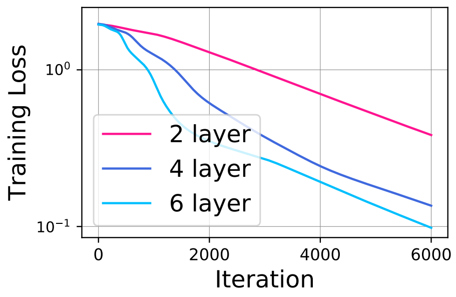
Figure F-1: **Multiscale skip connection accelerates GNN training**. We plot the training curves of GNNs with ReLU and linear activation on the *Cora* and *Citeseer* dataset. We use the GCN model with learning rate $5e-5$, six layers, and hidden dimension 32.



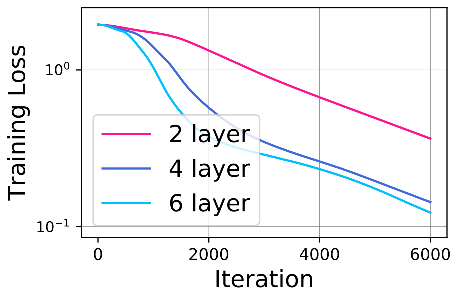
(a) Linear and non-multiscale.



(b) ReLU and non-multiscale.

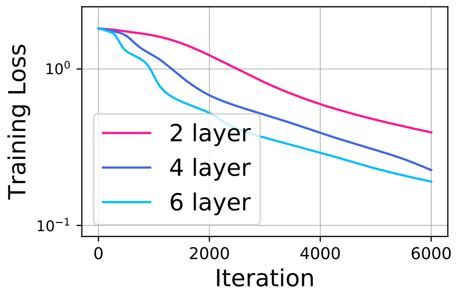


(c) Linear and multiscale.

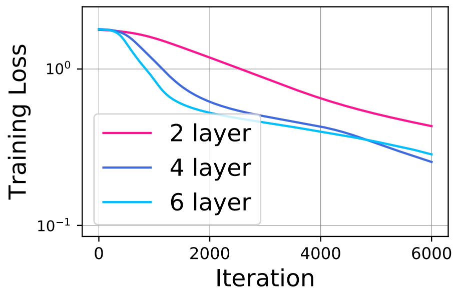


(d) ReLU and multiscale.

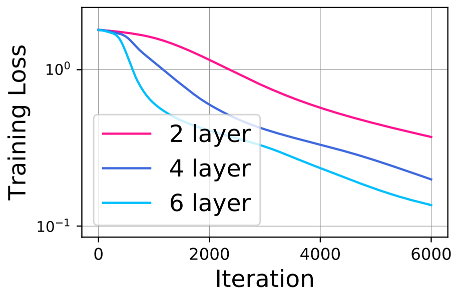
Figure F-2: **Depth accelerates GNN training**. We plot the training curves of GNNs with ReLU and linear activation, multiscale and non-multiscale on the **Cora** dataset. We use the GCN model with learning rate $5e - 5$ and hidden dimension 32.



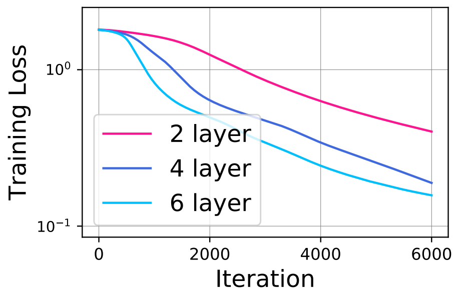
(a) Linear and non-multiscale.



(b) ReLU and non-multiscale.

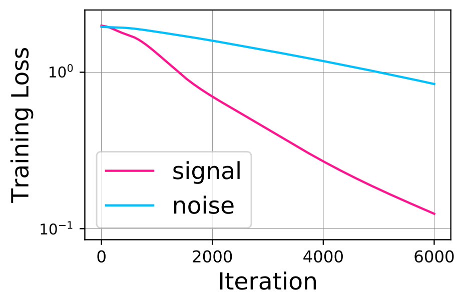


(c) Linear and multiscale.

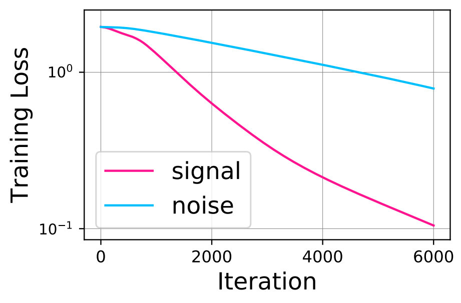


(d) ReLU and multiscale.

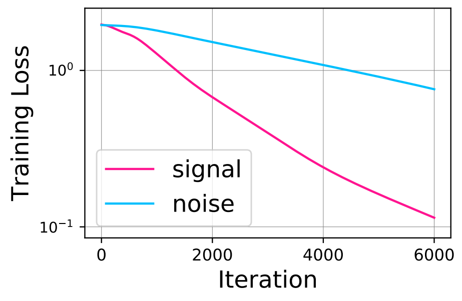
Figure F-3: **Depth accelerates GNN training**. We plot the training curves of GNNs with ReLU and linear activation, multiscale and non-multiscale on the **Citeseer** dataset. We use the GCN model with learning rate $5e-5$ and hidden dimension 32.



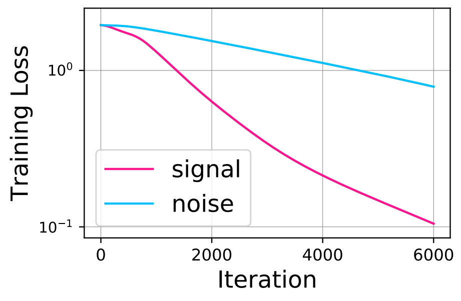
(a) Linear and non-multiscale.



(b) ReLU and non-multiscale.

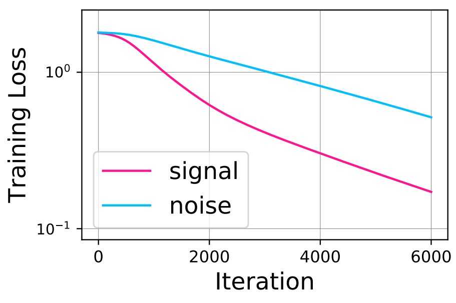


(c) Linear and multiscale.

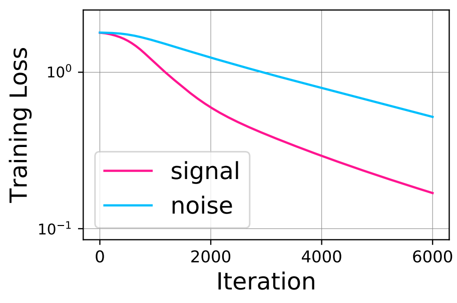


(d) ReLU and multiscale.

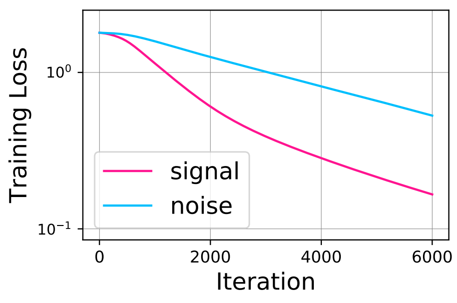
Figure F-4: **GNNs train faster when the labels have signal instead of random noise**. We plot the training curves of multiscale and non-multiscale GNNs with ReLU and linear activation, on the **Cora** dataset. We use the two-layer GCN model with learning rate $1e-4$ and hidden dimension 32.



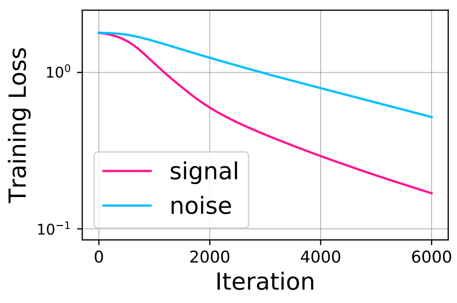
(a) Linear and non-multiscale.



(b) ReLU and non-multiscale.

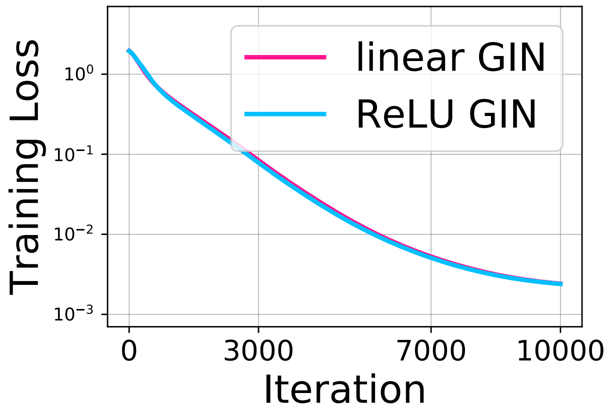


(c) Linear and multiscale.

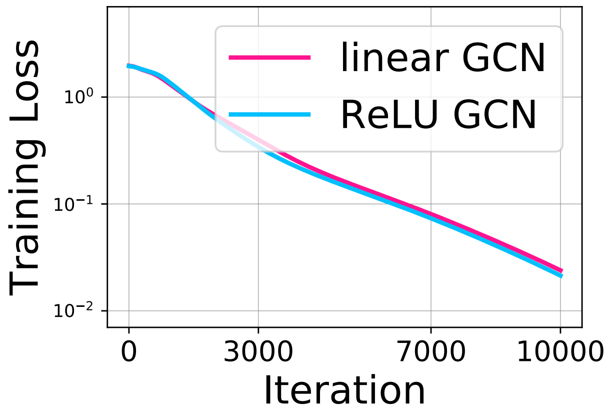


(d) ReLU and multiscale.

Figure F-5: **GNNs train faster when the labels have signal instead of random noise**. We plot the training curves of multiscale and non-multiscale GNNs with ReLU and linear activation, on the **Citeseer** dataset. We use the two-layer GCN model with learning rate $1e-4$ and hidden dimension 32.



(a) Linear GIN vs. ReLU GIN.



(b) Linear GCN vs. ReLU GCN.

Figure F-6: **Linear GNNs vs. ReLU GNNs**. We plot the training curves of GCN and GIN with ReLU and linear activation on the Cora dataset. The training curves of linear GNNs and ReLU GNNs are similar, both converging to nearly zero training loss with the same linear rate. Moreover, GIN trains faster than GCN, which agrees with our bound in Theorem 9.1. We use the learning rate $1e-4$, two layers, and hidden dimension 32.

F.3 Experimental Setup

In this section, we describe the experimental setup for reproducing our experiments.

Dataset. We perform all experiments on the Cora and Citeseer datasets Sen et al. [2008]. Cora and Citeer are citation networks and the goal is to classify academic documents into different subjects. The dataset contains bag-of-words features for each document (node) and citation links (edges) between documents. The tasks are semi-supervised node classification. Only a subset of nodes have training labels. In our experiments, we use the default dataset split, i.e., which nodes have training labels, and minimize the training loss accordingly. Tabel F.1 shows an overview of the dataset statistics.

Dataset	Nodes	Edges	Classes	Features
Citeseer	3,327	4,732	6	3,703
Cora	2,708	5,429	7	1,433

Table F.1: Dataset statistics

Training details. We describe the training settings for our experiments. Let us first describe some common hyperparameters and settings, and then for each experiment or figure we describe the other hyperparameters. For our experiments, to more closely align with the common practice in GNN training, we use the Adam optimizer and keep optimizer-specific hyperparameters except initial learning rate default. We set weight decay to zero. Next, we describe the settings for each experiment respectively.

For the experiment in Figure 9-1, i.e., the training curves of linear vs. ReLU GNNs, we train the GCN and GIN with two layers on Cora with cross-entropy loss and learning rate 1e-4. We set the hidden dimension to 32.

For the experiment in Figure 9-2a, i.e., computing the graph condition for linear GNNs, we use the linear GCN and GIN model with three layers on Cora and Citeseer. For linear GIN, we set ϵ to zero and MLP layer to one.

For the experiment in Figure 9-2b, i.e., computing and plotting the time-dependent condition for linear GNNs, we train a linear GCN with two layers on Cora with squared loss and learning rate 1e-4. We set the hidden dimension the input dimension for both Cora and for CiteSeer, because the global convergence theorem requires the hidden dimension to be at least the same as input dimension. Note that this requirement is standard in previous works as well, such as Arora et al. [2019a]. We use the default random initialization of PyTorch. The formula for computing the time-dependent λ_T is given in the main paper.

For the experiment in Figure 9-2c, i.e., computing and plotting the time-dependent condition for linear GNNs across multiple training settings, we consider the following settings:

1. Dataset: Cora and Citeseer.

2. Model: GCN and GIN.

3. Depth: Two and four layers.

4. Activation: Linear and ReLU.

We train the GNN with the settings above with squared loss and learning rate 1e-4. We set the hidden dimension to input dimension for Cora and CiteSeer. We use the default random initialization of PyTorch. The formula for computing the time-dependent λ_T is given in the main paper. For each point, we report the λ_T at last epoch.

For the experiment in Figure 9-3a, i.e., computing the graph condition for multiscale linear GNNs, we use the linear GCN and GIN model with three layers on Cora and Citeseer. For linear GIN, we set ϵ to zero and MLP layer to one.

For the experiment in Figure 9-3b, i.e., computing and plotting the time-dependent condition for multiscale linear GNNs, we train a linear GCN with two layers on Cora with squared loss and learning rate 1e-4. We set the hidden dimension to 2000 for Cora and 4000 for CiteSeer. We use the default random initialization of PyTorch. The formula for computing the time-dependent λ_T is given in the main paper.

