TOWARDS EXPRESSIVE AND SCALABLE DEEP REPRESENTATION LEARNING FOR GRAPHS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Rex (Zhitao) Ying
October 2021

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Jure Leskovec, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Chris Re

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Dan Yamins

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

The ubiquity of graph structures in sciences and industry necessitates effective and scalable machine learning models capable of capturing the underlying inductive biases of the relational data. However, traditional representation learning algorithms on graph structure faces many limitations. Firstly, traditional methods including matrix factorization and distributed embeddings cannot scale to large real-world graphs with billions of nodes and edges due to their sizes of parameter space. Secondly, they lack expressiveness compared to recent advances of deep learning architectures. Lastly, they fail in inductive scenarios where they required to make prediction on nodes unseen during training. Finally, interpretation of what model learns from data is elusive to domain experts.

In this thesis I present a series of work that pioneers the use of graph neural networks (GNNs) to tackle the challenges of representation learning on graphs in the aspects of explainability, scalability, and expressiveness. In the first part, I demonstrate my framework of GraphSAGE as a general but powerful overarching graph neural network framework. To tackle the challenge of model interpretability with the new GraphSAGE framework, I further introduce an extension model to obtain meaningful explanations from the trained graph neural network model.

Under the framework of GraphSAGE, the second part presents a series of works that improves the expressive power of GNNs through the use of hierarchical structure, geometric embedding space, as well as multi-hop attention. These GNN-based architectures achieved unprecedented performance improvement over traditional methods on tasks in a variety of contexts, such as graph classification for molecules, hierarchical knowledge graphs and large-scale citation networks.

In the third part, I further demonstrate a variety of applications of GNNs. Based on GraphSAGE, I developed PinSAGE, the first deployed GNN model that scales to billion-sized graphs. PinSAGE is deployed at Pinterest, to make recommendations for billions of users at Pinterest. In the area of grahics and simulations, we apply expressive architectures to accurately predict the physics of different materials and allow generalization to unseen dynamic systems. Finally, I discuss BiDyn, a dynamic GNN model for abuse detection before concluding the thesis.

Acknowledgments

Time truly flies as I make my way towards the end of the Stanford PhD program. I had always treated it as a distant dream to finally finish all my work as a PhD student and write the concluding words in my dissertation. Yet suddenly, it is time to conclude my journey at Stanford and take all the valuable lessons I learned here to my next step in life. Looking back at my journey at Stanford, I realize my words cannot fully describe how incredible it is to grow from a fresh graduate to an experienced researcher in the field of graph representation learning. Under the guidance of my advisor Jure Leskovec, my experience in the Stanford CS department not only shaped my career, but also fundamentally influenced the way I see, think and live as a researcher. I realize that knowledge is not the only thing I gain at Stanford, but also the memory of wonderful research experience and the friends I made along the way. I am deeply grateful to so many professors, lab mates and friends who supported me through my PhD experience. Without them, my accomplishment would be impossible.

First of all, I would like to deeply thank my advisor Jure Leskovec for guiding me throughout my PhD experience. Jure has been my role model since I joined the lab. I learned so much about graphs and representation learning from him over the years. However, the knowledge from his advising is only a very small portion of what I actually learned from him. Jure's insights on high-level research directions, ideas about interplay between multiple disciplines and research fields, as well as strategies for producing high-impact research have all been great inspirations for me. Working in Jure's lab has been an amazing experience. Apart from research, as I grow more senior in the lab, I even have the opportunity to teach with him, to contribute to research grant proposals, and to advise junior students in the lab. Gradually he has imparted me the necessary skills in being a successful researcher and professor, and get me ready for the next step in my career.

I am also grateful to Christopher Ré and Daniel Yamins, whose researches inspired me in many of my research projects. I am fortunate to have collaborated with Chris and his student Ines Chami. Every meeting with Chris always ends with excitement, as his clear thought process and wonderful ideas become great motivation for me to do further research and perform experiments. Daniel's research similarly inspired me. I not only ask for his advise, but also frequently discuss with his

many students on their projects in the intersection of graphs, computer vision and physics. The fruitful discussions often inspired me with new ideas and help me get unstuck during research. Chris and Dan eventually became my reading committee members and gave valuable advice to my thesis defense.

Inevitably, similar to all other PhD graduates, the path of research is full of uncertainty and difficulties. There had been countless sleepless nights, when I faced setbacks in experiments and paper writing. Looking back, I could not have overcome all of them without the brilliant and encouraging collaborators. I am fortunate to have worked with the senior students, William L. Hamilton and Marinka Zitnik. When I was new to the lab and the field of representation learning, Will and Marinka offered me so much help and got me on board with many aspects of doing research and writing papers. Thanks to their role model, my first step into graph learning research has been a painless and exciting journey. Furthermore, I also learned so much from my peers who have similar experience level as me or even more junior than me. For example, my lab mate Jiaxuan You has been an amazing researcher to work with. Over the past 4 years, together we have accomplished more than 10 projects and went to 6 conferences together to present our works. His innovative ideas and persistence are truly a gift and I very much enjoyed working with him. Similarly, I collaborated with many other lab mates and friends from other schools, including Andrew Wang, Bowen Liu, Christopher Morris, Dylan Bourgeois, Guangtao Wang, Hongyu Ren, Ines Chami, Matilde Padovano, Qi Peng, Xiang Ren, Yushi Bai, Zhihao Jia. I want to thank all of them who gave me tremendous support during the collaborations.

Interestingly, my personal growth in the PhD program is accompanied by the rapid growth of the research field I work on. When I first presented GraphSAGE in NeurIPS 2017 as a junior PhD student, graph learning was not even a subcategory in the conference, and very few people have heard of graph representation learning, or appreciated its potential. Back then it was hard to imagine that in 2021, graph representation learning would become so popular. For instance the number of submissions on graph representation learning is now on par with natural language processing in the ICLR 2021 conference. In this process, I have made many friends in academia and industry. Together we be pioneered many works in model GNNs and popularized its application in scientific domains and industry. I'm grateful for Thomas Kipf, who is a great friend and we discussed many ideas together. In the early days of graph neural networks, our works mutually inspired each other. He also later recommended me to intern at DeepMind, where I met so many great researchers in graph and relational learning, including Peter Battaglia, Alvero Sanchez, Tobias Pfaff, Jonathan Godwin, Petar Veličković, Jessica Hamrick and many others. Our fruitful collaborations on multiple publications have demonstrated wonderful applications of GNNs in diverse domains. The

internship was a truly unforgettable experience. At Stanford I am also fortunate to collaborate with many industry partners, including Siemens, Amazon, JD.com. and Aramco. For example, Jing Huang and Guangtao Wang from JD.com have become close collaboration partners of mine, and I enjoyed so much working with them on multiple projects. Furthermore, I would also like to thank Matthias Fey, the creator of PyTorch Geometric Library which empowered great number of graph learning researches. We are collaborating on new exciting projects together even after graduation. I want to thank Professor Le Song from Georgia Institute of Technology and his student Hanjun Dai too, for pioneering the area of graph neural networks together.

As a PhD student at Stanford, I also made many friends outside of academic environments. After a busy day of research, I often like to play table tennis, and have become friends to many students in the table tennis club. The joyful and relaxing time there has been a wonderful experience.

Last but not least, I would like to thank my mom Wenqun Yu, my dad Weiping Ying, and my love Xin Li. Being on the opposite side of the earth, I could still feel the love and care from my parents. Although they do not know too much about my research, they always would like to hear me talking about it. A video chat every week has always been the happiest moment for me, which washed away all negative feelings I ever experienced before. Xin and I met at Stanford when she was a master student. We share the same enthusiasm and interest about programming, computer gaming, movies and many other hobbies. Since then, Xin has supported me in so many ways. She often bring me food and care during sleepless nights of paper deadlines, or accompanied me to attend conferences. They are my dearest and I feel grateful to have shared our every moment together.

To my parents and Xin, for their unconditional love.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The ubiquity of graph structures in science and industry necessitates effective and efficient machine learning models capable of capturing the underlying inductive biases of the relational data. My research aims to learn deep representations that capture the highly complex connectivity information of the graph structure, and utilize these representations in many downstream tasks to make predictions at the level of nodes, links, subgraphs and entire graphs. My research contributes fundamental pieces of graph neural networks, which combine the high expressive power of neural networks with the inductive biases of graph connectivity information, and is applicable to diverse applications (illustrated in Figure 1).

Modeling relational data raises many fundamental challenges. **(1)** Real-world applications involve extremely large-scale datasets, such as a physical systems involving hundreds of millions of particles, and web-scale social applications with billions of interactions between users and content. They demand efficient and generalizable learning algorithms for graphs. **(2)** Rich hierarchical information is often present in relational structure, such as a gene ontology, a concept taxonomy and the occurrences of motifs in networks. Learning of such hierarchical structure for relational data is an open challenge in deep graph representation learning. **(3)** Modeling of expressive power and interpretability are two crucial aspects when analyzing complex networks with deep graph representation learning.

1.2 Contributions

My research focuses on developing scalable, expressive and interpretable deep learning algorithms to make predictions on graph-structured data. I pioneered many of the foundational methods in the field of graph neural networks (GNNs), which enabled a paradigm shift in the field of network analytics and has been delivering breakthrough performance in diverse domain applications. My research transformed the approaches of graph learning in domains such as recommender systems, knowledge graphs, and is being extended to empower scientific discovery, in chemistry and physics applications.

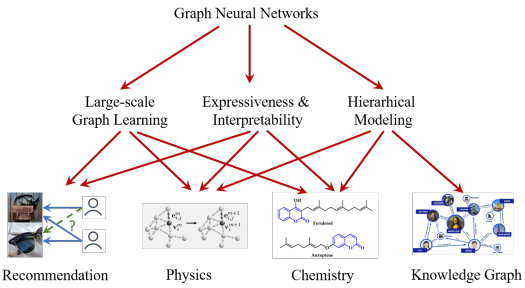


Figure 1.1: My research in graph neural networks focus on 3 key aspects, which enable a paradigm shift in graph representation learning for many scientific and social applications.

My approach to advance the field and address the challenges is to design new methodologies (with explainability) that leverage graph neural networks (GNNs) to capture structural information of the networks, and explore novel embedding geometries that learn expressive representations suitable for different graph topologies. The following is a list of key contributions:

- Proposed one of the most widely used GNN approaches (Chapter 2) and associated training techniques that enable deep representation learning for billion-scale graph-structured data.
- Created a number of state-of-the-art GNN architectures (Chapter 4, 5, 6) as well as general GNN frameworks to achieve expressiveness and interpretability (Chapter 3).
- Demonstrated applications of GNNs in a variety of scientific domains and industry use cases (Chapter 7, 8, 9).

My long-term research goal is to develop machine learning models with human-level reasoning capability. Compared to classical tasks such as classification, human-level reasoning emphasizes logical reasoning capability that can be interpreted as navigating over a graph structure via learned relations between concepts and entities. My research in learning from relational and hierarchical structures enables a new paradigm towards the goal of human-level reasoning.

1.3 Thesis Outline

This thesis consists of three parts — PART I GRAPHSAGE FRAMEWORK, EXPRESSIVE GRAPH NEURAL NETWORKS: ARCHITECTURES AND EMBEDDINGS, and SCALABLE GRAPH NEURAL NETWORKS: APPLICATIONS.

PART I focuses on the overview of GraphSAGE, which creats a framework and design space where many of the subsequent graph neural networks (GNNs) can be developed.

Chapter 2 introduces GraphSAGE, a deep learning framework for learning on graph-structured data. Previous state-of-the-art graph representation learning techniques such as DeepWalk and Node2Vec learns shallow embeddings which has important limitations: they cannot scale to extremely large networks since their parameter size scale with the number of nodes; they cannot be applied to nodes that are unseen during training. GraphSAGE uses an efficient minibatch algorithm that trains a set of aggregation functions for nodes' neighborhoods. As a result, it is efficient when scaling to extremely large graphs, and can perform inference on unseen nodes. GraphSAGE framework forms the basis of my research in GNNs, and has inspired many highly influential GNN architectures, which will be discussed in Part II.

In Chapter 3, I investigate the aspect of explainability of GNNs. The resulting model, GNNExplainer, can produce explanations for any prediction made by GNN under the GraphSAGE framework. The explanations comprise of both subgraph structure and important node features. It allows us to answer questions such as "why a molecule is classified as mutagenic", or "why a reddit user in the community is classified as malicious". Such explanations provide confidence to domain experts who use the model.

PART II extends the GraphSAGE framework and introduces advanced GNN architectures that have high expressive power and are suitable for specific graph learning tasks.

Chapter 4 explains the first learnable pooling mechanism to aggregate information of subgraphs in hierarchical ways. I built the model DiffPool, which learns graph embeddings by learning a hierarchically strategy to pool nodes at different levels of abstractions, and produce a vector embedding for the entire graph. The important innovation is to allow model to learn a cluster assignment strategy that is best suited for specific graph datasets.

In Chapter 5, we showcase the hyperbolic graph convolutional networks (HGCN), which performs graph convolution operations in hyperbolic space with trainable curvature of the hyperbolic space at every hidden layer. This architecture leverages the inductive bias of tree-like hierarchical structure that is present in many graph datasets. HGCN can out-performs state-of-the-art GNN models by 60% in area under receiver operating characteristic (AUROC) on tree-like hierarchical graphs, and no previous GNN variants have achieved improvements of this magnitude on these hierarchical graphs.

Chapter 6 presents another expressive GNN architecture, MAGNA, under the GraphSAGE framework. Specifically, it uses multi-hop neighborhood extraction via diffusion, and attention aggregation to achieve state-of-the-art results in most graph benchmarks.

PART III demonstrate the use of GNNs in real-world large-scale applications.

Chapter 7 introduces PinSAGE, which adapts and extends GraphSAGE to build the first deployed GNN-based recommender systems. We successfully trained our model on recommendation data 1000 times larger than the largest GNN-based recommender system at the time. PinSage is capable of making accurate recommendations that take into account of both graph structure and user/item features. Since 2018, PinSAGE has been deployed in production at Pinterest, and further extended to a user/pin deep embedding service used throughout the company.

Chapter 8 focuses on GNN for simulation. In collaboration with DeepMind Inc., I extend large-scale graph representation learning in the domain of simulation. I tackled the problem of simulating physical systems via particle interactions. We built a model that learns simulation leveraging the capability of GNN to learn particle interactions, and demonstrated that the model is capable of simulating complex interactions between objects of a variety of shapes and materials, and achieved state-of-the-art performance in learning large-scale and realistic physical systems.

Finally, Chapter 9 introduces Bidyn, a graph neural networks for tackling the challenging problem of e-commerce abuse detection. We create a dynamic GNN model, which effectively synergizes with a scalable training scheme and pretraining strategy. Bidyn dramatically improves the abuse detection performance and is now being deployed at Amazon.

We will finally conclude in Chapter 10.

Part I

Representation Learning with Graph Neural Networks: the GraphSAGE Framework

Chapter 2

An Overview of GraphSAGE

This chapter focuses on an overview of GraphSAGE, a generic graph neural network framework for representation learning on graphs. The goal is to leverage the expressive power of neural networks to obtain vector representations for nodes, edges and graphs in a given dataset, which can then be used to make predictions on nodes, edges and graphs based on graph structure and features associated with nodes, edges and graphs. We introduce the two key components of the framework, namely the neighborhood extraction (sampling), and the neighborhood aggregation neural operation.

2.1 Introduction

Low-dimensional vector embeddings of nodes in large graphs[1] have proved extremely useful as feature inputs for a wide variety of prediction and graph analysis tasks (Cao et al., 2015; Grover and Leskovec, 2016; Perozzi et al., 2014; Tang et al., 2015; Wang et al., 2016). The basic idea behind node embedding approaches is to use dimensionality reduction techniques to distill the high-dimensional information about a node's neighborhood into a dense vector embedding. These node embeddings can then be fed to downstream machine learning systems and aid in tasks such as node classification, clustering, and link prediction (Grover and Leskovec, 2016; Perozzi et al., 2014; Tang et al., 2015).

However, previous works have focused on embedding nodes from a single fixed graph, and many real-world applications require embeddings to be quickly generated for unseen nodes, or entirely new (sub)graphs. This inductive capability is essential for high-throughput, production machine learning systems, which operate on evolving graphs and constantly encounter unseen nodes

[1]While it is common to refer to these data structures as social or biological *networks*, we use the term *graph* to avoid ambiguity with neural network terminology.

(e.g., posts on Reddit, users and videos on Youtube). An inductive approach to generating node embeddings also facilitates generalization across graphs with the same form of features: for example, one could train an embedding generator on protein-protein interaction graphs derived from a model organism, and then easily produce node embeddings for data collected on new organisms using the trained model.

The inductive node embedding problem is especially difficult, compared to the transductive setting, because generalizing to unseen nodes requires "aligning" newly observed subgraphs to the node embeddings that the algorithm has already optimized on. An inductive framework must learn to recognize structural properties of a node's neighborhood that reveal both the nodes local role in the graph, as well as its global position.

Most existing approaches to generating node embeddings are inherently transductive. The majority of these approaches directly optimize the embeddings for each node using matrix-factorization-based objectives, and do not naturally generalize to unseen data, since they make predictions on nodes in a single, fixed graph (Cao et al., 2015; Grover and Leskovec, 2016; Ng et al., 2001; Perozzi et al., 2014; Tang et al., 2015; Wang et al., 2016, 2017; Xu et al., 2017). These approaches can be modified to operate in an inductive setting (e.g., (Perozzi et al., 2014)), but these modifications tend to be computationally expensive, requiring additional rounds of gradient descent before new predictions can be made. There are also recent approaches to learning over graph structures using convolution operators that offer promise as an embedding methodology (Kipf and Welling, 2016a). So far, graph convolutional networks (GCNs) have only been applied in the transductive setting with fixed graphs (Kipf and Welling, 2016a,b). In this work we both extend GCNs to the task of inductive, unsupervised learning and propose a framework which generalizes the GCN approach to use trainable aggregation functions (beyond simple convolutions).

Present work. We propose a general framework, called GRAPHSAGE (SAmple and aggreGatE), for inductive node embedding. Unlike embedding approaches that are based on matrix factorization, we leverage node features (e.g., text attributes, node profile information, node degrees) in order to learn an embedding function that generalizes to unseen nodes. By incorporating node features in the learning algorithm, we simultaneously learn the topological structure of each node's neighborhood as well as the distribution of node features in the neighborhood. While we focus on feature-rich graphs (e.g., citation data with text attributes, biological data with functional/molecular markers), our approach can also make use of structural features that are present in all graphs (e.g., node degrees). Thus, our algorithm can also be applied to graphs without node features.

Instead of training a distinct embedding vector for each node, we train a set of *aggregator functions* that learn to recursively aggregate feature information from a node's local neighborhood

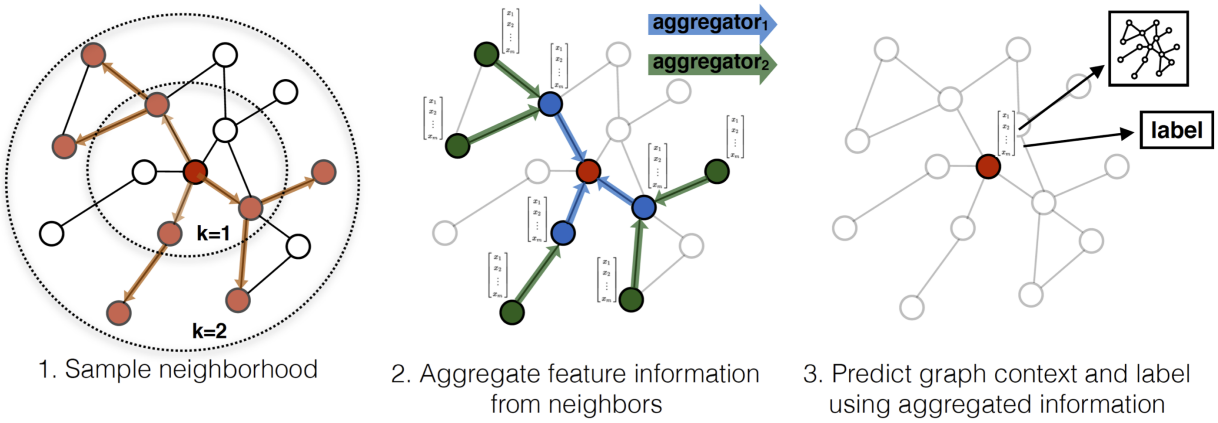


Figure 2.1: Visual illustration of the GRAPHSAGE sample and aggregate approach.

(Figure 2.1). Each aggregator function aggregates information from a different number of hops, or search depth, away from a given node. At test, or inference time, we use our trained system to generate embeddings for entirely unseen nodes by applying the learned aggregation functions. Following previous work on generating node embeddings, we design an unsupervised loss function that allows GRAPHSAGE to be trained without task-specific supervision. We also show that GRAPHSAGE can be trained in a fully supervised manner.

We evaluate our algorithm on three node-classification benchmarks, which test GRAPHSAGE's ability to generate useful embeddings on unseen data. We use two evolving document graphs based on citation data and Reddit post data (predicting paper and post categories, respectively), and a multi-graph generalization experiment based on a dataset of protein-protein interactions (predicting protein functions). Using these benchmarks, we show that our approach is able to effectively generate representations for unseen nodes and outperform relevant baselines by a significant margin: across domains, our supervised approach improves classification accuracy by an average of 43% compared to using node features alone and GRAPHSAGE consistently outperforms a strong, transductive baseline (Perozzi et al., 2014), despite this baseline taking $\sim$100$\times$ longer to run on unseen nodes. We also show that the new aggregator architectures we propose provide significant gains (6.5% on average) compared to an aggregator inspired by graph convolutional networks (Kipf and Welling, 2016a). Lastly, we probe the expressive capability of our approach and show, through theoretical analysis, that GRAPHSAGE is capable of learning structural information about a node's role in a graph, despite the fact that it is inherently based on features (Section 2.5).

2.2 Comparison to Prior Graph Representation Learning Approaches

Our algorithm is conceptually related to previous node embedding approaches, general supervised approaches to learning over graphs, and recent advancements in applying convolutional neural networks to graph-structured data.