For the experiment in Figure 9-3c, i.e., computing and plotting the time-dependent condition for multiscale linear GNNs across multiple training settings, we consider the following settings:

1. Dataset: Cora and Citeseer.

2. Model: Multiscale GCN and GIN.

3. Depth: Two and four layers.

4. Activation: Linear and ReLU.

We train the multiscale GNN with the settings above with squared loss and learning rate 1e-4. We set the hidden dimension to 2000 for Cora and 4000 for CiteSeer. We use the default random initialization of PyTorch. The formula for computing the time-dependent λ_T is given in the main paper. For each point, we report the λ_T at last epoch.

For the experiment in Figure 9-4a, i.e., multiscale vs. non-multiscale, we train the GCN with six layers and ReLU activation on Cora with cross-entropy loss and learning rate 5e-5. We set the hidden dimension to 32.

We perform more extensive experiments to verify the conclusion for multiscale vs. non-multiscale in Figure F-6. There, we train the GCN with six layers with both ReLU and linear activation on both Cora and Citeseer with cross-entropy loss and learning rate 5e-5. We set the hidden dimension to 32.

For the experiment in Figure 9-4b, i.e., acceleration with depth, we train the non-multiscale GCN with two, four, six layers and ReLU activation on Cora with cross-entropy loss and learning rate 5e-5. We set the hidden dimension to 32.

We perform more extensive experiments to verify the conclusion for acceleration with depth in Figure F-2 and Figure F-3. There, we train both multiscale and non-multiscale GCN with 2, 4, 6 layers with both ReLU and linear activation on both Cora and Citeseer with cross-entropy loss and learning rate 5e-5. We set the hidden dimension to 32.

For the experiment in Figure 9-4c, i.e., signal vs. noise, we train the non-multiscale GCN with two layers and ReLU activation on Cora with cross-entropy loss and learning rate 1e-4. We set the hidden dimension to 32. For signal, we use the default labels of Cora. For noise, we randomly choose a class as the label.

We perform more extensive experiments to verify the conclusion for signal vs. noise in Figure F-4 and Figure F-5. There, we train both multiscale and non-multiscale GCN with two layers with both ReLU and linear activation on both Cora and Citeseer with cross-entropy loss and learning rate 1e-4. We set the hidden dimension to 32.

For the experiment in Figure 9-5, i.e., first term vs. second term, we use the same setting as in Figure 9-4c. We use the formula of our Theorem in the main paper.

Computing resources. The computing hardware is based on the CPU and the NVIDIA GeForce RTX 1080 Ti GPU. The software implementation is based on PyTorch and PyTorch Geometric [Fey and Lenssen, 2019]. For all experiments, we train the GNNs with CPU and compute the eigenvalues with GPU.

Appendix G

Accelerating Training with GraphNorm

G.1 Proofs

G.1.1 Proof of Theorem 10.1

We first introduce the Cauchy interlace theorem:

Lemma G.1 (Cauchy interlace theorem (Theorem 4.3.17 in Horn and Johnson [2012])). *Let $S \in \mathbb{R}^{(n-1)\times(n-1)}$ be symmetric, $y \in \mathbb{R}^n$ and $a \in \mathbb{R}$ be given, and let $R = \begin{pmatrix} S & y \\ y^\top & a \end{pmatrix} \in \mathbb{R}^{n\times n}$. Let $\lambda_1 \le \lambda_2 \le \cdots \le \lambda_n$ be the eigenvalues of R and $\mu_1 \le \mu_2 \le \cdots \le \mu_{n-1}$ be the eigenvalues of S. Then*

$$\lambda_1 \le \mu_1 \le \lambda_2 \le \cdots \le \lambda_{n-1} \le \mu_{n-1} \le \lambda_n, \tag{G.1}$$

where $\lambda_i = \mu_i$ only when there is a nonzero $z \in \mathbb{R}^{n-1}$ such that $Sz = \mu_i z$ and $y^\top z = 0$; if $\lambda_i = \mu_{i-1}$ then there is a nonzero $z \in \mathbb{R}^{n-1}$ such that $Sz = \mu_{i-1} z$, $y^\top z = 0$.

Using Lemma G.1, the theorem can be proved as below.

Proof. For any matrices $P, R \in \mathbb{R}^{n\times n}$, we use $P \sim R$ to denote that the matrix P is similar to the matrix R. Note that if $P \sim R$, the eigenvalues of P and R are the same. As the singular values of P are equal to the square root of the eigenvalues of $P^\top P$, we have the eigenvalues of $Q^\top Q$ and that of $NQ^\top QN$ are $\{\lambda_i^2\}_{i=1}^n$ and $\{\mu_i^2\}_{i=1}^n$, respectively.

Note that N is a projection operator onto the orthogonal complement space of the subspace spanned by $\mathbf{1}$, and N can be decomposed as $N = U \operatorname{diag}\left(\underbrace{1, \cdots, 1}_{\times n-1}, 0\right) U^\top$ where U is an orthogonal matrix. Since $\mathbf{1}$ is the eigenvector of N associated with eigenvalue 0, we have

$$U = \begin{pmatrix} U_1 & \frac{1}{\sqrt{n}}\mathbf{1} \end{pmatrix}, \tag{G.2}$$

where $U_1 \in \mathbb{R}^{n\times(n-1)}$ satisfies $U_1\mathbf{1} = 0$ and $U_1^\top U_1 = I_{n-1}$.

Then we have $NQ^\top QN = U \operatorname{diag}(1, \cdots, 1, 0)\, U^\top Q^\top QU \operatorname{diag}(1, \cdots, 1, 0)\, U^\top \sim \operatorname{diag}(1, \cdots, 1, 0)\, U^\top Q^\top QU \operatorname{diag}(1, \cdots, 1, 0)$.

Let

$$D = \operatorname{diag}(1, \cdots, 1, 0) = \begin{pmatrix} I_{n-1} & \mathbf{0} \\ \mathbf{0}^\top & 0 \end{pmatrix}, \tag{G.3}$$

$$B = \begin{pmatrix} I_{n-1} \\ \mathbf{0}^\top \end{pmatrix}, \tag{G.4}$$

$$\bar{C} = Q^\top Q, \tag{G.5}$$

where $\mathbf{0} = \left[\underbrace{0, \cdots, 0}_{\times n-1}\right]^\top$.

We have

$$
\begin{aligned}
NQ^\top QN &\sim DU^\top \bar{C}UD && \text{(G.6)}\\
&= D\begin{pmatrix} U_1^\top \\ \frac{1}{\sqrt{n}}\mathbf{1}^\top \end{pmatrix} \bar{C} \begin{pmatrix} U_1 & \frac{1}{\sqrt{n}}\mathbf{1} \end{pmatrix} D && \text{(G.7)}\\
&= D\begin{pmatrix} U_1^\top \bar{C}U_1 & \frac{1}{\sqrt{n}}U_1^\top \bar{C}\mathbf{1} \\ \frac{1}{\sqrt{n}}\mathbf{1}^\top \bar{C}U_1 & \frac{1}{n}\mathbf{1}^\top \bar{C}\mathbf{1} \end{pmatrix} D && \text{(G.8)}\\
&= \begin{pmatrix} B^\top & \\ \mathbf{0}^\top & 0 \end{pmatrix} \begin{pmatrix} U_1^\top \bar{C}U_1 & \frac{1}{\sqrt{n}}U_1^\top \bar{C}\mathbf{1} \\ \frac{1}{\sqrt{n}}\mathbf{1}^\top \bar{C}U_1 & \frac{1}{n}\mathbf{1}^\top \bar{C}\mathbf{1} \end{pmatrix} \begin{pmatrix} B & \mathbf{0} \\ & 0 \end{pmatrix} && \text{(G.9)}\\
&= \begin{pmatrix} U_1^\top \bar{C}U_1 & \mathbf{0} \\ \mathbf{0}^\top & 0 \end{pmatrix}. && \text{(G.10)}
\end{aligned}
$$

Using Lemma G.1 and taking $R = U^\top \bar{C}U$ and $S = U_1^\top \bar{C}U_1$, we have the eigenvalues of $U_1^\top \bar{C}U_1$ are interlacing between the eigenvalues of $U^\top \bar{C}U$. Note that the eigenvalues of $DU^\top \bar{C}UD$ are $\mu_1^2 \le \mu_2^2 \le \cdots \le \mu_{n-1}^2$ and $\mu_n^2 = 0$, and by Eq. (G.10), the eigenvalues of $DU^\top \bar{C}UD$ contain the eigenvalues of $U_1^\top \bar{C}U_1$ and 0. Since the eigenvalues of $U^\top \bar{C}U$ are $\lambda_1^2 \le \lambda_2^2 \le \cdots \le \lambda_n^2$ (By similarity of $U^\top \bar{C}U$ and $\bar{C}$), we then have

$$
\lambda_1^2 \le \mu_1^2 \le \lambda_2^2 \le \cdots \le \lambda_{n-1}^2 \le \mu_{n-1}^2 \le \lambda_n^2. \tag{G.11}
$$

Moreover, the equality holds only when there is a nonzero $z \in \mathbb{R}^{n-1}$ that satisfies

$$
\begin{aligned}
U_1^\top \bar{C}U_1 z &= \mu z, && \text{(G.12)}\\
\mathbf{1}^\top \bar{C}U_1 z &= 0, && \text{(G.13)}
\end{aligned}
$$

where μ is one of μ_i^2s.

Since U_1 forms an orthogonal basis of the orthogonal complement space of $\mathbf{1}$ and Eq. (G.13) is equivalent to "$\bar{C}U_1 z$ lies in the orthogonal complement space", we have that there is a vector $y \in \mathbb{R}^{n-1}$ such that

$$\bar{C} U_1 z = U_1 y. \tag{G.14}$$

Substituting this into Eq. (G.12), we have

$$U_1^\top U_1 y = \mu z. \tag{G.15}$$

Since $U_1^\top U_1 = I_{n-1}$, the equation above is equivalent to

$$y = \mu z, \tag{G.16}$$

which means

$$\bar{C} U_1 z = U_1 y = \mu U_1 z, \tag{G.17}$$

i.e., $U_1 z$ is the eigenvector of $\bar{C}$ associated with μ. By noticing $U_1 z$ lies in the orthogonal complement space of $\mathbf{1}$ and the eigenvector of $\bar{C}$ is right singular vector of Q, we complete the proof. □

G.1.2 Concrete example of the acceleration

To get more intuition on how the preconditioning effect of the shift can accelerate the training of GNNs, we provide a concrete example showing that shift indeed improves the convergence rate. To make things clear without loss of intuition, we focus on a simple linear GNN applied to a *well-specified task* where we are able to explicitly compare the convergence rates.

Settings

Data. We describe each sample, i.e., graph, with n nodes by a tuple $G = \{X, Q, \mathbf{p}, y\}$, where

- $X \in \mathbb{R}^{d\times n}$ is the feature matrix of the graph, where d is the dimension of the of each feature vector.

- $Q \in \mathbb{R}^{n\times n}$ representing the matrix representing the neighbor aggregation as Eq. (10.3). Note that this matrix depends on the aggregation scheme used by the chosen architecture, but for simplicity, we model this as a part of data structure.

- $\mathbf{p} \in \mathbb{R}^{n\times 1}$ is a weight vector representing the importance of each node. This will be used to calculate the $\mathrm{READOUT}$ step. Note that this vector is not provided in many real-world datasets, so the $\mathrm{READOUT}$ step usually takes operations such as summation.

- $y \in \mathbb{R}$ is the label.

The whole dataset $S = \{G_1, \cdots, G_m\}$ consists of m graphs where $G_i = \{X_i, Q_i, \mathbf{p}_i, y_i\}$. We make the following assumptions on the data generation process:

Assumption 1 (Independency)**.** *We assume X_i, Q_i, $\mathbf{p}_i$ are drawn from three independent distributions in an i.i.d. manner, e.g., $X_1, \cdots, X_m$ are i.i.d..*

Assumption 2 (Structure of data distributions)**.** *For clearness and simplicity of statement, we assume the number of nodes in each graph G_i are the same, we will use n to denote this number and we further assume $n = d$. We assume that the distribution of $\mathbf{p}_i$ satisfies $\mathbb{E}\left[\mathbf{p}\mathbf{p}^\top\right] = I_n, \mathbb{E}\mathbf{p} = 0$, which means the importance vector is non-degenerate. Let $\mathbb{E}XQ = Y$, we assume that Y is* full ranl. *We make the following assumptions on XQ: $\mathbf{1}^\top Y^{-1}XQ = 0$, which ensures that there is no information in the direction $\mathbf{1}^\top Y^{-1}$; there is a constant δ_1 such that $\mathbb{E}(XQ-Y)(XQ-Y)^\top \preceq \delta_1 I_d$ and $\mathbb{E}(XQ-Y)N(XQ-Y)^\top \preceq \delta_1 I_d$, where δ_1 characterizes the noise level; none of the eigenvectors of $YY^\top$ is orthogonal to* $\mathbf{1}$.

Remark G.1. A few remarks are in order, firstly, the assumption that each graph has the same number of nodes and the number n is equal to feature dimension d can be achieved by "padding", i.e., adding dummy points or features to the graph or the feature matrix. The assumption that $\mathbf{1}^\top Y^{-1}XQ = 0$ is used to guarantee that there is no information loss caused by shift ($\mathbf{1}^\top Y^{-1}YNY^\top = 0$). Though we make this strong assumption to ensure no information loss in theoretical part, we introduce "learnable shift" to mitigate this problem in the practical setting. The theory taking learnable shift into account is an interesting future direction.

Assumption 3 (Boundness)**.** *We make the technical assumption that there is a constant b such that the distributions of $X_i, Q_i, \mathbf{p}_i$ ensures*

$$\|X_i\| \, \|Q_i\| \, \|\mathbf{p}_i\| \leq \sqrt{b}. \tag{G.18}$$

Model. We consider a simple *linear graph neural network* with parameter $\mathbf{w} \in \mathbb{R}^{d\times 1}$:

$$f_{\mathbf{w}}^{\text{Vanilla}}(X, Q, \mathbf{p}) = \mathbf{w}^\top XQ\mathbf{p}. \tag{G.19}$$

Then, the model with shift can be represented as:

$$f_{\mathbf{w}}^{\text{Shift}}(X, Q, \mathbf{p}) = \mathbf{w}^\top XQN\mathbf{p}, \tag{G.20}$$

where $N = I_n - \frac{1}{n}\mathbf{1}\mathbf{1}^\top$.