Factorization-based embedding approaches. There are a number of recent node embedding approaches that learn low-dimensional embeddings using random walk statistics and matrix factorization-based learning objectives (Cao et al., 2015; Grover and Leskovec, 2016; Perozzi et al., 2014; Tang et al., 2015; Wang et al., 2016). These methods also bear close relationships to more classic approaches to spectral clustering (Ng et al., 2001), multi-dimensional scaling (Kruskal, 1964), as well as the PageRank algorithm (Page et al., 1999). Since these embedding algorithms directly train node embeddings for individual nodes, they are inherently transductive and, at the very least, require expensive additional training (e.g., via stochastic gradient descent) to make predictions on new nodes. In addition, for many of these approaches (e.g., (Grover and Leskovec, 2016; Perozzi et al., 2014; Tang et al., 2015; Wang et al., 2016)) the objective function is invariant to orthogonal transformations of the embeddings, which means that the embedding space does not naturally generalize between graphs and can drift during re-training. One notable exception to this trend is the Planetoid-I algorithm introduced by (Yang et al., 2016), which is an inductive, embedding-based approach to semi-supervised learning. However, Planetoid-I does not use any graph structural information during inference; instead, it uses the graph structure as a form of regularization during training. Unlike these previous approaches, we leverage feature information in order to train a model to produce embeddings for unseen nodes.

Supervised learning over graphs. Beyond node embedding approaches, there is a rich literature on supervised learning over graph-structured data. This includes a wide variety of kernel-based approaches, where feature vectors for graphs are derived from various graph kernels (see (Shervashidze et al., 2011b) and references therein). There have also been a number of recent neural network approaches to supervised learning over graph structures (Dai et al., 2016a; Gori et al., 2005; Li et al., 2015; Scarselli et al., 2009b). Our approach is conceptually inspired by a number of these algorithms. However, whereas these previous approaches attempt to classify entire graphs (or subgraphs), the focus of this work is generating useful representations for individual nodes.

Graph convolutional networks. In recent years, several convolutional neural network architectures for learning over graphs have been proposed (e.g., (Bruna et al., 2014b; Duvenaud et al., 2015c; Defferrard et al., 2016b; Kipf and Welling, 2016a; Niepert et al., 2016)). The majority of

these methods do not scale to large graphs or are designed for whole-graph classification (or both) (Bruna et al., 2014b; Duvenaud et al., 2015c; Defferrard et al., 2016b; Niepert et al., 2016). Our approach is closely related to the graph convolutional network (GCN), introduced by Kipf et al. (Kipf and Welling, 2016a,b). The original GCN algorithm (Kipf and Welling, 2016a) is designed for semi-supervised learning in a transductive setting, and the exact algorithm requires that the full graph Laplacian is known during training. A simple variant of our algorithm can be viewed as an extension of the GCN framework to the setting of inductive, unsupervised learning, a point which we revisit in Section 2.3.3.

2.3 Proposed Framework: GRAPHSAGE

The key idea behind our approach is that we learn how to aggregate feature information from a node's local neighborhood (e.g., the degrees or text attributes of nearby nodes). We first describe the GRAPHSAGE embedding generation (i.e., forward propagation) algorithm, which generates embeddings for nodes assuming that the GRAPHSAGE model parameters are already learned (Section 2.3.1). We then describe how the GRAPHSAGE model parameters can be learned using standard stochastic gradient descent and backpropagation techniques (Section 2.3.2).

2.3.1 Embedding generation (i.e., forward propagation) algorithm

Algorithm 1: GRAPHSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\{\mathbf{W}^k, \forall k \in [1, K]\}$; non-linearity σ; differentiable aggregator functions $\{\text{AGGREGATE}_k, \forall k \in [1, K]\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output: Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  end
  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

In this section, we describe the embedding generation, or forward propagation algorithm (Algorithm 1), which assumes that the model has already been trained and that the parameters are fixed. In particular, we assume that we have learned the parameters of K aggregator functions (denoted $\{\text{AGGREGATE}_k, \forall k \in [1, K]\}$), which aggregate information from node neighbors, as well as a set of weight matrices $\{\mathbf{W}^k, \forall k \in [1, K]\}$, which are used to propagate information between different layers of the model or "search depths". Section 2.3.2 describes how we train these parameters.

The intuition behind Algorithm 1 is that at each iteration, or search depth, nodes aggregate information from their local neighbors, and as this process iterates, nodes incrementally gain more and more information from further reaches of the graph.

Algorithm 1 describes the embedding generation process in the case where the entire graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and features for all nodes $\mathbf{x}_v, \forall v \in \mathcal{V}$, are provided as input. We describe how to generalize this to the minibatch setting below. Each step in the outer loop of Algorithm 1 proceeds as follows, where k denotes the current step in the outer loop (or the "depth" of the recursive search) and $\mathbf{h}^k$ denotes a node's representation at this step: First, each node $v \in \mathcal{V}$ aggregates the representations of the nodes in its immediate neighborhood, $\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\}$, into a single vector $\mathbf{h}_{\mathcal{N}(v)}^{k-1}$. Note that this aggregation step depends on the representations generated at the previous iteration of the outer loop (i.e., $k-1$), and the $k = 0$ ("base case") representations are defined as the input node features. After aggregating the neighboring feature vectors, GRAPHSAGE then concatenates the node's current representation, $\mathbf{h}_v^{k-1}$, with the aggregated neighborhood vector, $\mathbf{h}_{\mathcal{N}(v)}^{k-1}$, and this concatenated vector is fed through a fully connected layer with nonlinear activation function σ, which transforms the representations to be used at the next step of the algorithm (i.e., $\mathbf{h}_v^k, \forall v \in \mathcal{V}$). For notational convenience, we denote the final representations output at depth K as $\mathbf{z}_v \equiv \mathbf{h}_v^K, \forall v \in \mathcal{V}$. The aggregation of the neighbor representations can be done by a variety of aggregator architectures (denoted by the AGGREGATE placeholder in Algorithm 1), and we discuss different architecture choices in Section 2.3.3 below.

To extend Algorithm 1 to the minibatch setting, given a set of input nodes, we first forward sample the required neighborhood sets (up to depth K) and then we run the inner loop (line 3 in Algorithm 1), but instead of iterating over all nodes, we compute only the representations that are necessary to satisfy the recursion at each depth.

Relation to the Weisfeiler-Lehman Isomorphism Test. The GRAPHSAGE algorithm is conceptually inspired by a classic algorithm for testing graph isomorphism. If, in Algorithm 1, we (i) set $K = |\mathcal{V}|$, (ii) set the weight matrices as the identity, and (iii) use an appropriate hash function as an aggregator (with no non-linearity), then Algorithm 1 is an instance of the Weisfeiler-Lehman (WL) isomorphism test, also known as "naive vertex refinement" (Shervashidze et al., 2011b). If

the set of representations $\{\mathbf{z}_v, \forall v \in \mathcal{V}\}$ output by Algorithm 1 for two subgraphs are identical then the WL test declares the two subgraphs to be isomorphic. This test is known to fail in some cases, but is valid for a broad class of graphs (Shervashidze et al., 2011b). GRAPHSAGE is a continuous approximation to the WL test, where we replace the hash function with trainable neural network aggregators. Of course, we use GRAPHSAGE to generate useful node representations–not to test graph isomorphism. Nevertheless, the connection between GRAPHSAGE and the classic WL test provides theoretical context for our algorithm design to learn the topological structure of node neighborhoods.

Neighborhood definition. In this work, we uniformly sample a fixed-size set of neighbors, instead of recursing on full neighborhood sets in Algorithm 1, in order to keep the computational footprint of each batch fixed.[2] That is, using overloaded notation, we define $\mathcal{N}(v)$ as a fixed-size, uniform draw from the set $\{u \in \mathcal{V} : (u, v) \in \mathcal{E}\}$, and we draw different uniform samples at each iteration, k, in Algorithm 1. Without this sampling the memory and expected runtime of a single batch is unpredictable and in the worst case $O(|\mathcal{V}|)$. In contrast, the per-batch space and time complexity for GRAPHSAGE is fixed at $O(\prod_{i=1}^{K} S_i)$, where $\{S_i, i \in [1, ..., K]\}$ and K are user-specified constants. Practically speaking we found that our approach could achieve high performance with $K = 2$ and $S_1 \cdot S_2 \leq 500$ (Section 2.4.3).

2.3.2 Learning the parameters of GRAPHSAGE

In order to learn useful, predictive representations in a fully unsupervised setting, we apply a graph-based loss function to the output representations, $\{\mathbf{z}_u, \forall u \in \mathcal{V}\}$, and tune the weight matrices ($\{\mathbf{W}^k\}, \forall k \in [1, K]\}$) and parameters of the aggregator functions via stochastic gradient descent. The graph-based loss function encourages nearby nodes to have similar representations, while enforcing that the representations of disparate nodes are highly distinct:

$$J_{\mathcal{G}}(\mathbf{z}_u) = -\log\left(\sigma(\mathbf{z}_u^{\top}\mathbf{z}_v)\right) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log\left(\sigma(-\mathbf{z}_u^{\top}\mathbf{z}_{v_n})\right), \tag{2.1}$$

where v is a node that co-occurs near u on fixed-length random walk, σ is the sigmoid function, P_n is a negative sampling distribution, and Q defines the number of negative samples. Importantly, unlike previous embedding approaches, the representations $\mathbf{z}_u$ that we feed into this loss function are generated from the features contained within a node's local neighborhood, rather than training a unique embedding for each node (via an embedding look-up).

This unsupervised setting emulates situations where node features are provided to downstream

[2] Exploring non-uniform samplers is an important direction for future work.

machine learning applications, as a service or in a static repository. In cases where representations are to be used only on a specific downstream task, the unsupervised loss (Equation 2.1) can simply be replaced, or augmented, by a task-specific objective (e.g., cross-entropy loss).

2.3.3 Aggregator Architectures

Unlike machine learning over N-D lattices (e.g., sentences, images, or 3-D volumes), a node's neighbors have no natural ordering; thus, the aggregator functions in Algorithm 1 must operate over an unordered set of vectors. Ideally, an aggregator function would be symmetric (i.e., invariant to permutations of its inputs) while still being trainable and maintaining high representational capacity. The symmetry property of the aggregation function ensures that our neural network model can be trained and applied to arbitrarily ordered node neighborhood feature sets. We examined three candidate aggregator functions:

Mean aggregator. Our first candidate aggregator function is the mean operator, where we simply take the elementwise mean of the vectors in $\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\}$. The mean aggregator is nearly equivalent to the convolutional propagation rule used in the transductive GCN framework (Kipf and Welling, 2016a). In particular, we can derive an inductive variant of the GCN approach by replacing lines 4 and 5 in Algorithm 1 with the following:[3]

$$\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W} \cdot \text{MEAN}(\{\mathbf{h}_v^{k-1}\} \cup \{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\}). \tag{2.2}$$

We call this modified mean-based aggregator *convolutional* since it is a rough, linear approximation of a localized spectral convolution (Kipf and Welling, 2016a). The mean and convolutional aggregators are efficient and naturally symmetric, but they lack in expressive power and include no trainable parameters.

LSTM aggregator. We also examined a more complex aggregator based on an LSTM architecture (Hochreiter and Schmidhuber, 1997). Compared to the mean aggregator, LSTMs have the advantage of larger expressive capability. However, it is important to note that LSTMs are not inherently symmetric (i.e., they are not permutation invariant), since they process their inputs in a sequential manner. We adapt LSTMs to operate on an unordered set by simply applying the LSTMs to a random permutation of a node's neighbors.

Pooling aggregator. The final aggregator we examine is both symmetric and trainable. In this *pooling* approach, each neighbor's vector is independently fed through a fully-connected neural

[3] Note that this differs from Kipf et al's exact equation by a minor normalization constant (Kipf and Welling, 2016a).

network; following this transformation, an elementwise max-pooling operation is applied to aggregate information across the neighbor set:

$$\text{AGGREGATE}_k^{\text{pool}} = \max(\{\sigma\left(\mathbf{W}_{\text{pool}}\mathbf{h}_{u_i}^k + \mathbf{b}\right), \forall u_i \in \mathcal{N}(v)\}), \tag{2.3}$$

where $\max$ denotes the element-wise max operator and σ is a nonlinear activation function. In principle, the function applied before the max pooling can be an arbitrarily deep multi-layer perceptron, but we focus on simple single-layer architectures in this work. This approach is inspired by recent advancements in applying neural network architectures to learn over general point sets (Qi et al., 2017). Intuitively, the multi-layer perceptron can be thought of as a set of functions that compute features for each of the node representations in the neighbor set. By applying the max-pooling operator to each of the computed features, the model effectively captures different aspects of the neighborhood set.

2.4 GraphSAGE Performance

We test the performance of GRAPHSAGE on three benchmark tasks: classifying academic papers into different subjects using the Web of Science citation dataset, classifying Reddit posts as belonging to different communities, and classifying protein functions across various biological protein-protein interaction (PPI) graphs. Sections 2.4.1 and 2.4.2 summarize the datasets, and the supplementary material contains additional information. In all these experiments, we perform predictions on nodes that are not seen during training, and, in the case of the PPI dataset, we test on entirely unseen graphs.

Experimental set-up. To contextualize the empirical results on our inductive benchmarks, we compare against four baselines: a random classifer, a logistic regression feature-based classifier (that ignores graph structure), the DeepWalk algorithm (Perozzi et al., 2014) as a representative factorization-based approach, and a concatenation of the raw features and DeepWalk embeddings. We also compare four variants of GRAPHSAGE that use the different aggregator functions (Section 2.3.3). Since, the "convolutional" variant of GRAPHSAGE is an extended, inductive version of Kipf et al's semi-supervised GCN (Kipf and Welling, 2016a), we term this variant GRAPHSAGE-GCN. We test unsupervised variants of GRAPHSAGE trained according to the loss in Equation (2.1), as well as supervised variants that are trained directly on classification cross-entropy loss. For all the GRAPHSAGE variants we used rectified linear units as the non-linearity and set $K = 2$ with neighborhood sample sizes $S_1 = 25$ and $S_2 = 10$ (see Section 2.4.3 for sensitivity analyses).

For the Reddit and citation datasets, we use "online" training for DeepWalk as described in

(Perozzi et al., 2014), where we run a new round of SGD optimization to embed the new test nodes before making predictions, while keeping the embeddings for the training nodes fixed. In the multi-graph setting, we cannot apply DeepWalk, since the embedding spaces generated by running the DeepWalk algorithm on different disjoint graphs can be arbitrarily rotated with respect to each other.

All models were implemented in Tensorflow (Abadi et al., 2016) with the Adam optimizer (Kingma and Ba, 2015) (except DeepWalk, which performed better with the vanilla gradient descent optimizer). In order to provide a fair comparison, all models also share an identical implementation of their minibatch iterators, loss function and neighborhood sampler (when applicable).
Hyperparameter settings. In all settings, we performed hyperparameter selection on the learning rate and the model dimension. With the exception of DeepWalk, we performed a parameter sweep on initial learning rates $\{0.01, 0.001, 0.0001\}$ for the supervised models and $\{0.001, 0.0001, 0.00001\}$ for the unsupervised models. When applicable, we tested a "big" and "small" version of each model, where we tried to keep the overall model sizes comparable. For the pooling aggregator, the "big" model had a pooling dimension of 1024, while the "small" model had a dimension of 512. For the LSTM aggregator, the "big" model had a hidden dimension of 256, while the "small" model had a hidden dimension of 128; note that the actual parameter count for the LSTM is roughly $4\times$ this number, due to weights for the different gates. In all experiments and for all models we specify the output dimension of the $\mathbf{h}_i^k$ vectors at every depth k of the recursion to be 256. All models use rectified linear units as a non-linear activation function. All the unsupervised GRAPHSAGE models and DeepWalk used 20 negative samples with context distribution smoothing over node degrees using a smoothing parameter of 0.75, following (Grover and Leskovec, 2016; Mikolov et al., 2013; Perozzi et al., 2014). Initial experiments revealed that DeepWalk performed much better with large learning rates, so we swept over rates in the set $\{0.2, 0.4, 0.8\}$. For the supervised GRAPHSAGE methods, we ran 10 epochs for all models. All methods except DeepWalk use batch sizes of 512. We found that DeepWalk achieved faster wall-clock convergence with a smaller batch size of 64.

2.4.1 Inductive learning on evolving graphs: Citation and Reddit data

Our first two experiments are on classifying nodes in evolving information graphs, a task that is especially relevant to high-throughput production systems, which constantly encounter unseen data.
Citation data. Our first task is predicting paper subject categories on a large citation dataset. We use an undirected citation graph dataset derived from the Thomson Reuters Web of Science Core Collection, corresponding to all papers in six biology-related fields for the years 2000-2005.

Table 2.1: Prediction results for the three datasets (micro-averaged F1 scores). Results for unsupervised and fully supervised GRAPHSAGE are shown. Analogous trends hold for macro-averaged scores.

	Citation		Reddit		PPI	
Name	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1
Random	0.206	0.206	0.043	0.042	0.396	0.396
Raw features	0.575	0.575	0.585	0.585	0.422	0.422
DeepWalk	0.565	0.565	0.324	0.324	—	—
DeepWalk + features	0.701	0.701	0.691	0.691	—	—
GRAPHSAGE-GCN	0.742	0.772	**0.908**	0.930	0.465	0.501
GRAPHSAGE-mean	0.778	0.820	0.897	0.950	0.486	0.551
GRAPHSAGE-LSTM	0.788	0.832	**0.907**	**0.954**	0.482	**0.598**
GRAPHSAGE-pool	**0.798**	**0.839**	0.892	0.948	**0.502**	0.557
% gain over feat.	39%	46%	55%	63%	19%	41%

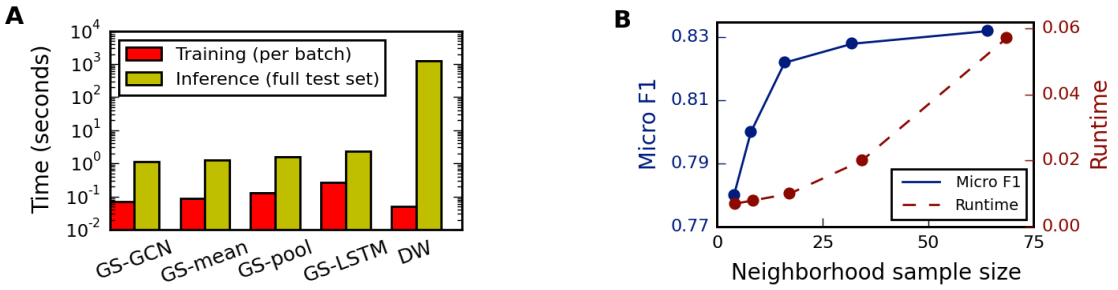


Figure 2.2: **A**: Timing experiments on Reddit data, with training batches of size 512 and inference on the full test set (79,534 nodes). **B**: Model performance with respect to the size of the sampled neighborhood, where the "neighborhood sample size" refers to the number of neighbors sampled at each depth for $K = 2$ with $S_1 = S_2$ (on the citation data, using GRAPHSAGE-mean).

The node labels for this dataset correspond to the six different field labels. In total, this is dataset contains 302,424 nodes with an average degree of 9.15. We train all the algorithms on the 2000-2004 data and use the 2005 data for testing (with 30% used for validation). For features, we used node degrees and processed the paper abstracts according Arora et al (2017)'s (Arora et al., 2017) sentence embedding approach, with 300-dimensional word vectors trained using GenSim (Řehůřek and Sojka, 2010).

Reddit data. In our second task, we predict which community different Reddit posts belong to. Reddit is a large online discussion forum where users post and comment on content in different topical communities. We constructed a graph dataset from Reddit posts made in the month of September, 2014. The node label in this case is the community, or "subreddit", that a post belongs to. We sampled 50 large communities and built a post-to-post graph, connecting posts if the same user comments on both. In total this dataset contains 232,965 posts with an average degree of 492. We use the first 20 days for training and the remaining days for testing (with 30% used for validation). For features, we use off-the-shelf 300-dimensional GloVe CommonCrawl word vectors (Pennington et al., 2014); for each post, we concatenated (i) the average embedding of the post title, (ii) the average embedding of all the post's comments (iii) the post's score, and (iv) the number of comments made on the post.

The first four columns of Table 2.1 summarize the performance of GRAPHSAGE as well as the baseline approaches on these two datasets. We find that GRAPHSAGE outperforms all the baselines by a significant margin, and the trainable, neural network aggregators provide significant gains compared to the GCN approach. For example, the unsupervised variant GRAPHSAGE-pool outperforms the concatenation of the DeepWalk embeddings and the raw features by 13.8% on the citation data and 29.1% on the Reddit data, while the supervised version provides a gain of 19.7% and 37.2%, respectively. Interestingly, the LSTM based aggregator shows strong performance, despite the fact that it is designed for sequential data and not unordered sets. Lastly, we see that the performance of unsupervised GRAPHSAGE is reasonably competitive with the fully supervised version, indicating that our framework can achieve strong performance without task-specific fine-tuning.

2.4.2 Generalizing across graphs: Protein-protein interactions

We now consider the task of generalizing across graphs, which requires learning about node roles rather than community structure. We classify protein roles—in terms of their cellular functions from gene ontology—in various protein-protein interaction (PPI) graphs, with each graph corresponding to a different human tissue (Zitnik and Leskovec, 2017). We use positional gene sets,

motif gene sets and immunological signatures as features and gene ontology sets as labels (121 in total), collected from the Molecular Signatures Database (Subramanian et al., 2005). The average graph contains 3000 nodes, with an average degree of 28.8. We train all algorithms on 20 graphs and then average prediction F1 scores on 2 test graphs (with 2 other graphs used for validation).

The final two columns of Table 2.1 summarize the accuracies of the various approaches on this data. Again we see that GRAPHSAGE significantly outperforms the baseline approaches, with the LSTM and max-pooling based aggregators providing substantial gains over the mean and GCN-based aggregators.

2.4.3 Runtime and parameter sensitivity

Figure 2.2.A summarizes the training and test runtimes for the different approaches. The training time for the methods are comparable (with GRAPHSAGE-LSTM being the slowest). However, the need to sample new random walks and run new rounds of SGD to embed unseen nodes makes DeepWalk $100\text{-}500\times$ slower at test time.

For the GRAPHSAGE variants, we found that setting $K = 2$ provided a consistent boost in accuracy of around $10\text{-}15\%$, on average, compared to $K = 1$; however, increasing K beyond 2 gave marginal returns in performance ($0\text{-}5\%$) while increasing the runtime by a prohibitively large factor of $10\text{-}100\times$, depending on the neighborhood sample size. We also found diminishing returns for sampling large neighborhoods (Figure 2.2.B). Thus, despite the higher variance induced by sub-sampling neighborhoods, GRAPHSAGE is still able to maintain strong predictive accuracy, while significantly improving the runtime.