Criterion. We consider using square loss as training objective, i.e.,

$$L(f) = \sum_{i=1}^{m} \frac{1}{2} \left(f(X_i, Q_i, \mathbf{p}_i) - y_i\right)^2. \tag{G.21}$$

Algorithm. We consider using gradient descent to optimize the objective function. Let the initial parameter $\mathbf{w}_0 = 0$. The update rule of w from step t to $t+1$ can be described as:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla_{\mathbf{w}} L(f_{\mathbf{w}_t}), \tag{G.22}$$

where η is the learning rate.

Theorem G.2. *Under Assumption 1,2,3, for any $\epsilon > 0$ there exists constants C_1, C_2, such that for $\delta_1 < C_1, m > C_2$, with probability $1 - \epsilon$, the parameter $\mathbf{w}_t^{\text{Vanilla}}$ of vanilla model* *converges to the optimal parameter* $\mathbf{w}_*^{\text{Vanilla}}$ *linearly:*

$$\left\|\mathbf{w}_t^{\text{Vanilla}} - \mathbf{w}_*^{\text{Vanilla}}\right\|_2 \leq O\left(\rho_1^t\right), \tag{G.23}$$

while the parameter $\mathbf{w}_t^{\text{Shfit}}$ *of the shifted model converges to the optimal parameter* $\mathbf{w}_*^{\text{Shfit}}$ *linearly:*

$$\left\|\mathbf{w}_t^{\text{Shift}} - \mathbf{w}_*^{\text{Shfit}}\right\|_2 \leq O\left(\rho_2^t\right), \tag{G.24}$$

where

$$1 > \rho_1 > \rho_2, \tag{G.25}$$

which indicates the shifted model has a faster convergence rate.

Proof. We firstly reformulate the optimization problem in matrix form.

Notice that in our linear model, the representation and structure of a graph $G_i = \{X_i, Q_i, \mathbf{p}_i, y_i\}$ can be encoded as a whole in a single vector, i.e., $\mathbf{z}_i^{\text{Vanilla}} = X_i Q_i \mathbf{p}_i \in \mathbb{R}^{d\times 1}$ for vanilla model in Eq. (G.19), and $\mathbf{z}_i^{\text{Shift}} = X_i Q_i N \mathbf{p}_i \in \mathbb{R}^{d\times 1}$ for shifted model in Eq. (G.20). We call $\mathbf{z}_i$ and $\mathbf{z}_i^{\text{Shift}}$ "combined features". Let $Z^{\text{Vanilla}} = \left[\mathbf{z}_1^{\text{Vanilla}}, \cdots, \mathbf{z}_m^{\text{Vanilla}}\right] \in \mathbb{R}^{d\times m}$ and $Z^{\text{Shift}} = \left[\mathbf{z}_1^{\text{Shift}}, \cdots, \mathbf{z}_m^{\text{Shift}}\right] \in \mathbb{R}^{d\times m}$ be the matrix of combined features of valinna linear model and shifted linear model respectively. For clearness of the proof, we may abuse the notations and use Z to represent Z^{Vanilla}. Then the objective in Eq. (G.21) for vanilla linear model can be reformulated as:

$$L(f_{\mathbf{w}}) = \frac{1}{2}\left\|Z^\top \mathbf{w} - \mathbf{y}\right\|_2^2, \tag{G.26}$$

where $\mathbf{y} = [y_1, \cdots, y_m]^\top \in \mathbb{R}^{m\times 1}$.

Then the gradient descent update can be explicitly writen as:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta\left(ZZ^\top \mathbf{w}_t - Z\mathbf{y}\right) \tag{G.27}$$

$$= (I_d - \eta ZZ^\top)\mathbf{w}_t + \eta Z\mathbf{y}, \tag{G.28}$$

which converges to $\mathbf{w}_* = \left(ZZ^\top\right)^\dagger Z\mathbf{y}$ according to classic theory of least square problem [Horn and Johnson, 2012], where $\left(ZZ^\top\right)^\dagger$ is the Moore–Penrose inverse of $ZZ^\top$.

By simultaneously subtracting $\mathbf{w}_*$ in the update rule, we have

$$\mathbf{w}_{t+1} - \mathbf{w}_* = \left(I_d - \eta ZZ^\top\right)\left(\mathbf{w}_t - \mathbf{w}_*\right). \tag{G.29}$$

So the residual of $\mathbf{w}_t$ is

$$\begin{aligned}\|\mathbf{w}_t - \mathbf{w}_*\| &= \left\|\left(I_d - \eta ZZ^\top\right)^t \mathbf{w}_*\right\| && \text{(G.30)}\\ &\leq \left\|I_d - \eta ZZ^\top\right\|^t \|\mathbf{w}_*\|. && \text{(G.31)}\end{aligned}$$

Let $\sigma_{\max}(A)$ and $\sigma_{\min}(A)$ be the maximal and mininal *positive* eigenvalues of A, respectively. Then the optimial learning rate (the largest learning rate that ensures $I_d - \eta ZZ^\top$ is positive semidefinite) is $\eta = \frac{1}{\sigma_{\max}(ZZ^\top)}$. Under this learning rate we have the convergence rate following Eq. (G.31):

$$\begin{aligned}\|\mathbf{w}_t - \mathbf{w}_*\| &\leq \left\|I_d - \eta ZZ^\top\right\|^t \|\mathbf{w}_*\| && \text{(G.32)}\\ &\leq \left(1 - \frac{\sigma_{\min}\left(ZZ^\top\right)}{\sigma_{\max}\left(ZZ^\top\right)}\right)^t \|\mathbf{w}_*\|. && \text{(G.33)}\end{aligned}$$

For now, we show that the convergence rate of the optimization problem with vanilla model depends on $\frac{\sigma_{\min}\left(ZZ^\top\right)}{\sigma_{\max}\left(ZZ^\top\right)}$. Follwing the same argument, we can show the convergence rate of the optimization problem with shifted model depends on $\frac{\sigma_{\min}\left(Z^{\text{Shift}}Z^{\text{Shfit}\top}\right)}{\sigma_{\max}\left(Z^{\text{Shift}}Z^{\text{Shfit}\top}\right)}$. We then aim to bound this term, which we call effective condition number.

Similarly, we investigate the effective condition number for $ZZ^\top$ first, and the analysis of $Z^{\text{Shift}}Z^{\text{Shift}\top}$ follows the same manner. As multiplying a constant does not affect the effective condition number, we first scale $ZZ^\top$ by $\frac{1}{m}$ and expand it as:

$$\frac{1}{m}ZZ^\top = \frac{1}{m}\sum_{i=1}^{m}\mathbf{z}_i\mathbf{z}_i^\top, \tag{G.34}$$

which is the empirical estimation of the covariance matrix of the combined feature. By concentration inequality, we know this quantity is concentrated to the covariance matrix, i.e.,

$$\begin{aligned}\mathbb{E}_{\mathbf{z}}\mathbf{z}\mathbf{z}^\top &= \mathbb{E}_{X,Q,\mathbf{p}} XQ\mathbf{p}\left(XQ\mathbf{p}\right)^\top \\ &= \mathbb{E}_{X,Q} XQ\left(\mathbb{E}\left[\mathbf{p}\mathbf{p}^\top\right]\right)(XQ)^\top \\ &= \mathbb{E}_{X,Q} XQ(XQ)^\top \quad \text{(By Assumption 1)} \\ &= YY^\top + \mathbb{E}_{X,Q}(XQ-Y)(XQ-Y)^\top.\end{aligned}$$

Noticing that $\mathbf{0} \preceq \mathbb{E}_{X,Q}(XQ-Y)(XQ-Y)^\top \preceq \delta_1 I_d$ by Assumption 2, and Y is full rank, we can conclude that $\sigma_{\max}\left(YY^\top\right) \le \sigma_{\max}\left(\mathbb{E}_{\mathbf{z}}\mathbf{z}\mathbf{z}^\top\right) \le \sigma_{\max}\left(YY^\top\right) + \delta_1$, and $\sigma_{\min}\left(YY^\top\right) \le \sigma_{\min}\left(\mathbb{E}_{\mathbf{z}}\mathbf{z}\mathbf{z}^\top\right) \le \sigma_{\min}\left(YY^\top\right) + \delta_1$ by Weyl's inequality.

By similar argument, we have that $\frac{1}{m}Z^{\text{Shift}}Z^{\text{Shift}\top}$ concentrates to

$$\begin{aligned}&\mathbb{E}_{\mathbf{z}^{\text{Shift}}}\mathbf{z}^{\text{Shift}}\mathbf{z}^{\text{Shift}\top} \\ =&\mathbb{E}_{X,Q}(XQ)N^2(XQ)^\top \\ =&\mathbb{E}_{X,Q}(XQ)N(XQ)^\top \quad (N^2 = N) \\ =&YNY^\top + \mathbb{E}_{X,Q}(XQ-Y)N(XQ-Y)^\top.\end{aligned}$$

By Assumption 2, we have

$$\begin{aligned}0 =&\mathbf{1}^\top Y^{-1}\mathbb{E}_{\mathbf{z}^{\text{Shift}}}\mathbf{z}^{\text{Shift}}\mathbf{z}^{\text{Shift}\top} \\ =&\mathbf{1}^\top Y^{-1}\left(YNY^\top + \mathbb{E}_{X,Q}(XQ-Y)N(XQ-Y)^\top\right) \\ =&\mathbf{1}^\top Y^{-1}\mathbb{E}_{X,Q}(XQ-Y)N(XQ-Y)^\top,\end{aligned}$$

which means $\mathbb{E}_{X,Q}(XQ-Y)N(XQ-Y)^\top$ has the same eigenspace as $YNY^\top$ with respect to eigenvalue 0. Combining with $\mathbf{0} \preceq \mathbb{E}_{X,Q}(XQ-Y)N(XQ-Y)^\top \preceq \delta_1 I_d$, we have $\sigma_{\max}\left(YNY^\top\right) \le \sigma_{\max}\left(\mathbb{E}_{\mathbf{z}^{\text{Shift}}}\mathbf{z}^{\text{Shift}}\mathbf{z}^{\text{Shift}\top}\right) \le \sigma_{\max}\left(YNY^\top\right) + \delta_1$, and $\sigma_{\min}\left(YNY^\top\right) \le \sigma_{\min}\left(\mathbb{E}_{\mathbf{z}^{\text{Shift}}}\mathbf{z}^{\text{Shift}}\mathbf{z}^{\text{Shift}\top}\right) \le \sigma_{\min}\left(YNY^\top\right) + \delta_1$.

It remains to bound the finite sample error, i.e., $\left\|\frac{1}{m}ZZ^\top - \mathbb{E}_{\mathbf{z}}\mathbf{z}\mathbf{z}^\top\right\|_2$ and $\left\|\frac{1}{m}Z^{\text{Shift}}Z^{\text{Shfit}\top} - \mathbb{E}_{\mathbf{z}}\mathbf{z}\mathbf{z}^\top\right\|_2$. These bounds can be obtained by the following lemma:

Lemma G.3 (Corollary 6.1 in Wainwright [2019])**.** *Let $\mathbf{z}_1, \cdots, \mathbf{z}_m$ be i.i.d. zero-mean random vectors with covariance matrix Σ such that $\|\mathbf{z}\|_2 \leq \sqrt{b}$ almost surely. Then for all $\delta > 0$, the sample covariance matrix $\hat{\Sigma} = \frac{1}{m}\sum_{i=1}^{m} \mathbf{z}_i \mathbf{z}_i^\top$ satisfies*

$$\Pr\left[\left\|\hat{\Sigma} - \Sigma\right\|_2 \geq \delta\right] \leq 2d \exp\left(-\frac{\delta^2}{2b\left(\|\Sigma\|_2 + \delta\right)}\right). \tag{G.35}$$

By this lemma, we further have

Lemma G.4 (Bound on the sample covariance matrix)**.** *Let $\mathbf{z}_1, \cdots, \mathbf{z}_m$ be i.i.d. zero-mean random vectors with covariance matrix Σ such that $\|\mathbf{z}\|_2 \leq \sqrt{b}$ almost surely. Then with probability $1 - \epsilon$, the sample covariance matrix $\hat{\Sigma} = \frac{1}{m}\sum_{i=1}^{m} \mathbf{z}_i \mathbf{z}_i^\top$ satisfies*

$$\left\|\hat{\Sigma} - \Sigma\right\|_2 \leq O\left(\sqrt{\frac{\log(1/\epsilon)}{m}}\right), \tag{G.36}$$

where we hide constants $b, \|\Sigma\|_2, d$ in the big-O notation and highlight the dependence on the number of samples m.

Combining with previous results, we conclude that:

$$\begin{aligned}
&\sigma_{\max}\left(YY^\top\right) - O\left(\sqrt{\frac{\log(1/\epsilon)}{m}}\right)\\
\le&\sigma_{\max}\left(\frac{1}{m}ZZ^\top\right)\\
\le&\sigma_{\max}\left(YY^\top\right) + \delta_1 + O\left(\sqrt{\frac{\log(1/\epsilon)}{m}}\right);\\
&\sigma_{\min}\left(YY^\top\right) - O\left(\sqrt{\frac{\log(1/\epsilon)}{m}}\right)\\
\le&\sigma_{\min}\left(\frac{1}{m}ZZ^\top\right)\\
\le&\sigma_{\min}\left(YY^\top\right) + \delta_1 + O\left(\sqrt{\frac{\log(1/\epsilon)}{m}}\right);\\
&\sigma_{\max}\left(YNY^\top\right) - O\left(\sqrt{\frac{\log(1/\epsilon)}{m}}\right)\\
\le&\sigma_{\max}\left(\frac{1}{m}Z^{\text{Shift}}Z^{\text{Shift}\top}\right)\\
\le&\sigma_{\max}\left(YNY^\top\right) + \delta_1 + O\left(\sqrt{\frac{\log(1/\epsilon)}{m}}\right)
\end{aligned}$$

$$\begin{aligned}
&\sigma_{\min}\left(YNY^\top\right) - O\left(\sqrt{\frac{\log(1/\epsilon)}{m}}\right)\\
\le&\sigma_{\min}\left(\frac{1}{m}Z^{\text{Shift}}Z^{\text{Shift}\top}\right)\\
\le&\sigma_{\min}\left(YNY^\top\right) + \delta_1 + O\left(\sqrt{\frac{\log(1/\epsilon)}{m}}\right).
\end{aligned}$$

By now, we have transfered the analysis of $ZZ^\top$ and $Z^{\text{Shift}}Z^{\text{Shfit}\top}$ to the analysis of $YY^\top$ and $YNY^\top$. And the positive eigenvalues of $YNY^\top$ is interlaced between the positive eigenvalues of $YY^\top$ by the same argument as Theorem 10.1. Concretely, we have $\sigma_{\min}\left(YY^\top\right) \le \sigma_{\min}\left(YNY^\top\right) \le \sigma_{\max}\left(YNY^\top\right) \le \sigma_{\max}\left(YY^\top\right)$. Noticing that none of the eigenvectors of $YY^\top$ is orthogonal to $\mathbf{1}$, the first and last equalies can not be achieved, so $\sigma_{\min}\left(YY^\top\right) < \sigma_{\min}\left(YNY^\top\right) \le \sigma_{\max}\left(YNY^\top\right) < \sigma_{\max}\left(YY^\top\right)$. Finally, we can conclude for small enough δ_1 and large enough m, with probability ϵ,

$$\begin{aligned}
&\sigma_{\min}\left(\frac{1}{m}ZZ^\top\right)\\
\leq&\sigma_{\min}\left(YY^\top\right)+\delta_1+O\left(\sqrt{\frac{\log(1/\epsilon)}{m}}\right)\\
<&\sigma_{\min}\left(YNY^\top\right)-O\left(\sqrt{\frac{\log(1/\epsilon)}{m}}\right)\\
\leq&\sigma_{\min}\left(\frac{1}{m}Z^{\text{Shift}}Z^{\text{Shift}\top}\right)
\end{aligned}$$

$$\begin{aligned}
\leq&\sigma_{\max}\left(\frac{1}{m}Z^{\text{Shift}}Z^{\text{Shift}\top}\right)\\
\leq&\sigma_{\max}\left(YNY^\top\right)+\delta_1+O\left(\sqrt{\frac{\log(1/\epsilon)}{m}}\right)\\
<&\sigma_{\max}\left(YY^\top\right)-O\left(\sqrt{\frac{\log(1/\epsilon)}{m}}\right)\\
\leq&\sigma_{\max}\left(\frac{1}{m}ZZ^\top\right).
\end{aligned}$$

So

$$\begin{aligned}
\rho_2=&1-\frac{\sigma_{\min}\left(Z^{\text{Shift}}Z^{\text{Shift}\top}\right)}{\sigma_{\max}\left(Z^{\text{Shift}}Z^{\text{Shift}\top}\right)}\\
<\rho_1=&1-\frac{\sigma_{\min}\left(ZZ^\top\right)}{\sigma_{\max}\left(ZZ^\top\right)},
\end{aligned}$$

where ρ_1, ρ_2 are the constants in the statement of the theorem. This inequality means the shifted model has better convergence speed by Eq. (G.33). $\square$

G.1.3 Proof of Proposition 10.3

Proof. For r-regular graph, $A = r \cdot I_n$ and $Q_{\text{GIN}} = \left(r+1+\xi^{(1)}\right) I_n$. Since $H^{(0)}$ is given by one-hot encodings of node degrees, the row of $H^{(0)}$ can be represented as $c \cdot \mathbf{1}^\top$ where $c = 1$ for the r-th row and $c = 0$ for other rows. By the associative property of matrix multiplication, we only need to show $H^{(0)}Q_{\text{GIN}}N = 0$. This is because, for each row

$$\begin{aligned} c \cdot \mathbf{1}^\top Q_{\text{GIN}}N &= c \cdot \mathbf{1}^\top (r + 1 + \xi^{(1)}) I_n \left(I_n - \frac{1}{n}\mathbf{1}\mathbf{1}^\top \right) & \text{(G.37)} \\ &= c\left(r + 1 + \xi^{(1)}\right)\left(\mathbf{1}^\top - \mathbf{1}^\top \cdot \frac{1}{n}\mathbf{1}\mathbf{1}^\top\right) = 0. & \text{(G.38)} \end{aligned}$$

□

G.1.4 Proof of Proposition 10.4

Proof.

$$Q_{\text{GIN}}N = (A + I_n + \xi^{(k)} I_n)N == (\mathbf{1}\mathbf{1}^\top + \xi^{(k)I_n})N = \xi^{(k)}N, \tag{G.39}$$

□

G.1.5 Gradient of $W^{(k)}$

We first calculate the gradient of $W^{(k)}$ when using normalization. Denote $Z^{(k)} = \text{Norm}\left(W^{(k)}H^{(k-1)}Q\right)$ and $\mathcal{L}$ as the loss. Then the gradient of $\mathcal{L}$ w.r.t. the weight matrix $W^{(k)}$ is

$$\frac{\partial \mathcal{L}}{\partial W^{(k)}} = \left(\left(H^{(k-1)}QN\right)^\top \otimes S\right)\frac{\partial \mathcal{L}}{\partial Z^{(k)}}, \tag{G.40}$$

where $\otimes$ represents the Kronecker product, and thus $\left(H^{(k-1)}QN\right)^\top \otimes S$ is an operator on matrices.

Analogously, the gradient of $W^{(k)}$ without normalization consists a $\left(H^{(k-1)}Q\right)^\top \otimes I_n$ term. As suggested by Theorem 10.1, QN has a smoother distribution of spectrum than Q, so that the gradient of $W^{(k)}$ with normalization enjoys better optimization curvature than that without normalizaiton.

G.2 Experimental Setup

Network architecture. For the medium-scale bioinformatics and social network datasets, we use 5-layer GIN/GCN with a linear output head for prediction followed Xu et al. [2019] with residual connection. The hidden dimension of GIN/GCN is set to be 64. For the large-scale ogbg-molhiv dataset, we also use 5-layer GIN/GCN[Xu et al., 2019] architecture with residual connection. Following Hu et al. [2020a], we set the hidden dimension as 300.

Baselines. For the medium-scale bioinformatics and social network datasets, we compare several competitive baselines as in Xu et al. [2019], including the WL subtree kernel model [Shervashidze et al., 2011], diffusion-convolutional neural networks (DCNN) [Atwood and Towsley, 2016], Deep Graph CNN (DGCNN) [Zhang et al., 2018a] and Anonymous Walk Embeddings (AWL) [Ivanov and Burnaev, 2018]. We report the accuracies reported in the original paper [Xu et al., 2019]. For the large-scale ogbg-molhiv dataset, we use the baselines in Hu et al. [2020a], including the Graph-agnostic MLP model, GCN [Kipf and Welling, 2017] and GIN [Xu et al., 2019]. We also report the roc-auc values reported in the original paper [Hu et al., 2020a].

Hyper-parameter configurations. We use Adam [Kingma and Ba, 2015] optimizer with a linear learning rate decay schedule. We follow previous work Xu et al. [2019] and Hu et al. [2020a] to use hyper-parameter search (grid search) to select the best hyper-parameter based on validation performance. In particular, we select the batch size $\in \{64, 128\}$, the dropout ratio $\in \{0, 0.5\}$, weight decay $\in \{5e-2, 5e-3, 5e-4, 5e-5\} \cup \{0.0\}$, the learning rate $\in \{1e-4, 1e-3, 1e-2\}$. For the drawing of the training curves in Figure 10-2, for simplicity, we set batch size to be 128, dropout ratio to be 0.5, weight decay to be 0.0, learning rate to be 1e-2, and train the models for 400 epochs for all settings.

Evaluation. Using the chosen hyper-parameter, we report the averaged test performance over different random seeds (or cross-validation). In detail, for the medium-scale datasets, following Xu et al. [2019], we perform a 10-fold cross-validation as these datasets do not have a clear train-validate-test splitting format. The mean and standard deviation of the validation accuracies across the 10 folds are reported. For the ogbg-molhiv dataset, we follow the official setting [Hu et al., 2020a]. We repeat the training process with 10 different random seeds.

For all experiments, we select the best model checkpoint with the best validation accuracy and record the corresponding test performance.

G.3 Additional Experimental Results

G.3.1 Visualization of the singular value distributions

As stated in Theorem 10.1, the shift operation N serves as a preconditioner of Q which makes the singular value distribution of Q smoother. To check the improvements, we sample graphs from 6 median-scale datasets (PROTEINS, NCI1, MUTAG, PTC, IMDB-BINARY, COLLAB) for visualization, as in Figure G-2.

G.3.2 Visualization of noise in the batch statistics

We show the noise of the batch statistics on the PROTEINS task in the main body. Here we provide more experiment details and results.

For graph tasks (PROTEINS, PTC, NCI1, MUTAG, IMDB-BINARY datasets), we train a 5-layer GIN with BatchNorm as in Xu et al. [2019] and the number of sub-layers in MLP is set to 2. For image task (CIFAR10 dataset), we train a ResNet18 [He et al., 2016]. Note that for a 5-layer GIN model, it has four graph convolution layers (indexed from 0 to 3) and each graph convolution layer has two BatchNorm layers; for a ResNet18 model, except for the first 3$\times$3 convolution layer and the final linear prediction layer, it has four basic layers (indexed from 0 to 3) and each layer consists of two basic blocks (each block has two BatchNorm layers). For image task, we set the batch size as 128, epoch as 100, learning rate as 0.1 with momentum 0.9 and weight decay as 5e-4. For graph tasks, we follow the setting of Figure 10-2 (described in Appendix G.2).

The visualization of the noise in the batch statistics is obtained as follows. We first train the models and dump the model checkpoints at the end of each epoch; Then we randomly sample one feature dimension and fix it. For each model checkpoint, we feed different batches to the model and record the maximum/minimum batch-level statistics (mean and standard deviation) of the feature dimension across different batches. We also calculate dataset-level statistics.

As Figure 10-4 in the main body, pink line denotes the dataset-level statistics, and green/blue line denotes the maximum/minimum value of the batch-level statistics respectively. First, we provide more results on PTC, NCI1, MUTAG, IMDB-BINARY tasks, as in Figure G-3. We visualize the statistics from the first (layer-0) and the last (layer-3) BatchNorm layers in GIN for comparison. Second, we further visualize the statistics from different BatchNorm layers (layer 0 to layer 3) in GIN on PROTEINS and ResNet18 in CIFAR10, as in Figure G-4. Third, we conduct experiments to investigate the influence of the batch size. We visualize the statistics from BatchNorm layers under different settings of batch sizes [8, 16, 32, 64], as in Figure G-5. We can see that the observations are consistent and the batch statistics on graph data are noisy, as in Figure 10-4 in the main body.

G.3.3 Training Curves on GCN

As shown in the main body, we train GCNs with different normalization methods (GraphNorm, InstanceNorm, BatchNorm and LayerNorm) and GCN without normalization in graph classification tasks and plot the training curves in Figure 6. It is obvious that the GraphNorm also enjoys the fastest convergence on all tasks. Remarkably, GCN with InstanceNorm even underperforms GCNs with other normalizations, while our GraphNorm with learnable shift significantly boosts the training upon InstanceNorm and achieves the fastest convergence.

G.3.4 Further Results of Ablation Study

BatchNorm with learnable shift. We conduct experiments on BatchNorm to investigate whether simply introducing a learnable shift can already improve the existing normalization methods without concrete motivation of overcoming expressiveness degradation. Specifically, we equip BatchNorm with a similar learnable shift (α-BatchNorm for short) as

GraphNorm and evaluate its performance. As shown in Figure below, the α-BatchNorm cannot outperform the BatchNorm on the three datasets. Moreover, as shown in Figure 5 in the main body, the learnable shift significantly improve upon GraphNorm on IMDB-BINARY dataset, while it cannot further improve upon BatchNorm, which suggests the introduction of learnable shift in GraphNorm is critical.

BatchNorm with running statistics. We study the variant of BatchNorm which uses running statistics (MS-BatchNorm for short) to replace the batch-level mean and standard deviation. At first glance, this method may seem to be able to mitigate the problem of large batch noise. However, the running statistics change a lot during training, and using running statistics disables the model to back-propagate the gradients through mean and standard deviation. Thus, we also train GIN with BatchNorm which stops the back-propagation of the graidients through mean and standard deviation (DT-BatchNorm for short). Both the MS-BatchNorm and DT-BatchNorm underperform the BatchNorm by a large margin, which shows that the problem of the heavy batch noise cannot be mitigated by simply using the running statistics.

The effect of batch size. We further compare the GraphNorm and BatchNorm with different batch sizes (8, 16, 32, 64). As shown in Figure below, our GraphNorm consistently outperforms the BatchNorm on all the settings.