2.5 Theoretical analysis

In this section, we probe the expressive capabilities of GRAPHSAGE in order to provide insight into how GRAPHSAGE can learn about graph structure, even though it is inherently based on features. As a case-study, we consider whether GRAPHSAGE can learn to predict the clustering coefficient of a node, i.e. the proportion of triangles that are closed within the node's 1-hop neighborhood (Watts and Strogatz, 1998). The clustering coefficient is a popular measure of how clustered a node's local neighborhood is, and it serves as a building block for many more complicated structural motifs (Benson et al., 2016). We can show that Algorithm 1 is capable of approximating clustering coefficients to an arbitrary degree of precision:

Theorem 1 *Let $\mathbf{x}_v \in U, \forall v \in \mathcal{V}$ denote the feature inputs for Algorithm 1 on graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where U is any compact subset of $\mathbb{R}^d$. Suppose that there exists a fixed positive constant $C \in \mathbb{R}^+$ such that $\|\mathbf{x}_v - \mathbf{x}_{v'}\|_2 > C$ for all pairs of nodes. Then we have that $\forall \epsilon > 0$ there exists a parameter setting $\mathbf{\Theta}^*$ for Algorithm 1 such that after $K = 4$ iterations*

$$|z_v - c_v| < \epsilon, \forall v \in \mathcal{V},$$

where $z_v \in \mathbb{R}$ are final output values generated by Algorithm 1 and c_v are node clustering coefficients.

Theorem 1 states that for any graph there exists a parameter setting for Algorithm 1 such that it can approximate clustering coefficients in that graph to an arbitrary precision, if the features for every node are distinct (and if the model is sufficiently high-dimensional). Note that as a corollary of Theorem 1, GRAPHSAGE can learn about local graph structure, even when the node feature inputs are sampled from an absolutely continuous random distribution (see the original publication for a full proof). The basic idea behind the proof is that if each node has a unique feature representation, then we can learn to map nodes to indicator vectors and identify node neighborhoods. The proof of Theorem 1 relies on some properties of the pooling aggregator, which also provides insight into why GRAPHSAGE-pool outperforms the GCN and mean-based aggregators.

2.6 Conclusion

The GraphSAGE framework as introduced in Chapter 2 comprises of 2 important components: **neighborhood extraction** (the sampling step), and **feature aggregation** of neighborhood (the aggregation step). The exact neighborhood extraction algorithm and feature aggregation operators are flexible and have been extended into many powerful GNN architectures.

Chapter 3

GNNExplainer: Towards Explainable GNNs

To apply GraphSAGE in real-world graph learning applications, it is often not sufficient to only make accurate predictions on nodes, edges and graphs. Although not previously well-studied, model explainability for graph neural networks is equally important, since domain experts often need to gain understanding of why the given model makes certain predictions. This chapter introduces GNNExplainer, the first model that can be applied to any trained GNN under the GraphSAGE framework, and produce insights and explanations about the underlying reason for its predictions on unseen nodes, edges and graphs.

3.1 Introduction

In many real-world applications, including social, information, chemical, and biological domains, data can be naturally modeled as graphs (Cho et al., 2011; You et al., 2018a; Zitnik et al., 2018a). Graphs are powerful data representations but are challenging to work with because they require modeling of rich relational information as well as node feature information (Zhang et al., 2018c; Zhou et al., 2018). To address this challenge, Graph Neural Networks (GNNs) have emerged as state-of-the-art for machine learning on graphs, due to their ability to recursively incorporate information from neighboring nodes in the graph, naturally capturing both graph structure and node features (Hamilton et al., 2017a; Kipf and Welling, 2016a; Ying et al., 2018c; Zhang and Chen, 2018).

Despite their strengths, GNNs lack transparency as they do not easily allow for a human-intelligible explanation of their predictions. Yet, the ability to understand GNN's predictions is

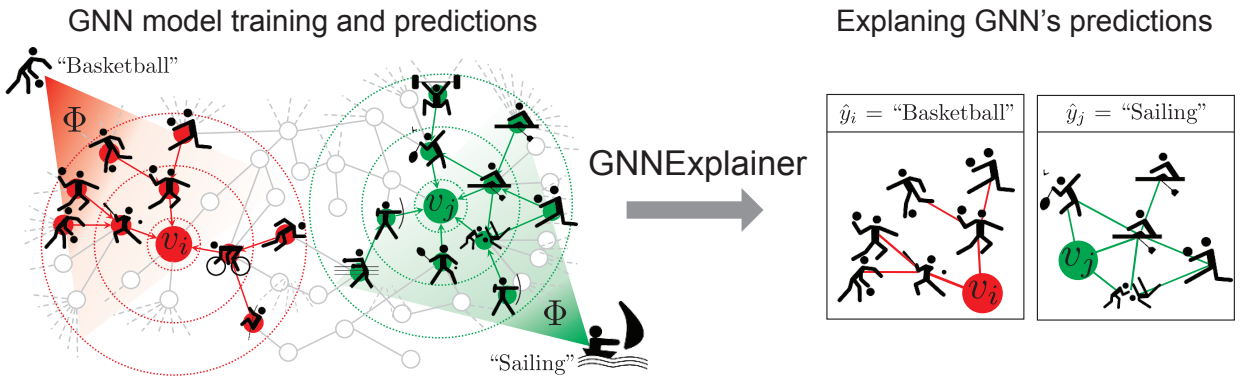


Figure 3.1: GNNEXPLAINER provides interpretable explanations for predictions made by any GNN model on any graph-based machine learning task. Shown is a hypothetical node classification task where a GNN model Φ is trained on a social interaction graph to predict future sport activities. Given a trained GNN Φ and a prediction $\hat{y}_i$ = "Basketball" for person v_i, GNNEXPLAINER generates an explanation by identifying a small subgraph of the input graph together with a small subset of node features (shown on the right) that are most influential for $\hat{y}_i$. Examining explanation for $\hat{y}_i$, we see that many friends in one part of v_i's social circle enjoy ball games, and so the GNN predicts that v_i will like basketball. Similarly, examining explanation for $\hat{y}_j$, we see that v_j's friends and friends of his friends enjoy water and beach sports, and so the GNN predicts $\hat{y}_j$ = "Sailing."

important and useful for several reasons: (i) it can increase trust in the GNN model, (ii) it improves model's transparency in a growing number of decision-critical applications pertaining to fairness, privacy and other safety challenges (Doshi-Velez and Kim, 2017), and (iii) it allows practitioners to get an understanding of the network characteristics, identify and correct systematic patterns of mistakes made by models before deploying them in the real world.

While currently there are no methods for explaining GNNs, recent approaches for explaining other types of neural networks have taken one of two main routes. One line of work locally approximates models with simpler surrogate models, which are then probed for explanations (Lakkaraju et al., 2017; Ribeiro et al., 2016; Schmitz et al., 1999). Other methods carefully examine models for relevant features and find good qualitative interpretations of high level features (Chen et al., 2018c; Erhan et al., 2009; Lundberg and Lee, 2017; Sundararajan et al., 2017) or identify influential input instances (Koh and Liang, 2017; Yeh et al., 2018). However, these approaches fall short in their ability to incorporate relational information, the essence of graphs. Since this aspect is crucial for the success of machine learning on graphs, any explanation of GNN's predictions should leverage rich relational information provided by the graph as well as node features.

Here we propose GNNEXPLAINER, an approach for explaining predictions made by GNNs. GNNEXPLAINER takes a trained GNN and its prediction(s), and it returns an explanation in the form of a small subgraph of the input graph together with a small subset of node features that are most influential for the prediction(s) (Figure 3.1). The approach is model-agnostic and can explain

predictions of any GNN on any machine learning task for graphs, including node classification, link prediction, and graph classification. It handles single- as well as multi-instance explanations. In the case of single-instance explanations, GNNEXPLAINER explains a GNN's prediction for one particular instance (*i.e.*, a node label, a new link, a graph-level label). In the case of multi-instance explanations, GNNEXPLAINER provides an explanation that consistently explains a set of instances (*e.g.*, nodes of a given class).

GNNEXPLAINER specifies an explanation as a rich subgraph of the entire graph the GNN was trained on, such that the subgraph maximizes the mutual information with GNN's prediction(s). This is achieved by formulating a mean field variational approximation and learning a real-valued *graph mask* which selects the important subgraph of the GNN's computation graph. Simultaneously, GNNEXPLAINER also learns a *feature mask* that masks out unimportant node features (Figure 3.1).

We evaluate GNNEXPLAINER on synthetic as well as real-world graphs. Experiments show that GNNEXPLAINER provides consistent and concise explanations of GNN's predictions. On synthetic graphs with planted network motifs, which play a role in determining node labels, we show that GNNEXPLAINER accurately identifies the subgraphs/motifs as well as node features that determine node labels outperforming alternative baseline approaches by up to 43.0% in explanation accuracy. Further, using two real-world datasets we show how GNNEXPLAINER can provide important domain insights by robustly identifying important graph structures and node features that influence a GNN's predictions. Specifically, using molecular graphs and social interaction networks, we show that GNNEXPLAINER can identify important domain-specific graph structures, such as NO_2 chemical groups or ring structures in molecules, and star structures in Reddit threads. Overall, experiments demonstrate that GNNEXPLAINER provides consistent and concise explanations for GNN-based models for different machine learning tasks on graphs.

3.2 Related work on Model Explanation

Although the problem of explaining GNNs is not well-studied, the related problems of interpretability and neural debugging received substantial attention in machine learning. At a high level, we can group those interpretability methods for non-graph neural networks into two main families.

Methods in the first family formulate simple proxy models of full neural networks. This can be done in a model-agnostic way, usually by learning a locally faithful approximation around the prediction, for example through linear models (Ribeiro et al., 2016) or sets of rules, representing

sufficient conditions on the prediction (Augasta and Kathirvalavakumar, 2012; Lakkaraju et al., 2017; Zilke et al., 2016). Methods in the second family identify important aspects of the computation, for example, through feature gradients (Erhan et al., 2009; Zeiler and Fergus, 2014), backpropagation of neurons' contributions to the input features (Chen et al., 2018c; Shrikumar et al., 2017; Sundararajan et al., 2017), and counterfactual reasoning (Kang et al., 2019). However, the saliency maps (Zeiler and Fergus, 2014) produced by these methods have been shown to be misleading in some instances (Adebayo et al., 2018) and prone to issues like gradient saturation (Shrikumar et al., 2017; Sundararajan et al., 2017). These issues are exacerbated on discrete inputs such as graph adjacency matrices since the gradient values can be very large but only on very small intervals. Because of that, such approaches are not suitable for explaining predictions made by neural networks on graphs.

Instead of creating new, inherently interpretable models, post-hoc interpretability methods (Adadi and Berrada, 2018; Fisher et al., 2018; Guidotti et al., 2018; Hooker, 2004; Koh and Liang, 2017; Yeh et al., 2018) consider models as black boxes and then probe them for relevant information. However, no work has been done to leverage relational structures like graphs. The lack of methods for explaining predictions on graph-structured data is problematic, as in many cases, predictions on graphs are induced by a complex combination of nodes and paths of edges between them. For example, in some tasks, an edge is important only when another alternative path exists in the graph to form a cycle, and those two features, only when considered together, can accurately predict node labels (Debnath et al., 1991; Duvenaud et al., 2015a). Their joint contribution thus cannot be modeled as a simple linear combinations of individual contributions.