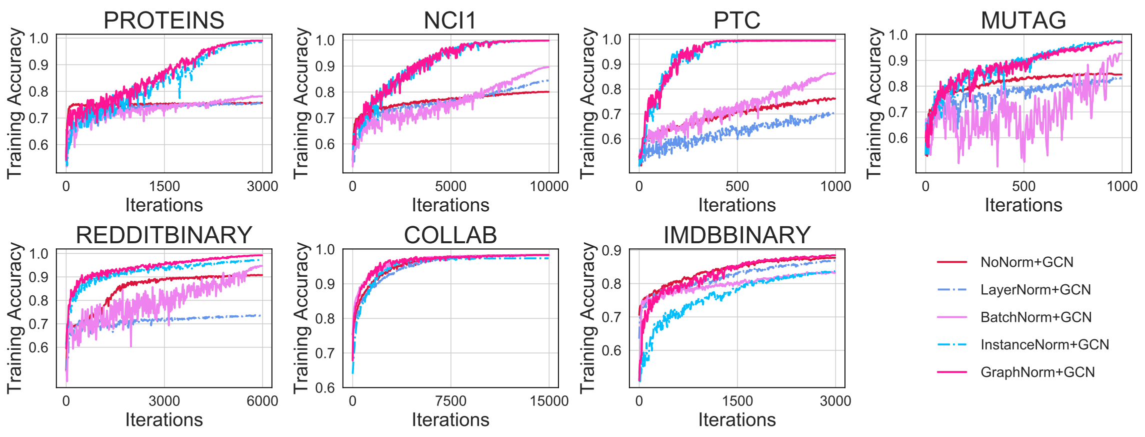


Figure G-1: **Training performance** of GCN with different normalization methods and GCN without normalization in graph classification tasks.

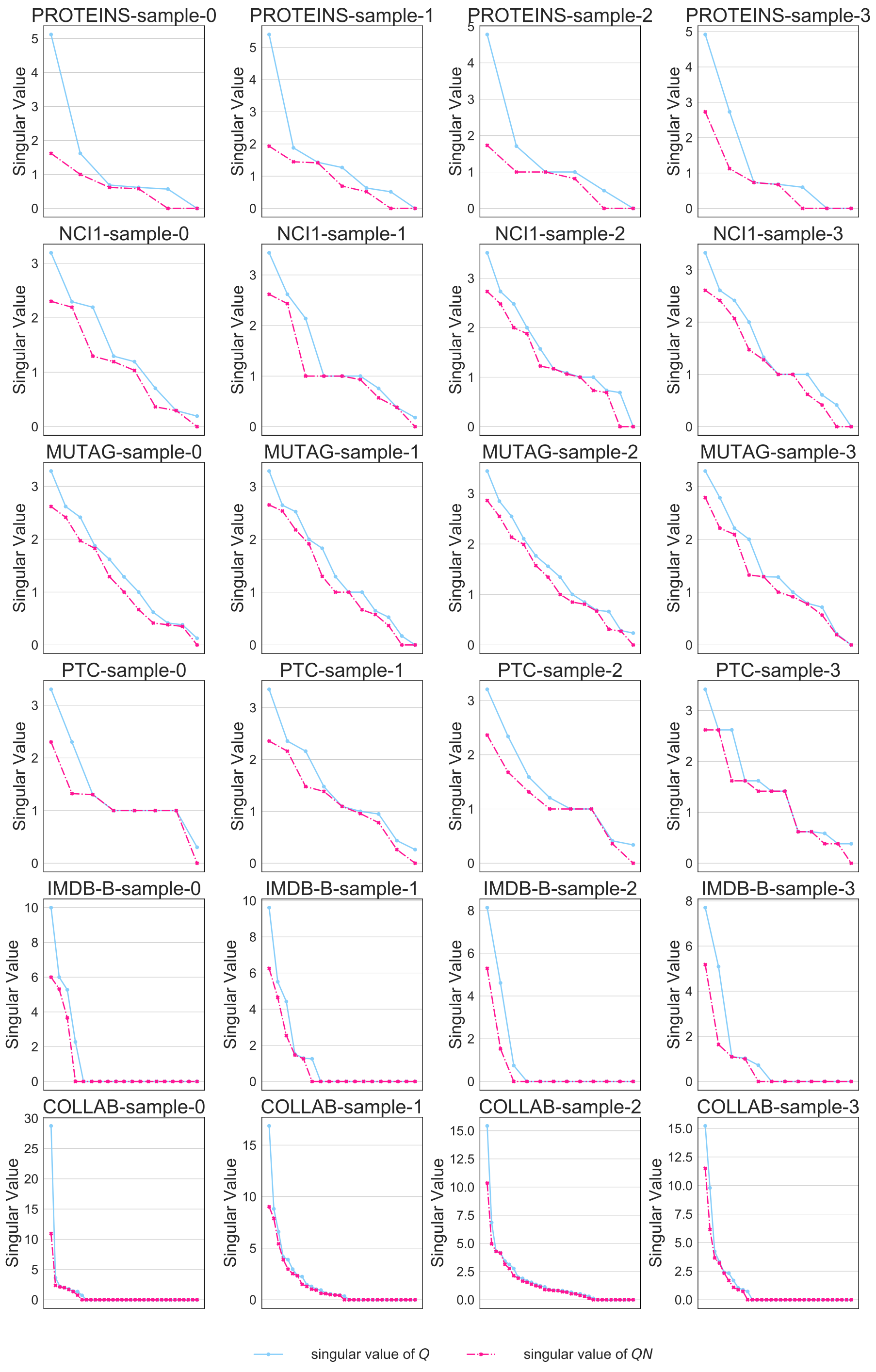


Figure G-2: **Singular value distribution of Q and QN.** Graph samples from PROTEINS, NCI1, MUTAG, PTC, IMDB-BINARY, COLLAB are presented.

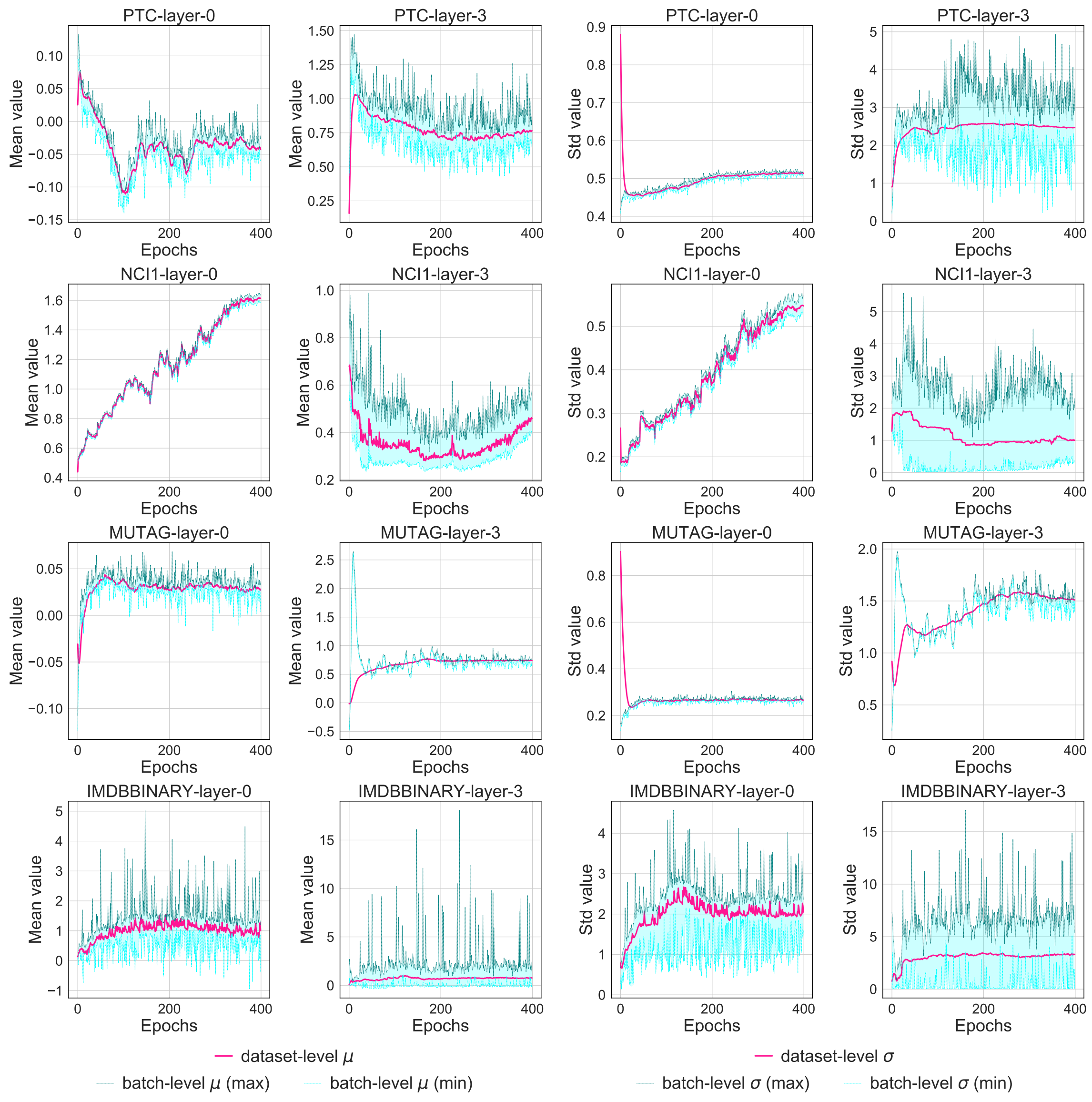


Figure G-3: **Batch-level statistics are noisy for GNNs** (Examples from PTC, NCI1, MUTAG, IMDB-BINARY datasets). We plot the batch-level mean/standard deviation and dataset-level mean/standard deviation of the first (layer 0) and the last (layer 3) BatchNorm layers in different checkpoints. GIN with 5 layers is employed.

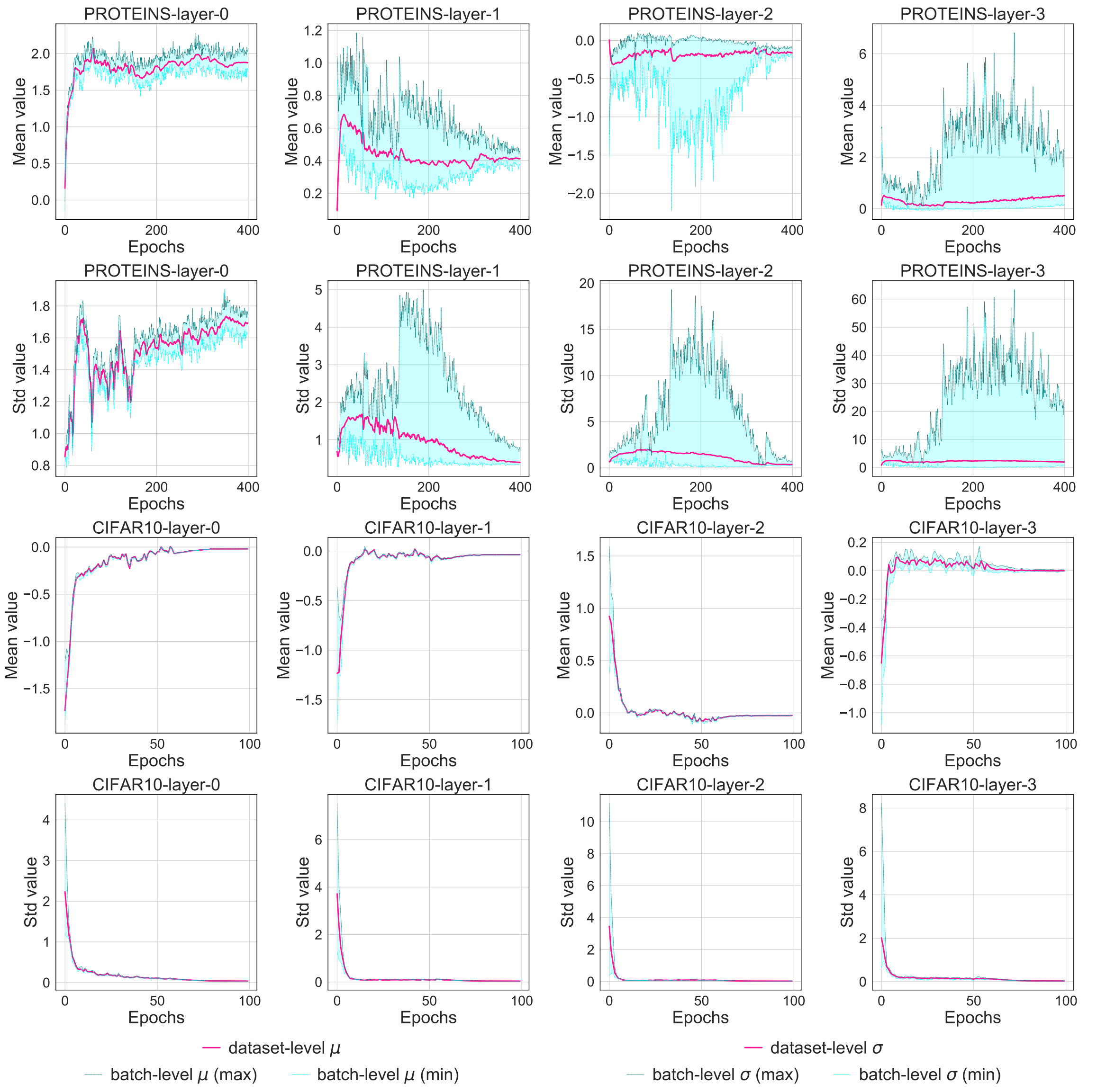


Figure G-4: **Batch-level statistics are noisy for GNNs of different depth.** We plot the batch-level mean/standard deviation and dataset-level mean/standard deviation of different BatchNorm layers (from layer 0 to layer 3) in different checkpoints. We use a five-layer GIN on PROTEINS and ResNet18 on CIFAR10 for comparison.