Finally, recent GNN models augment interpretability via attention mechanisms (Neil et al., 2018; Velickovic et al., 2018; Xie and Grossman, 2018). However, although the learned edge attention values can indicate important graph structure, the values are the same for predictions across all nodes. Thus, this contradicts with many applications where an edge is essential for predicting the label of one node but not the label of another node. Furthermore, these approaches are either limited to specific GNN architectures or cannot explain predictions by jointly considering both graph structure and node feature information.

3.3 Formulating explanations for graph neural networks

Let G denote a graph on edges E and nodes V that are associated with d-dimensional node features $\mathcal{X} = \{x_1, \ldots, x_n\}$, $x_i \in \mathbb{R}^d$. Without loss of generality, we consider the problem of explaining a node classification task. Let f denote a label function on nodes $f : V \mapsto \{1, \ldots, C\}$ that maps

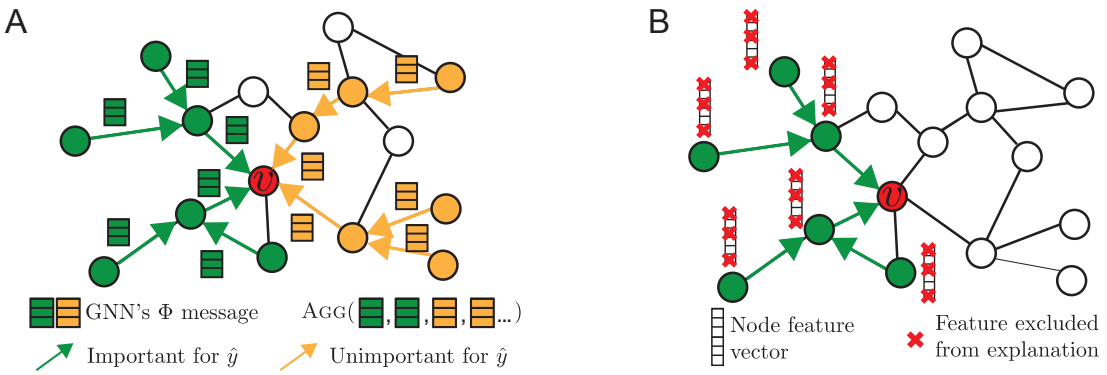


Figure 3.2: **A.** GNN computation graph G_c (green and orange) for making prediction $\hat{y}$ at node v. Some edges in G_c form important neural message-passing pathways (green), which allow useful node information to be propagated across G_c and aggregated at v for prediction, while other edges do not (orange). However, GNN needs to aggregate important as well as unimportant messages to form a prediction at node v, which can dilute the signal accumulated from v's neighborhood. The goal of GNNEXPLAINER is to identify a small set of important features and pathways (green) that are crucial for prediction. **B.** In addition to G_S (green), GNNEXPLAINER identifies what feature dimensions of G_S's nodes are important for prediction by learning a node feature mask.

every node in V to one of C classes. The GNN model Φ is optimized on all nodes in the training set and is then used for prediction, *i.e.*, to approximate f on new nodes.

3.3.1 Background on graph neural networks

At layer l, the update of GNN model Φ involves three key computations (Battaglia et al., 2018; Zhang et al., 2018c; Zhou et al., 2018). (1) First, the model computes neural messages between every pair of nodes. The message for node pair (v_i, v_j) is a function MSG of v_i's and v_j's representations $\mathbf{h}_i^{l-1}$ and $\mathbf{h}_j^{l-1}$ in the previous layer and of the relation r_{ij} between the nodes: $m_{ij}^l = \text{MSG}(\mathbf{h}_i^{l-1}, \mathbf{h}_j^{l-1}, r_{ij})$. (2) Second, for each node v_i, GNN aggregates messages from v_i's neighborhood $\mathcal{N}_{v_i}$ and calculates an aggregated message M_i via an aggregation method AGG (Hamilton et al., 2017a; Xu et al., 2019): $M_i^l = \text{AGG}(\{m_{ij}^l | v_j \in \mathcal{N}_{v_i}\})$, where $\mathcal{N}_{v_i}$ is neighborhood of node v_i whose definition depends on a particular GNN variant. (3) Finally, GNN takes the aggregated message M_i^l along with v_i's representation $\mathbf{h}_i^{l-1}$ from the previous layer, and it non-linearly transforms them to obtain v_i's representation $\mathbf{h}_i^l$ at layer l: $\mathbf{h}_i^l = \text{UPDATE}(M_i^l, \mathbf{h}_i^{l-1})$. The final embedding for node v_i after L layers of computation is $\mathbf{z}_i = \mathbf{h}_i^L$. Our GNNEXPLAINER provides explanations for any GNN that can be formulated in terms of MSG, AGG, and UPDATE computations.

3.3.2 GNNEXPLAINER: Problem formulation

Our key insight is the observation that the computation graph of node v, which is defined by the GNN's neighborhood-based aggregation (Figure 3.2), fully determines all the information the GNN uses to generate prediction $\hat{y}$ at node v. In particular, v's computation graph tells the GNN how to generate v's embedding $\mathbf{z}$. Let us denote that computation graph by $G_c(v)$, the associated binary adjacency matrix by $A_c(v) \in \{0,1\}^{n \times n}$, and the associated feature set by $X_c(v) = \{x_j | v_j \in G_c(v)\}$. The GNN model Φ learns a conditional distribution $P_\Phi(Y|G_c, X_c)$, where Y is a random variable representing labels $\{1, \ldots, C\}$, indicating the probability of nodes belonging to each of C classes.

A GNN's prediction is given by $\hat{y} = \Phi(G_c(v), X_c(v))$, meaning that it is fully determined by the model Φ, graph structural information $G_c(v)$, and node feature information $X_c(v)$. In effect, this observation implies that we only need to consider graph structure $G_c(v)$ and node features $X_c(v)$ to explain $\hat{y}$ (Figure 3.2A). Formally, GNNEXPLAINER generates explanation for prediction $\hat{y}$ as (G_S, X_S^F), where G_S is a small subgraph of the computation graph and X_S^F is a small subset of node features (*i.e.*, $X_S^F = \{x_j^F | v_j \in G_S\}$) that are most important for explaining $\hat{y}$ (Figure 3.2B).

3.4 GNNEXPLAINER

Next we describe our approach GNNEXPLAINER. Given a trained GNN model Φ and a prediction on nodes, edges or graphs, the GNNEXPLAINER will generate an explanation by identifying a subgraph of the computation graph and a subset of node features that are most influential for the model Φ's prediction. In the case of explaining a set of predictions, GNNEXPLAINER will aggregate individual explanations in the set and automatically summarize it with a prototype. We conclude this section with a discussion on how GNNEXPLAINER can be used for any machine learning task on graphs, including link prediction and graph classification (Section 3.4.3).

3.4.1 Explanation through optimization

Given a node v, our goal is to identify a subgraph $G_S \subseteq G_c$ and the associated features $X_S = \{x_j | v_j \in G_S\}$ that are important for the GNN's prediction $\hat{y}$. For now, we assume that X_S is a small subset of d-dimensional node features; we will later discuss how to automatically determine which dimensions of node features need to be included in explanations (Section 3.4.2). We formalize the notion of importance using mutual information MI and formulate the GNNEXPLAINER as the

following optimization framework:

$$\max_{G_S} MI\left(Y, (G_S, X_S)\right) = H(Y) - H(Y|G = G_S, X = X_S). \tag{3.1}$$

For node v, MI quantifies the change in the probability of prediction $\hat{y} = \Phi(G_c, X_c)$ when v's computation graph is limited to explanation subgraph G_S and its node features are limited to X_S.

For example, consider the situation where $v_j \in G_c(v_i)$, $v_j \neq v_i$. Then, if removing v_j from $G_c(v_i)$ strongly decreases the probability of prediction $\hat{y}_i$, the node v_j is a good counterfactual explanation for the prediction at v_i. Similarly, consider the situation where $(v_j, v_k) \in G_c(v_i)$, $v_j, v_k \neq v_i$. Then, if removing an edge between v_j and v_k strongly decreases the probability of prediction $\hat{y}_i$ then the absence of that edge is a good counterfactual explanation for the prediction at v_i.

Examining Eq. (3.1), we see that the entropy term $H(Y)$ is constant because Φ is fixed for a trained GNN. As a result, maximizing mutual information between the predicted label distribution Y and explanation (G_S, X_S) is equivalent to minimizing conditional entropy $H(Y|G = G_S, X = X_S)$, which can be expressed as follows:

$$H(Y|G=G_S, X=X_S) = -\mathbb{E}_{Y|G_S,X_S}\left[\log P_\Phi(Y|G=G_S, X=X_S)\right]. \tag{3.2}$$

Explanation for prediction $\hat{y}$ is thus a subgraph G_S that minimizes uncertainty of Φ when the GNN computation is limited to G_S. In effect, G_S maximizes probability of $\hat{y}$ (Figure 3.2). To obtain a compact explanation, we impose a constraint on G_S's size as: $|G_S| \leq K_M$, so that G_S has at most K_M nodes. In effect, this implies that GNNEXPLAINER aims to denoise G_c by taking K_M edges that give the highest mutual information with the prediction.

GNNEXPLAINER's optimization framework. Direct optimization of GNNEXPLAINER's objective is not tractable as G_c has exponentially many subgraphs G_S that are candidate explanations for $\hat{y}$. We thus consider a fractional adjacency matrix[1] for subgraphs G_S, *i.e.*, $A_S \in [0, 1]^{n \times n}$, and enforce the subgraph constraint as: $A_S[j, k] \leq A_c[j, k]$ for all j, k. This continuous relaxation can be interpreted as a variational approximation of distribution of subgraphs of G_c. In particular, if we treat $G_S \sim \mathcal{G}$ as a random graph variable, the objective in Eq. (3.2) becomes:

$$\min_{\mathcal{G}} \mathbb{E}_{G_S \sim \mathcal{G}} H(Y|G = G_S, X = X_S), \tag{3.3}$$

[1]For typed edges, we define $G_S \in [0, 1]^{C_e \times n \times n}$ where C_e is the number of edge types.

With convexity assumption, Jensen's inequality gives the following upper bound:

$$\min_{\mathcal{G}} H(Y|G = \mathbb{E}_{\mathcal{G}}[G_S], X = X_S). \tag{3.4}$$

In practice, due to the complexity of neural networks, the convexity assumption does not hold. However, experimentally, we found that minimizing this objective with regularization often leads to a local minimum corresponding to high-quality explanations.

To tractably estimate $\mathbb{E}_{\mathcal{G}}$, we use mean-field variational approximation and decompose $\mathcal{G}$ into a multivariate Bernoulli distribution as: $P_{\mathcal{G}}(G_S) = \prod_{(j,k)\in G_c} A_S[j,k]$. This allows us to estimate the expectation with respect to the mean-field approximation, thereby obtaining A_S in which (j,k)-th entry represents the expectation on whether edge (v_j, v_k) exists. We observed empirically that this approximation together with a regularizer for promoting discreteness (Ying et al., 2018c) converges to good local minima despite the non-convexity of GNNs. The conditional entropy in Equation 3.4 can be optimized by replacing the $\mathbb{E}_{\mathcal{G}}[G_S]$ to be optimized by a masking of the computation graph of adjacency matrix, $A_c \odot \sigma(M)$, where $M \in \mathbb{R}^{n\times n}$ denotes the mask that we need to learn, $\odot$ denotes element-wise multiplication, and σ denotes the sigmoid that maps the mask to $[0,1]^{n\times n}$.

In some applications, instead of finding an explanation in terms of model's confidence, the users care more about "why does the trained model predict a certain class label", or "how to make the trained model predict a desired class label". We can modify the conditional entropy objective in Equation 3.4 with a cross entropy objective between the label class and the model prediction[2]. To answer these queries, a computationally efficient version of GNNEXPLAINER's objective, which we optimize using gradient descent, is as follows:

$$\min_{M} -\sum_{c=1}^{C} \mathbb{1}[y = c] \log P_{\Phi}(Y = y|G = A_c \odot \sigma(M), X = X_c), \tag{3.5}$$

The masking approach is also found in Neural Relational Inference (Kipf et al., 2018), albeit with different motivation and objective. Lastly, we compute the element-wise multiplication of $\sigma(M)$ and A_c and remove low values in M through thresholding to arrive at the explanation G_S for the GNN model's prediction $\hat{y}$ at node v.

[2]The label class is the predicted label class by the GNN model to be explained, when answering "why does the trained model predict a certain class label". "how to make the trained model predict a desired class label" can be answered by using the ground-truth label class.

3.4.2 Joint learning of graph structural and node feature information

To identify what node features are most important for prediction $\hat{y}$, GNNEXPLAINER learns a feature selector F for nodes in explanation G_S. Instead of defining X_S to consists of all node features, *i.e.*, $X_S = \{x_j | v_j \in G_S\}$, GNNEXPLAINER considers X_S^F as a subset of features of nodes in G_S, which are defined through a binary feature selector $F \in \{0,1\}^d$ (Figure 3.2B):

$$X_S^F = \{x_j^F | v_j \in G_S\}, \quad x_j^F = [x_{j,t_1}, \ldots, x_{j,t_k}] \text{ for } F_{t_i} = 1, \tag{3.6}$$

where x_j^F has node features that are not masked out by F. Explanation (G_S, X_S) is then jointly optimized for maximizing the mutual information objective:

$$\max_{G_S,F} MI\left(Y, (G_S, F)\right) = H(Y) - H(Y|G = G_S, X = X_S^F), \tag{3.7}$$

which represents a modified objective function from Eq. (3.1) that considers structural and node feature information to generate an explanation for prediction $\hat{y}$.

Learning binary feature selector F. We specify X_S^F as $X_S \odot F$, where F acts as a feature mask that we need to learn. Intuitively, if a particular feature is not important, the corresponding weights in GNN's weight matrix take values close to zero. In effect, this implies that masking the feature out does not decrease predicted probability for $\hat{y}$. Conversely, if the feature is important then masking it out would decrease predicted probability. However, in some cases this approach ignores features that are important for prediction but take values close to zero. To address this issue we marginalize over all feature subsets and use a Monte Carlo estimate to sample from empirical marginal distribution for nodes in X_S during training (Zintgraf et al., 2017). Further, we use a reparametrization trick (Kingma and Welling, 2013) to backpropagate gradients in Eq. (3.7) to the feature mask F. In particular, to backpropagate through a d-dimensional random variable X we reparametrize X as: $X = Z + (X_S - Z) \odot F$ s.t. $\sum_j F_j \leq K_F$, where Z is a d-dimensional random variable sampled from the empirical distribution and K_F is a parameter representing the maximum number of features to be kept in the explanation.

Integrating additional constraints into explanations. To impose further properties on the explanation we can extend GNNEXPLAINER's objective function in Eq. (3.7) with regularization terms. For example, we use element-wise entropy to encourage structural and node feature masks to be discrete. Further, GNNEXPLAINER can encode domain-specific constraints through techniques like Lagrange multiplier of constraints or additional regularization terms. We include a number of regularization terms to produce explanations with desired properties. We penalize large size of the explanation by adding the sum of all elements of the mask paramters as the regularization term.

Finally, it is important to note that each explanation must be a valid computation graph. In particular, explanation (G_S, X_S) needs to allow GNN's neural messages to flow towards node v such that GNN can make prediction $\hat{y}$. Importantly, GNNEXPLAINER automatically provides explanations that represent valid computation graphs because it optimizes structural masks across entire computation graphs. Even if a disconnected edge is important for neural message-passing, it will not be selected for explanation as it cannot influence GNN's prediction. In effect, this implies that the explanation G_S tends to be a small connected subgraph.

3.4.3 GNNEXPLAINER model extensions

Any machine learning task on graphs. In addition to explaining node classification, GNNEXPLAINER provides explanations for link prediction and graph classification with no change to its optimization algorithm. When predicting a link (v_j, v_k), GNNEXPLAINER learns two masks $X_S(v_j)$ and $X_S(v_k)$ for both endpoints of the link. When classifying a graph, the adjacency matrix in Eq. (3.5) is the union of adjacency matrices for all nodes in the graph whose label we want to explain. However, note that in graph classification, unlike node classification, due to the aggregation of node embeddings, it is no longer true that the explanation G_S is necessarily a connected subgraph. Depending on application, in some scenarios such as chemistry where explanation is a functional group and should be connected, one can extract the largest connected component as the explanation.

Any GNN model. Modern GNNs are based on message passing architectures on the input graph. The message passing computation graphs can be composed in many different ways and GNNEXPLAINER can account for all of them. Thus, GNNEXPLAINER can be applied to: Graph Convolutional Networks (Kipf and Welling, 2016a), Gated Graph Sequence Neural Networks (Li et al., 2015), Jumping Knowledge Networks (Xu et al., 2018a), Attention Networks (Velickovic et al., 2018), Graph Networks (Battaglia et al., 2018), GNNs with various node aggregation schemes (Chen et al., 2018d,b; Huang et al., 2018; Hamilton et al., 2017a; Ying et al., 2018c,a; Xu et al., 2019), Line-Graph NNs (Chen et al., 2019), position-aware GNN (You et al., 2019), and many other GNN architectures.

Computational complexity. The number of parameters in GNNEXPLAINER's optimization depends on the size of computation graph G_c for node v whose prediction we aim to explain. In particular, $G_c(v)$'s adjacency matrix $A_c(v)$ is equal to the size of the mask M, which needs to be learned by GNNEXPLAINER. However, since computation graphs are typically relatively small, compared to the size of exhaustive L-hop neighborhoods (*e.g.*, 2-3 hop neighborhoods (Kipf and

Welling, 2016a), sampling-based neighborhoods (Ying et al., 2018a), neighborhoods with attention (Velickovic et al., 2018)), GNNEXPLAINER can effectively generate explanations even when input graphs are large.

3.5 Experiments

We begin by describing the graphs, alternative baseline approaches, and experimental setup. We then present experiments on explaining GNNs for node classification and graph classification tasks. Our qualitative and quantitative analysis demonstrates that GNNEXPLAINER is accurate and effective in identifying explanations, both in terms of graph structure and node features.

Synthetic datasets. We construct four kinds of node classification datasets (Table 1). (1) In BA-SHAPES, we start with a base Barabási-Albert (BA) graph on 300 nodes and a set of 80 five-node "house"-structured network motifs, which are attached to randomly selected nodes of the base graph. The resulting graph is further perturbed by adding $0.1N$ random edges. Nodes are assigned to 4 classes based on their structural roles. In a house-structured motif, there are 3 types of roles: the top, middle and bottom node of the house. Therefore there are 4 different classes, corresponding to nodes at the top, middle, bottom of houses, and nodes that do not belong to a house. (2) BA-COMMUNITY dataset is a union of two BA-SHAPES graphs. Nodes have normally distributed feature vectors and are assigned to one of 8 classes based on their structural roles and community memberships. (3) In TREE-CYCLES, we start with a base 8-level balanced binary tree and 80 six-node cycle motifs, which are attached to random nodes of the base graph. (4) TREE-GRID is the same as TREE-CYCLES except that 3-by-3 grid motifs are attached to the base tree graph in place of cycle motifs.

Real-world datasets. We consider two graph classification datasets: (1) MUTAG is a dataset of $4{,}337$ molecule graphs labeled according to their mutagenic effect on the Gram-negative bacterium *S. typhimurium* (Debnath et al., 1991). (2) REDDIT-BINARY is a dataset of $2{,}000$ graphs, each representing an online discussion thread on Reddit. In each graph, nodes are users participating in a thread, and edges indicate that one user replied to another user's comment. Graphs are labeled according to the type of user interactions in the thread: *r/IAmA* and *r/AskReddit* contain Question-Answer interactions, while *r/TrollXChromosomes* and *r/atheism* contain Online-Discussion interactions (Yanardag and Vishwanathan, 2015b).

Alternative baseline approaches. Many explainability methods cannot be directly applied to graphs (Section 9.2). Nevertheless, we here consider the following alternative approaches that can provide insights into predictions made by GNNs: (1) GRAD is a gradient-based method. We

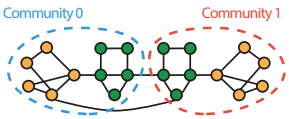
	BA-Shapes	BA-Community	Tree-Cycles	Tree-Grid
Base				
Motif				
Node Features	None	$\mathcal{N}(\mu_l, \sigma_l)$ where l = community ID	None	None
Explanation content	Graph structure	Graph structure Node feature information	Graph structure	Graph structure
Explanation accuracy				
Att	0.815	0.739	0.824	0.612
Grad	0.882	0.750	0.905	0.667
GNNExplainer	**0.925**	**0.836**	**0.948**	**0.875**

Table 3.1: Illustration of synthetic datasets (refer to "Synthetic datasets" for details) together with performance evaluation of GNNEXPLAINER and alternative baseline explainability approaches.

compute gradient of the GNN's loss function with respect to the adjacency matrix and the associated node features, similar to a saliency map approach. (2) ATT is a graph attention GNN (GAT) (Velickovic et al., 2018) that learns attention weights for edges in the computation graph, which we use as a proxy measure of edge importance. While ATT does consider graph structure, it does not explain using node features and can only explain GAT models. Furthermore, in ATT it is not obvious which attention weights need to be used for edge importance, since a 1-hop neighbor of a node can also be a 2-hop neighbor of the same node due to cycles. Each edge's importance is thus computed as the average attention weight across all layers.

Setup and implementation details. For each dataset, we first train a single GNN for each dataset, and use GRAD and GNNEXPLAINER to explain the predictions made by the GNN. Note that the ATT baseline requires using a graph attention architecture like GAT (Velickovic et al., 2018). We thus train a separate GAT model on the same dataset and use the learned edge attention weights for explanation. Hyperparameters K_M, K_F control the size of subgraph and feature explanations respectively, which is informed by prior knowledge about the dataset. For synthetic datasets, we set K_M to be the size of ground truth. On real-world datasets, we set $K_M = 10$. We set $K_F = 5$ for all datasets. We further fix our weight regularization hyperparameters across all node and graph classification experiments. We refer readers to the Appendix for more training details (Code and datasets are available at https://github.com/RexYing/gnn-model-explainer).

Results. We investigate questions: Does GNNEXPLAINER provide sensible explanations? How do explanations compare to the ground-truth knowledge? How does GNNEXPLAINER perform on various graph-based prediction tasks? Can it explain predictions made by different GNNs?