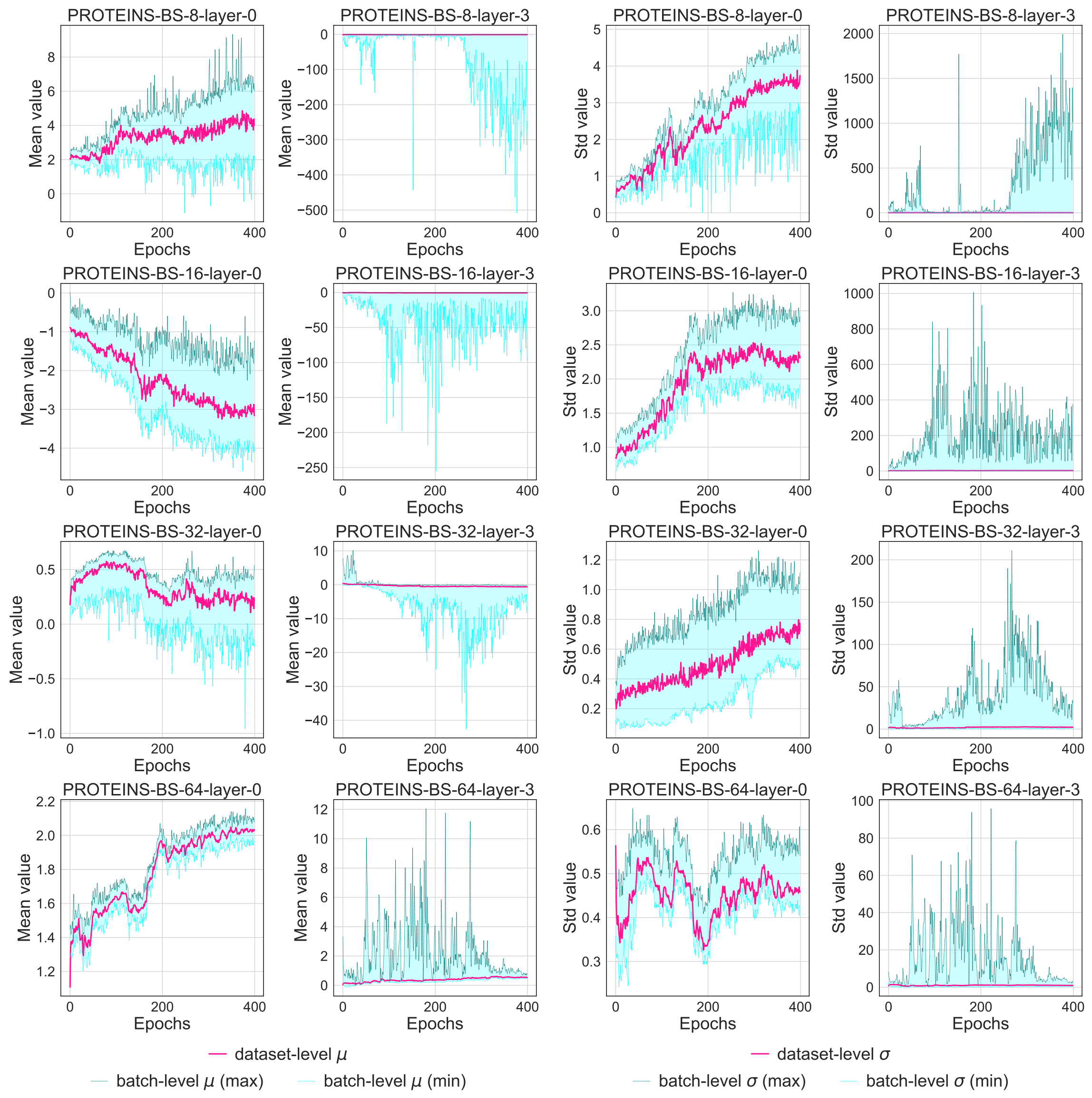


Figure G-5: **Batch-level statistics are noisy for GNNs of different batch sizes.** We plot the batch-level mean/standard deviation and dataset-level mean/standard deviation of different BatchNorm layers (layer 0 and layer 3) in different checkpoints. Specifically, different batch sizes (8, 16, 32, 64) are chosed for comparison. GIN with 5 layers is employed.

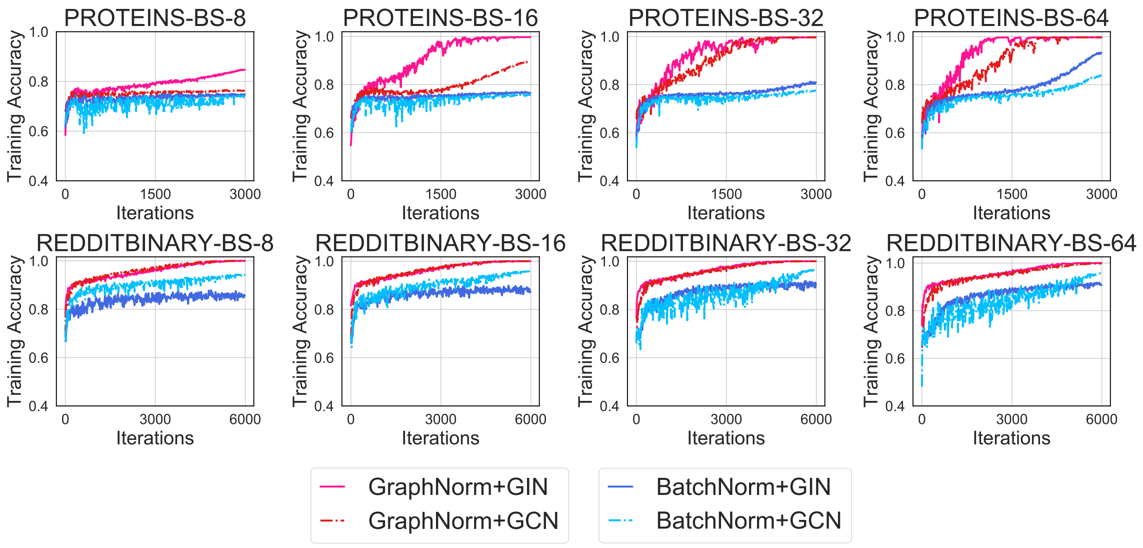


Figure G-6: **Training performance** of GIN/GCN with GraphNorm and BatchNorm with batch sizes of (8, 16, 32, 64) on PROTEINS and REDDITBINARY datasets.

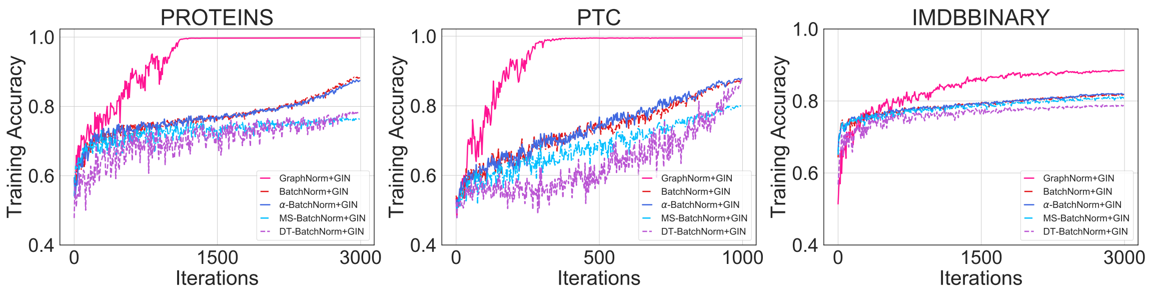


Figure G-7: **Training performance** of GIN with GraphNorm and variant BatchNorms (α-BatchNorm, MS-BatchNorm and DT-BatchNorm) on PROTEINS, PTC and IMDB-BINARY datasets.

Bibliography

Sami Abu-El-Haija, Amol Kapoor, Bryan Perozzi, and Joonseok Lee. N-gcn: Multi-scale graph convolution for semi-supervised node classification. In *Uncertainty in Artificial Intelligence*, pages 841–851. PMLR, 2020.

Kartik Ahuja, Jun Wang, Amit Dhurandhar, Karthikeyan Shanmugam, and Kush R. Varshney. Empirical or invariant risk minimization? a sample complexity perspective. In *International Conference on Learning Representations*, 2021.

Zeyuan Allen-Zhu, Yuanzhi Li, and Yingyu Liang. Learning and generalization in overparameterized neural networks, going beyond two layers. In *Advances in Neural Information Processing Systems*, pages 6155–6166, 2019a.

Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. In *International Conference on Machine Learning*, pages 242–252, 2019b.

Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C Lawrence Zitnick, and Devi Parikh. Vqa: Visual question answering. In *Proceedings of the IEEE international conference on computer vision*, pages 2425–2433, 2015.

Martin Arjovsky, Léon Bottou, Ishaan Gulrajani, and David Lopez-Paz. Invariant risk minimization. *arXiv preprint arXiv:1907.02893*, 2019.

Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in ak-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. Understanding deep neural networks with rectified linear units. In *International Conference on Learning Representations*, 2018a.

Sanjeev Arora, Nadav Cohen, and Elad Hazan. On the optimization of deep networks: Implicit acceleration by overparameterization. In *International Conference on Machine Learning*, pages 244–253. PMLR, 2018b.

Sanjeev Arora, Zhiyuan Li, and Kaifeng Lyu. Theoretical analysis of auto rate-tuning by batch normalization. *arXiv preprint arXiv:1812.03981*, 2018c.

Sanjeev Arora, Nadav Cohen, Noah Golowich, and Wei Hu. A convergence analysis of gradient descent for deep linear neural networks. In *International Conference on Learning Representations*, 2019a.

Sanjeev Arora, Simon Du, Wei Hu, Zhiyuan Li, and Ruosong Wang. Fine-grained analysis of optimization and generalization for overparameterized two-layer neural networks. In *International Conference on Machine Learning*, pages 322–332, 2019b.

Sanjeev Arora, Simon S Du, Wei Hu, Zhiyuan Li, Russ R Salakhutdinov, and Ruosong Wang. On exact computation with an infinitely wide neural net. In *Advances in Neural Information Processing Systems*, pages 8141–8150, 2019c.

Sanjeev Arora, Simon S. Du, Zhiyuan Li, Ruslan Salakhutdinov, Ruosong Wang, and Dingli Yu. Harnessing the power of infinitely wide deep nets on small-data tasks. In *International Conference on Learning Representations*, 2020.

James Atwood and Don Towsley. Diffusion-convolutional neural networks. pages 1993–2001, 2016.

Owe Axelsson. A survey of preconditioned iterative methods for linear systems of algebraic equations. *BIT Numerical Mathematics*, 25(1):165–187, 1985.

Jimmy Ba, Murat Erdogdu, Taiji Suzuki, Denny Wu, and Tianzong Zhang. Generalization of two-layer neural networks: An asymptotic viewpoint. In *International Conference on Learning Representations*, 2020.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

László Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 684–697. ACM, 2016.

László Babai and Ludik Kucera. Canonical labelling of graphs in linear average time. In *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pages 39–46. IEEE, 1979.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Rolf W Banz. The relationship between return and market value of common stocks. *Journal of financial economics*, 9(1):3–18, 1981.

Pablo Barceló, Egor V. Kostylev, Mikael Monet, Jorge Pérez, Juan Reutter, and Juan Pablo Silva. The logical expressiveness of graph neural networks. In *International Conference on Learning Representations*, 2020.

Etienne Barnard and LFA Wessels. Extrapolation and interpolation in neural network classifiers. *IEEE Control Systems Magazine*, 12(5):50–53, 1992.

Peter L Bartlett and Shahar Mendelson. Rademacher and gaussian complexities: Risk bounds and structural results. *Journal of Machine Learning Research*, 3(Nov):463–482, 2002.

Peter L Bartlett, Dylan J Foster, and Matus J Telgarsky. Spectrally-normalized margin bounds for neural networks. In *Advances in Neural Information Processing Systems*, pages 6240–6249, 2017.

Peter L Bartlett, David P Helmbold, and Philip M Long. Gradient descent with identity initialization efficiently learns positive-definite linear transformations by deep residual networks. *Neural computation*, 31(3):477–502, 2019.

Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. Interaction networks for learning about objects, relations and physics. In *Advances in Neural Information Processing Systems*, pages 4502–4510, 2016.

Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

Shai Ben-David, John Blitzer, Koby Crammer, Alex Kulesza, Fernando Pereira, and Jennifer Wortman Vaughan. A theory of learning from different domains. *Machine learning*, 79(1-2):151–175, 2010.

Alberto Bietti and Julien Mairal. On the inductive bias of neural tangent kernels. In *Advances in Neural Information Processing Systems*, pages 12873–12884, 2019.

John Blitzer, Koby Crammer, Alex Kulesza, Fernando Pereira, and Jennifer Wortman. Learning bounds for domain adaptation. In *Advances in neural information processing systems*, pages 129–136, 2008.

Jin-Yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, 1992.

Tianle Cai, Shengjie Luo, Keyulu Xu, Di He, Tie-yan Liu, and Liwei Wang. Graphnorm: A principled approach to accelerating graph neural network training. In *International Conference on Machine Learning*, 2021.

Yuan Cao and Quanquan Gu. Generalization bounds of stochastic gradient descent for wide and deep neural networks. In *Advances in Neural Information Processing Systems*, pages 10835–10845, 2019.

Quentin Cappart, Didier Chételat, Elias Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *arXiv preprint arXiv:2102.09544*, 2021.

Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27, 2011.

Michael Chang, Abhishek Gupta, Sergey Levine, and Thomas L. Griffiths. Automatically composing representation transformations as a means for generalization. In *International Conference on Learning Representations*, 2019.

Michael B Chang, Tomer Ullman, Antonio Torralba, and Joshua B Tenenbaum. A compositional object-based approach to learning physical dynamics. In *International Conference on Learning Representations*, 2017.

Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. In *International Conference on Machine Learning*, pages 942–950, 2018a.

Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations*, 2018b.

Lei Chen, Zhengdao Chen, and Joan Bruna. On graph neural networks versus graph-augmented mlps. In *International Conference on Learning Representations*, 2021.

Ming Chen, Zhewei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. Simple and deep graph convolutional networks. In *International Conference on Machine Learning*, pages 1725–1735. PMLR, 2020a.

Zhengdao Chen, Soledad Villar, Lei Chen, and Joan Bruna. On the equivalence between graph isomorphism testing and function approximation with gnns. In *Advances in Neural Information Processing Systems*, pages 15894–15902, 2019.

Zhengdao Chen, Lei Chen, Soledad Villar, and Joan Bruna. Can graph neural networks count substructures? *arXiv preprint arXiv:2002.04025*, 2020b.

Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 257–266, 2019.

Lenaic Chizat and Francis Bach. On the global convergence of gradient descent for over-parameterized models using optimal transport. In *Advances in neural information processing systems*, pages 3036–3046, 2018.

Lenaic Chizat, Edouard Oyallon, and Francis Bach. On lazy training in differentiable programming. In *Advances in Neural Information Processing Systems*, pages 2937–2947, 2019.

Anna Choromanska, Yann LeCun, and Gérard Ben Arous. Open problem: The landscape of the loss surfaces of multilayer networks. In *Proceedings of The 28th Conference on Learning Theory*, pages 1756–1760, 2015.

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

Andreea Deac, Petar Veličković, Ognjen Milinković, Pierre-Luc Bacon, Jian Tang, and Mladen Nikolić. Xlvin: executed latent value iteration nets. *arXiv preprint arXiv:2010.13146*, 2020.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3844–3852, 2016.

Nima Dehmamy, Albert-Laszlo Barabasi, and Rose Yu. Understanding the representation power of graph neural networks in learning graph topology. *Advances in Neural Information Processing Systems 32*, 2019.

James W Demmel. *Applied numerical linear algebra*, volume 56. Siam, 1997.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 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, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

Paul L DeVries and Patrick Hamill. A first course in computational physics, 1995.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Brendan L Douglas. The weisfeiler-lehman method and graph isomorphism testing. *arXiv preprint arXiv:1101.5211*, 2011.

Simon Du and Wei Hu. Width provably matters in optimization for deep linear neural networks. In *International Conference on Machine Learning*, pages 1655–1664, 2019.

Simon Du, Jason Lee, Haochuan Li, Liwei Wang, and Xiyu Zhai. Gradient descent finds global minima of deep neural networks. In *International Conference on Machine Learning*, pages 1675–1685, 2019a.

Simon S. Du and Jason D. Lee. On the power of over-parametrization in neural networks with quadratic activation. In *International Conference on Machine Learning*, 2018.

Simon S Du, Kangcheng Hou, Russ R Salakhutdinov, Barnabas Poczos, Ruosong Wang, and Keyulu Xu. Graph neural tangent kernel: Fusing graph neural networks with graph kernels. In *Advances in Neural Information Processing Systems*, pages 5724–5734, 2019b.

Simon S. Du, Xiyu Zhai, Barnabas Poczos, and Aarti Singh. Gradient descent provably optimizes over-parameterized neural networks. In *International Conference on Learning Representations*, 2019c.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul): 2121–2159, 2011.

David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, pages 2224–2232, 2015.

Sergei Evdokimov and Ilia Ponomarenko. Isomorphism of coloured graphs with slowly increasing multiplicity of jordan blocks. *Combinatorica*, 19(3):321–333, 1999.

Eugene F Fama and Kenneth R French. Common risk factors in the returns on stocks and bonds. *Journal of financial economics*, 33(1):3–56, 1993.

Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

François Fleuret, Ting Li, Charles Dubout, Emma K Wampler, Steven Yantis, and Donald Geman. Comparing machines and humans on a visual categorization test. *Proceedings of the National Academy of Sciences*, 108(43):17621–17625, 2011.

Katerina Fragkiadaki, Pulkit Agrawal, Sergey Levine, and Jitendra Malik. Learning visual predictive models of physics for playing billiards. In *International Conference on Learning Representations*, 2016.

K. Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3):183–192, 1989.

Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-adversarial training of neural networks. *The Journal of Machine Learning Research*, 17(1):2096–2030, 2016.

Michael R Garey. A guide to the theory of np-completeness. *Computers and intractability*, 1979.

Michael R Garey and David S Johnson. *Computers and intractability*, volume 29. wh freeman New York, 2002.

Vikas K Garg, Stefanie Jegelka, and Tommi Jaakkola. Generalization and representational limits of graph neural networks. In *International Conference on Machine Learning*, 2020.

Dobrik Georgiev and Pietro Lió. Neural bipartite matching. *arXiv preprint arXiv:2005.11304*, 2020.

Behrooz Ghorbani, Song Mei, Theodor Misiakiewicz, and Andrea Montanari. Linearized two-layers neural networks in high dimension. *arXiv preprint arXiv:1904.12191*, 2019.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pages 1273–1272, 2017.

Joel Goh and Melvyn Sim. Distributionally robust optimization and its tractable approximations. *Operations research*, 58(4-part-1):902–917, 2010.

Noah Golowich, Alexander Rakhlin, and Ohad Shamir. Size-independent sample complexity of neural networks. In *Conference On Learning Theory*, pages 297–299, 2018.

Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 729–734. IEEE, 2005.

Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. pages 855–864, 2016.

Pamela J Haley and DONALD Soloway. Extrapolation limitations of multilayer feedforward neural networks. In *International Joint Conference on Neural Networks*, volume 4, pages 25–30. IEEE, 1992.

William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1025–1035, 2017.

Boris Hanin and David Rolnick. Complexity of linear regions in deep networks. In *International Conference on Machine Learning*, pages 2596–2604, 2019.

Moritz Hardt and Tengyu Ma. Identity matters in deep learning. In *International Conference on Learning Representations*, 2017.

Moritz Hardt, Ben Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. In *International Conference on Machine Learning*, pages 1225–1234, 2016.

Nicholas JA Harvey and Keyulu Xu. Generating random spanning trees via fast matrix multiplication. In *LATIN 2016: Theoretical Informatics*, pages 522–535. Springer, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Matthias Hein, Maksym Andriushchenko, and Julian Bitterwolf. Why relu networks yield high-confidence predictions far away from the training data and how to mitigate the problem. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 41–50, 2019.

Dan Hendrycks, Xiaoyuan Liu, Eric Wallace, Adam Dziedzic, Rishabh Krishnan, and Dawn Song. Pretrained transformers improve out-of-distribution robustness. In *Association for Computational Linguistics*, 2020.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9 (8):1735–1780, 1997.

Elad Hoffer, Ron Banner, Itay Golan, and Daniel Soudry. Norm matters: efficient and accurate normalization schemes in deep networks. In *Advances in Neural Information Processing Systems*, pages 2160–2170, 2018.

Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, 2006.

Roger A Horn and Charles R Johnson. *Matrix analysis*. Cambridge university press, 2012.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Ronghang Hu, Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Kate Saenko. Learning to reason: End-to-end module networks for visual question answering. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 804–813, 2017.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020a.

Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. In *International Conference on Learning Representations*, 2020b.

Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. Densely connected convolutional networks. pages 2261–2269, 2017.

Jiaoyang Huang and Horng-Tzer Yau. Dynamics of deep neural networks and neural tangent hierarchy. In *International conference on machine learning*, 2020.

Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. Adaptive sampling towards fast graph representation learning. *Advances in neural information processing systems*, 31:4558–4567, 2018.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

Sergey Ivanov and Evgeny Burnaev. Anonymous walk embeddings. pages 2191–2200, 2018.

Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In *Advances in neural information processing systems*, pages 8571–8580, 2018.

Michael Janner, Sergey Levine, William T. Freeman, Joshua B. Tenenbaum, Chelsea Finn, and Jiajun Wu. Reasoning about physical interactions with object-centric models. In *International Conference on Learning Representations*, 2019.

Ziwei Ji and Matus Telgarsky. Directional convergence and alignment in deep learning. *arXiv preprint arXiv:2006.06657*, 2020.

Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2901–2910, 2017a.

Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Judy Hoffman, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Inferring and executing programs for visual reasoning. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2989–2998, 2017b.

Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

Kenji Kawaguchi. Deep learning without poor local minima. In *Advances in Neural Information Processing Systems*, pages 586–594, 2016.

Kenji Kawaguchi. On the theory of implicit deep learning: Global convergence with implicit layers. In *International Conference on Learning Representations*, 2021.

Steven Kearnes, Kevin McCloskey, Marc Berndl, Vijay Pande, and Patrick Riley. Molecular graph convolutions: moving beyond fingerprints. *Journal of computer-aided molecular design*, 30(8):595–608, 2016.

Nicolas Keriven and Gabriel Peyré. Universal invariant and equivariant graph neural networks. In *Advances in Neural Information Processing Systems*, pages 7092–7101, 2019.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2015.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. Predict then propagate: Graph neural networks meet personalized pagerank. *arXiv preprint arXiv:1810.05997*, 2018.

Johannes Klicpera, Janek Groß, and Stephan Günnemann. Directional message passing for molecular graphs. In *International Conference on Learning Representations*, 2020.

Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. pages 1885–1894, 2017.

Jonas Kohler, Hadi Daneshmand, Aurelien Lucchi, Thomas Hofmann, Ming Zhou, and Klaus Neymeyr. Exponential convergence rates for batch normalization: The power of length-direction decoupling in non-convex optimization. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 806–815, 2019.

Andrei N Kolmogorov. On tables of random numbers. *Theoretical Computer Science*, 207 (2):387–395, 1998.

V. Kurková. Kolmogorov's theorem and multilayer neural networks. *Neural networks*, 5(3): 501–506, 1992.

Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40, 2017.

Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *International Conference on Learning Representations*, 2020.

Alan Lapedes and Robert Farber. Nonlinear signal processing using neural networks: Prediction and system modelling. Technical report, 1987.

Thomas Laurent and James Brecht. Deep linear networks with arbitrary loss: All local minima are global. In *International conference on machine learning*, pages 2902–2907. PMLR, 2018.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553): 436–444, 2015.

Jaehoon Lee, Lechao Xiao, Samuel Schoenholz, Yasaman Bahri, Roman Novak, Jascha Sohl-Dickstein, and Jeffrey Pennington. Wide neural networks of any depth evolve as linear models under gradient descent. In *Advances in neural information processing systems*, pages 8572–8583, 2019.

Jure Leskovec and Julian J Mcauley. Learning to discover social circles in ego networks. pages 539–547, 2012.

Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

Chengtao Li, David Alvarez-Melis, Keyulu Xu, Stefanie Jegelka, and Suvrit Sra. Distributional adversarial networks. *arXiv preprint arXiv:1706.09549*, 2017.

Guohao Li, Matthias Muller, Ali Thabet, and Bernard Ghanem. Deepgcns: Can gcns go as deep as cnns? In *Proceedings of the IEEE International Conference on Computer Vision*, pages 9267–9276, 2019.

Guohao Li, Chenxin Xiong, Ali Thabet, and Bernard Ghanem. Deepergcn: All you need to train deeper gcns. *arXiv preprint arXiv:2006.07739*, 2020a.

Jingling Li, Mozhi Zhang, Keyulu Xu, John P Dickerson, and Jimmy Ba. Noisy labels can induce good representations. *arXiv preprint arXiv:2012.12896*, 2020b.

Mingxuan Li and Michael L Littman. Towards sample efficient agents through algorithmic alignment. *arXiv preprint arXiv:2008.03229*, 2020.

Yuanzhi Li and Yingyu Liang. Learning overparameterized neural networks via stochastic gradient descent on structured data. In *Advances in Neural Information Processing Systems*, pages 8157–8166, 2018.

Zhiyuan Li and Sanjeev Arora. An exponential learning rate schedule for deep learning. *arXiv preprint arXiv:1910.07454*, 2019.

Peiyuan Liao, Han Zhao, Keyulu Xu, Tommi Jaakkola, Geoffrey Gordon, Stefanie Jegelka, and Ruslan Salakhutdinov. Information obfuscation of graph neural networks. In *International conference on machine learning*, 2021.

Chaoyue Liu, Libin Zhu, and Mikhail Belkin. On the linearity of large non-linear models: when and why the tangent kernel is constant. *Advances in Neural Information Processing Systems*, 33, 2020.

Roi Livni, Shai Shalev-Shwartz, and Ohad Shamir. On the computational efficiency of training neural networks. In *Advances in Neural Information Processing Systems*, pages 855–863, 2014.

Andreas Loukas. How hard is to distinguish graphs with graph neural networks? In *Advances in neural information processing systems*, 2020a.

Andreas Loukas. What graph neural networks cannot learn: depth vs width. In *International Conference on Learning Representations*, 2020b.

László Lovász. Random walks on graphs. *Combinatorics, Paul erdos is eighty*, 2:1–46, 1993.

Andreas Madsen and Alexander Rosenberg Johansen. Neural arithmetic units. In *International Conference on Learning Representations*, 2020.

Takanori Maehara, Takuya Akiba, Yoichi Iwata, and Ken-ichi Kawarabayashi. Computing personalized pagerank quickly by exploiting graph structures. *Proceedings of the VLDB Endowment*, 7(12):1023–1034, 2014.

Hartmut Maennel, Olivier Bousquet, and Sylvain Gelly. Gradient Descent Quantizes ReLU Network Features. *arXiv e-prints*, art. arXiv:1803.08367, March 2018.

Yishay Mansour, Mehryar Mohri, and Afshin Rostamizadeh. Domain adaptation: Learning bounds and algorithms. In *Conference on Learning Theory*, 2009.

Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B. Tenenbaum, and Jiajun Wu. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. In *International Conference on Learning Representations*, 2019.

Haggai Maron, Heli Ben-Hamu, Hadar Serviansky, and Yaron Lipman. Provably powerful graph networks. In *Advances in Neural Information Processing Systems*, pages 2156–2167, 2019.

David B McCaughan. On the properties of periodic perceptrons. In *International Conference on Neural Networks*, 1997.

Tomas Mikolov, Quoc V Le, and Ilya Sutskever. Exploiting similarities among languages for machine translation. *arXiv preprint arXiv:1309.4168*, 2013a.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, 2013b.

Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. In *International Conference on Learning Representations*, 2018.

Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodolà, Jan Svoboda, and Michael M Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. pages 5425–5434, 2017.

Damian Mrowca, Chengxu Zhuang, Elias Wang, Nick Haber, Li F Fei-Fei, Josh Tenenbaum, and Daniel L Yamins. Flexible neural representation for physics prediction. In *Advances in Neural Information Processing Systems*, pages 8799–8810, 2018.

Ryan Murphy, Balasubramaniam Srinivasan, Vinayak Rao, and Bruno Ribeiro. Relational pooling for graph representations. In *International Conference on Machine Learning*, pages 4663–4673. PMLR, 2019.

J. A. Nelder and R. W. M. Wedderburn. Generalized linear models. *Journal of the Royal Statistical Society, Series A, General*, 135:370–384, 1972.

Behnam Neyshabur, Ryota Tomioka, and Nathan Srebro. Norm-based capacity control in neural networks. In *Proceedings of The 28th Conference on Learning Theory*, pages 1376–1401, 2015.

Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. pages 2014–2023, 2016.

Atsushi Nitanda and Taiji Suzuki. Optimal rates for averaged stochastic gradient descent under neural tangent kernel regime. In *International Conference on Learning Representations*, 2021.

Roman Novak, Lechao Xiao, Jiri Hron, Jaehoon Lee, Alexander A. Alemi, Jascha Sohl-Dickstein, and Samuel S. Schoenholz. Neural tangents: Fast and easy infinite neural networks in python. In *International Conference on Learning Representations*, 2020.

Kenta Oono and Taiji Suzuki. Graph neural networks exponentially lose expressive power for node classification. In *International Conference on Learning Representations*, 2020a.

Kenta Oono and Taiji Suzuki. Optimization and generalization analysis of transduction through gradient boosting and application to multi-scale graph neural networks. *Advances in Neural Information Processing Systems*, 33, 2020b.

Rasmus Palm, Ulrich Paquet, and Ole Winther. Recurrent relational networks. In *Advances in Neural Information Processing Systems*, pages 3368–3378, 2018.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. pages 701–710, 2014.

Jonas Peters, Peter Bühlmann, and Nicolai Meinshausen. Causal inference by using invariant prediction: identification and confidence intervals. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 78(5):947–1012, 2016.

Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237, 2018.

Omri Puny, Heli Ben-Hamu, and Yaron Lipman. From graph low-rank global attention to 2-fwl approximation. *arXiv preprint arXiv:2006.07846*, 2020.

Jorge Pérez, Javier Marinković, and Pablo Barceló. On the turing completeness of modern neural network architectures. In *International Conference on Learning Representations*, 2019.

Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, 1(2):4, 2017.

Mateo Rojas-Carulla, Bernhard Schölkopf, Richard Turner, and Jonas Peters. Invariant models for causal transfer learning. *The Journal of Machine Learning Research*, 19(1): 1309–1342, 2018.

Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. Dropedge: Towards deep graph convolutional networks on node classification. In *International Conference on Learning Representations*, 2020.

Elan Rosenfeld, Pradeep Kumar Ravikumar, and Andrej Risteski. The risks of invariant risk minimization. In *International Conference on Learning Representations*, 2021.

Stephen A Ross. The arbitrage theory of capital asset pricing. *Journal of Economic Theory*, 13(3):341–360, 1976.

Shiori Sagawa, Pang Wei Koh, Tatsunori B. Hashimoto, and Percy Liang. Distributionally robust neural networks. In *International Conference on Learning Representations*, 2020.

Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in neural information processing systems*, pages 901–909, 2016.

Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in neural information processing systems*, pages 2234–2242, 2016.

Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. Graph networks as learnable physics engines for inference and control. In *International Conference on Machine Learning*, pages 4467–4476, 2018.

Adam Santoro, David Raposo, David G Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Timothy Lillicrap. A simple neural network module for relational reasoning. In *Advances in neural information processing systems*, pages 4967–4976, 2017.

Adam Santoro, Felix Hill, David Barrett, Ari Morcos, and Timothy Lillicrap. Measuring abstract reasoning in neural networks. In *International Conference on Machine Learning*, pages 4477–4486, 2018.

Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *Advances in Neural Information Processing Systems*, pages 2483–2493, 2018.

Ryoma Sato, Makoto Yamada, and Hisashi Kashima. Approximation ratios of graph neural networks for combinatorial problems. In *Advances in Neural Information Processing Systems*, pages 4083–4092, 2019.

Ryoma Sato, Makoto Yamada, and Hisashi Kashima. Random features strengthen graph neural networks. *arXiv preprint arXiv:2002.03155*, 2020.

Pedro Savarese, Itay Evron, Daniel Soudry, and Nathan Srebro. How do infinite width bounded norm networks look in function space? 2019.

Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. In *International Conference on Learning Representations*, 2014.

David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. In *International Conference on Learning Representations*, 2019.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1): 61–80, 2009.

Franco Scarselli, Ah Chung Tsoi, and Markus Hagenbuchner. The vapnik–chervonenkis dimension of graph and recursive neural networks. *Neural Networks*, 108:248–259, 2018.

Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93, 2008.

Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep):2539–2561, 2011.

Aman Sinha, Hongseok Namkoong, and John Duchi. Certifying some distributional robustness with principled adversarial training. In *International Conference on Learning Representations*, 2018.

Vincent Sitzmann, Julien Martel, Alexander Bergman, David Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. *Advances in Neural Information Processing Systems*, 33, 2020.

Mei Song, Andrea Montanari, and P Nguyen. A mean field view of the landscape of two-layers neural networks. *Proceedings of the National Academy of Sciences*, 115: E7665–E7671, 2018.

JM Sopena and R Alquezar. Improvement of learning in recurrent networks by substituting the sigmoid activation function. In *International Conference on Artificial Neural Networks*, pages 417–420. Springer, 1994.

Daniel Soudry, Elad Hoffer, Mor Shpigel Nacson, Suriya Gunasekar, and Nathan Srebro. The implicit bias of gradient descent on separable data. *The Journal of Machine Learning Research*, 19(1):2822–2878, 2018.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Matthew Staib and Stefanie Jegelka. Distributionally robust optimization and generalization in kernel methods. In *Advances in Neural Information Processing Systems*, pages 9134–9144, 2019.

Weiwei Sun, Wei Jiang, Eduard Trulls, Andrea Tagliasacchi, and Kwang Moo Yi. Acne: Attentive context normalization for robust permutation-equivariant learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11286–11295, 2020.

Behrooz Tahmasebi and Stefanie Jegelka. Counting substructures with higher-order graph neural networks: Possibility and impossibility results. *arXiv preprint arXiv:2012.03174*, 2020.

Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. pages 1067–1077, 2015.

Kiran K Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. Attention-based graph neural network for semi-supervised learning. *arXiv preprint arXiv:1803.03735*, 2018.

Jos Thijssen. *Computational physics*. Cambridge university press, 2007.

Veronika Thost and Jie Chen. Directed acyclic graph neural networks. In *International Conference on Learning Representations*, 2021.

Andrew Trask, Felix Hill, Scott E Reed, Jack Rae, Chris Dyer, and Phil Blunsom. Neural arithmetic logic units. In *Advances in Neural Information Processing Systems*, pages 8035–8044, 2018.

Anastasios A Tsonis, Kyle L Swanson, and Paul J Roebber. What do networks have to do with climate? *Bulletin of the American Meteorological Society*, 87(5):585–595, 2006.

Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016.

Leslie G Valiant. A theory of the learnable. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 436–445. ACM, 1984.

Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 2013.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

Petar Veličković, Lars Buesing, Matthew C Overlan, Razvan Pascanu, Oriol Vinyals, and Charles Blundell. Pointer graph networks. *arXiv preprint arXiv:2006.06380*, 2020.

Petar Velickovic, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. In *International Conference on Learning Representations*, 2020.

Clement Vignac, Andreas Loukas, and Pascal Frossard. Building powerful and equivariant graph neural networks with message-passing. *Advances in neural information processing systems*, 2020.

Edward Wagstaff, Fabian B Fuchs, Martin Engelcke, Ingmar Posner, and Michael Osborne. On the limitations of representing functions on sets. In *International Conference on Machine Learning*, 2019.

Martin J Wainwright. *High-dimensional statistics: A non-asymptotic viewpoint*, volume 48. Cambridge University Press, 2019.

Nicholas Watters, Daniel Zoran, Theophane Weber, Peter Battaglia, Razvan Pascanu, and Andrea Tacchetti. Visual interaction networks: Learning a physics simulator from video. In *Advances in neural information processing systems*, pages 4539–4547, 2017.

Taylor Webb, Zachary Dulberg, Steven Frankland, Alexander Petrov, Randall O'Reilly, and Jonathan Cohen. Learning representations that support extrapolation. In *International Conference on Machine Learning*, pages 10136–10146. PMLR, 2020.

Boris Weisfeiler and AA Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968.

Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart van Merriënboer, Armand Joulin, and Tomas Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*, 2015.

Francis Williams, Matthew Trager, Daniele Panozzo, Claudio Silva, Denis Zorin, and Joan Bruna. Gradient dynamics of shallow univariate relu networks. In *Advances in Neural Information Processing Systems*, pages 8376–8385, 2019.

Jiajun Wu, Erika Lu, Pushmeet Kohli, Bill Freeman, and Josh Tenenbaum. Learning to see physics via visual de-animation. In *Advances in Neural Information Processing Systems*, pages 153–164, 2017.

Shijie Wu and Mark Dredze. Beto, bentz, becas: The surprising cross-lingual effectiveness of bert. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 833–844, 2019.

Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 3–19, 2018.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the transformer architecture. *arXiv preprint arXiv:2002.04745*, 2020.

Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. In *International Conference on Machine Learning*, pages 5453–5462, 2018.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, Ken ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? In *International Conference on Learning Representations*, 2020.

Keyulu Xu, Mozhi Zhang, Stefanie Jegelka, and Kenji Kawaguchi. Optimization of graph neural networks: Implicit acceleration by skip connections and more depth. In *International conference on machine learning*, 2021a.

Keyulu Xu, Mozhi Zhang, Jingling Li, Simon Shaolei Du, Ken-Ichi Kawarabayashi, and Stefanie Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. In *International Conference on Learning Representations*, 2021b.

Pinar Yanardag and SVN Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1365–1374. ACM, 2015.

Gilad Yehudai and Ohad Shamir. On the power and limitations of random features for understanding neural networks. In *Advances in Neural Information Processing Systems*, pages 6598–6608, 2019.

Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Josh Tenenbaum. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. In *Advances in Neural Information Processing Systems*, pages 1031–1042, 2018a.

Kwang Moo Yi, Eduard Trulls, Yuki Ono, Vincent Lepetit, Mathieu Salzmann, and Pascal Fua. Learning to find good correspondences. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2666–2674, 2018b.

Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. 2018.

Jiaxuan You, Rex Ying, and Jure Leskovec. Position-aware graph neural networks. In *International Conference on Machine Learning*, pages 7134–7143. PMLR, 2019.

Michelle Yuan, Mozhi Zhang, Benjamin Van Durme, Leah Findlater, and Jordan Boyd-Graber. Interactive refinement of cross-lingual word embeddings. In *Proceedings of Empirical Methods in Natural Language Processing*, 2020.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan R Salakhutdinov, and Alexander J Smola. Deep sets. In *Advances in Neural Information Processing Systems*, pages 3391–3401, 2017.

Chi Zhang, Feng Gao, Baoxiong Jia, Yixin Zhu, and Song-Chun Zhu. Raven: A dataset for relational and analogical visual reasoning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5317–5327, 2019a.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. In *International Conference on Learning Representations*, 2017.

Mozhi Zhang, Keyulu Xu, Ken-ichi Kawarabayashi, Stefanie Jegelka, and Jordan Boyd-Graber. Are girls neko or shōjo? cross-lingual alignment of non-isomorphic embeddings with iterative normalization. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3180–3189, 2019b.

Mozhi Zhang, Yoshinari Fujinuma, Michael Paul, and Jordan Boyd-Graber. Why overfitting isn't always bad: Retrofitting cross-lingual word embeddings to dictionaries. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2214–2220, 2020a.

Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. pages 4438–4445, 2018a.

Shuai Zhang, Meng Wang, Sijia Liu, Pin-Yu Chen, and Jinjun Xiong. Fast learning of graph neural networks with guaranteed generalizability: One-hidden-layer case. In *International Conference on Machine Learning*, pages 11268–11277, 2020b.

Yuyu Zhang, Xinshi Chen, Yuan Yang, Arun Ramamurthy, Bo Li, Yuan Qi, and Le Song. Can graph neural networks help logic reasoning? *arXiv preprint arXiv:1906.02111*, 2019c.

Zhen Zhang, Mianzhi Wang, Yijian Xiang, Yan Huang, and Arye Nehorai. Retgk: Graph kernels based on return probabilities of random walks. *arXiv preprint arXiv:1809.02670*, 2018b.

Han Zhao, Shanghang Zhang, Guanhang Wu, José MF Moura, Joao P Costeira, and Geoffrey J Gordon. Adversarial multiple source domain adaptation. In *Advances in neural information processing systems*, pages 8559–8570, 2018.

Han Zhao, Remi Tachet Des Combes, Kun Zhang, and Geoffrey Gordon. On learning invariant representations for domain adaptation. In *International Conference on Machine Learning*, pages 7523–7532, 2019.

Lingxiao Zhao and Leman Akoglu. Pairnorm: Tackling oversmoothing in gnns. In *International Conference on Learning Representations*, 2020.

Kaiyang Zhou, Yongxin Yang, Yu Qiao, and Tao Xiang. Domain generalization with mixstyle. In *International Conference on Learning Representations*, 2021.

Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. Layer-dependent importance sampling for training deep and large graph convolutional networks. In *Advances in Neural Information Processing Systems*, pages 11249–11259, 2019.

Difan Zou, Philip M Long, and Quanquan Gu. On the global convergence of training deep linear resnets. In *International Conference on Learning Representations*, 2020.