1) Quantitative analyses. Results on node classification datasets are shown in Table 3.1. We have ground-truth explanations for synthetic datasets and we use them to calculate explanation

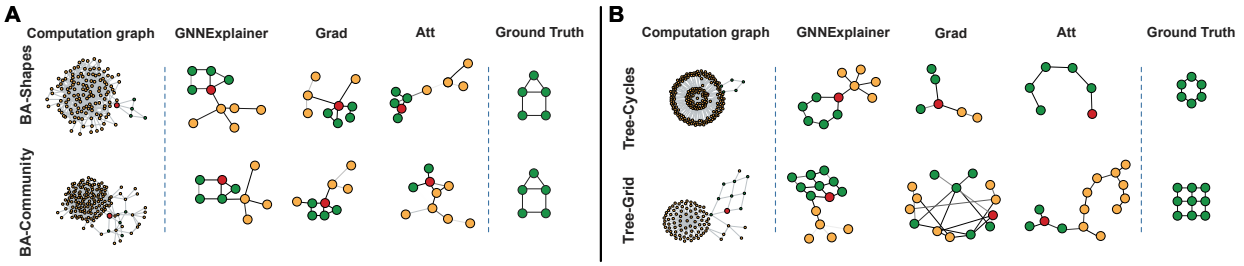


Figure 3.3: Evaluation of single-instance explanations. **A-B.** Shown are exemplar explanation subgraphs for node classification task on four synthetic datasets. Each method provides explanation for the red node's prediction.

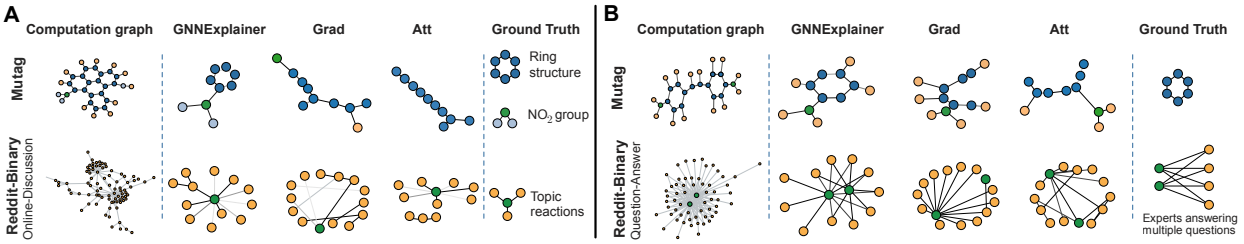


Figure 3.4: Evaluation of single-instance explanations. **A-B.** Shown are exemplar explanation subgraphs for graph classification task on two datasets, MUTAG and REDDIT-BINARY.

accuracy for all explanation methods. Specifically, we formalize the explanation problem as a binary classification task, where edges in the ground-truth explanation are treated as labels and importance weights given by explainability method are viewed as prediction scores. A better explainability method predicts high scores for edges that are in the ground-truth explanation, and thus achieves higher explanation accuracy. Results show that GNNEXPLAINER outperforms alternative approaches by 17.1% on average. Further, GNNEXPLAINER achieves up to 43.0% higher accuracy on the hardest TREE-GRID dataset.

2) Qualitative analyses. Results are shown in Figures 3.3–3.5. In a topology-based prediction task with no node features, *e.g.* BA-SHAPES and TREE-CYCLES, GNNEXPLAINER correctly identifies network motifs that explain node labels, *i.e.*structural labels (Figure 3.3). As illustrated in the figures, house, cycle and tree motifs are identified by GNNEXPLAINER but not by baseline methods. In Figure 3.4, we investigate explanations for graph classification task. In MUTAG example, colors indicate node features, which represent atoms (hydrogen H, carbon C, *etc*). GNNEXPLAINER correctly identifies carbon ring as well as chemical groups NH_2 and NO_2, which are known to be mutagenic (Debnath et al., 1991).

Further, in REDDIT-BINARY example, we see that Question-Answer graphs (2nd row in Figure 3.4B) have 2-3 high degree nodes that simultaneously connect to many low degree nodes,

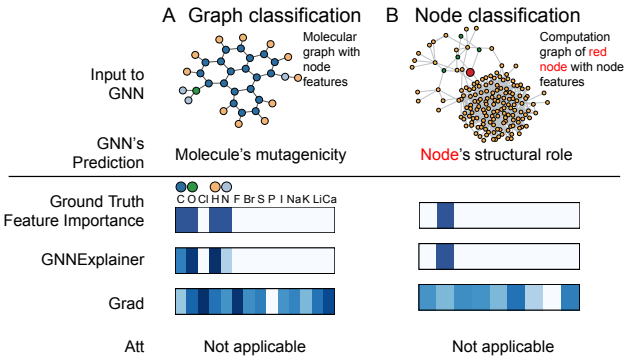


Figure 3.5: Visualization of features that are important for a GNN's prediction. **A.** Shown is a representative molecular graph from MUTAG dataset (top). Importance of the associated graph features is visualized with a heatmap (bottom). In contrast with baselines, GNNEXPLAINER correctly identifies features that are important for predicting the molecule's mutagenicity, *i.e.*C, O, H, and N atoms. **B.** Shown is a computation graph of a red node from BA-COMMUNITY dataset (top). Again, GNNEXPLAINER successfully identifies the node feature that is important for predicting the structural role of the node but baseline methods fail.

which makes sense because in QA threads on Reddit we typically have 2-3 experts who all answer many different questions (Kumar et al., 2018). Conversely, we observe that discussion patterns commonly exhibit tree-like patterns (2nd row in Figure 3.4A), since a thread on Reddit is usually a reaction to a single topic (Kumar et al., 2018). On the other hand, GRAD and ATT methods give incorrect or incomplete explanations. For example, both baseline methods miss cycle motifs in MUTAG dataset and more complex grid motifs in TREE-GRID dataset. Furthermore, although edge attention weights in ATT can be interpreted as importance scores for message passing, the weights are shared across all nodes in input the graph, and as such ATT fails to provide high quality single-instance explanations.

An essential criterion for explanations is that they must be interpretable, *i.e.*, provide a qualitative understanding of the relationship between the input nodes and the prediction. Such a requirement implies that explanations should be easy to understand while remaining exhaustive. This means that a GNN explainer should take into account both the structure of the underlying graph as well as the associated features when they are available. Figure 3.5 shows results of an experiment in which GNNEXPLAINER jointly considers structural information as well as information from a small number of feature dimensions[3]. While GNNEXPLAINER indeed highlights a compact feature representation in Figure 3.5, gradient-based approaches struggle to cope with the added noise, giving high importance scores to irrelevant feature dimensions.

[3]Feature explanations are shown for the two datasets with node features, *i.e.*, MUTAG and BA-COMMUNITY.

Part II

Expressive Graph Neural Networks: Architectures and Embeddings

Chapter 4

Hierarchical Differentiable Pooling

Part II of this thesis demonstrates my contributions in architectural improvement of GNNs for a variety of tasks and graph dataset properties. The goal is to extend the GraphSAGE framework to achieve even higher expressive power and performance on graph-structured data. I leverage the inductive biases of the properties of dataset, such as the graph clustering structure, hierarchical tree-like structure. Furthermore, I explore the combination of attention mechanism between nodes in graph, as well as multi-hop diffusion on graphs

In this chapter, I demonstrate a new technique of learnable hierarchical pooling, which can be applied to any GNNs under the GraphSAGE framework, to aggregate node vector representations into a graph-level vector representation, which better captures the information of the entire graph structure and the associated node features. Such technique is critical for many graph-level tasks, including classifying molecules, proteins and ego-networks, in the domain of chemistry, medicine, biology and social network analysis.

4.1 Introduction

In recent years there has been a surge of interest in developing graph neural networks (GNNs)—general deep learning architectures that can operate over graph structured data, such as social network data (Hamilton et al., 2017c; Kipf and Welling, 2017) or graph-based representations of molecules (Dai et al., 2016a; Duvenaud et al., 2015b; Gilmer et al., 2017). The general approach with GNNs is to view the underlying graph as a computation graph and learn neural network primitives that generate individual node embeddings by passing, transforming, and aggregating node feature information across the graph (Gilmer et al., 2017; Hamilton et al., 2017c). The generated node embeddings can then be used as input to any differentiable prediction layer, e.g., for node

classification (Hamilton et al., 2017c) or link prediction (Schütt et al., 2017), and the whole model can be trained in an end-to-end fashion.

However, a major limitation of current GNN architectures is that they are inherently *flat* as they only propagate information across the edges of the graph and are unable to infer and aggregate the information in a *hierarchical* way. For example, in order to successfully encode the graph structure of organic molecules, one would ideally want to encode the local molecular structure (e.g., individual atoms and their direct bonds) as well as the coarse-grained structure of the molecular graph (e.g., groups of atoms and bonds representing functional units in a molecule). This lack of hierarchical structure is especially problematic for the task of graph classification, where the goal is to predict the label associated with an *entire graph*. When applying GNNs to graph classification, the standard approach is to generate embeddings for all the nodes in the graph and then to *globally pool* all these node embeddings together, e.g., using a simple summation or neural network that operates over sets (Dai et al., 2016a; Duvenaud et al., 2015b; Gilmer et al., 2017; Li et al., 2016). This global pooling approach ignores any hierarchical structure that might be present in the graph, and it prevents researchers from building effective GNN models for predictive tasks over entire graphs.

Here we propose DIFFPOOL, a differentiable graph pooling module that can be adapted to various graph neural network architectures in an hierarchical and end-to-end fashion (Figure 4.1). DIFFPOOL allows for developing deeper GNN models that can learn to operate on hierarchical representations of a graph. We develop a graph analogue of the spatial pooling operation in CNNs (Krizhevsky et al., 2012), which allows for deep CNN architectures to iteratively operate on coarser and coarser representations of an image. The challenge in the GNN setting—compared to standard CNNs—is that graphs contain no natural notion of spatial locality, i.e., one cannot simply pool together all nodes in a "$m \times m$ patch" on a graph, because the complex topological structure of graphs precludes any straightforward, deterministic definition of a "patch". Moreover, unlike image data, graph data sets often contain graphs with varying numbers of nodes and edges, which makes defining a general graph pooling operator even more challenging.

In order to solve the above challenges, we require a model that learns how to cluster together nodes to build a hierarchical multi-layer scaffold on top of the underlying graph. Our approach DIFFPOOL learns a differentiable soft assignment at each layer of a deep GNN, mapping nodes to a set of clusters based on their learned embeddings. In this framework, we generate deep GNNs by "stacking" GNN layers in a hierarchical fashion (Figure 4.1): the input nodes at the layer l GNN module correspond to the clusters learned at the layer $l-1$ GNN module. Thus, each layer of

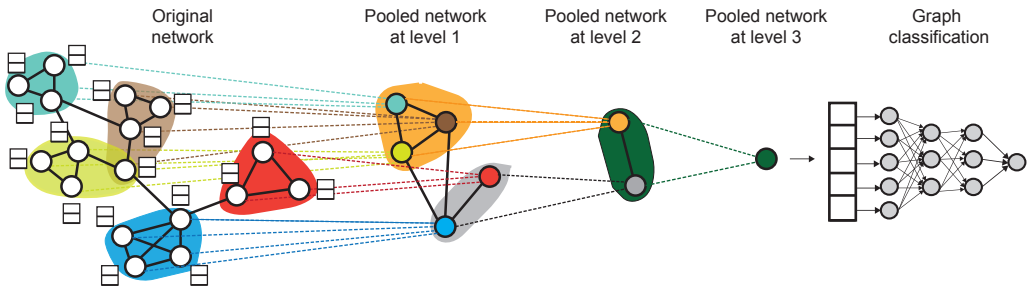


Figure 4.1: High-level illustration of our proposed method DIFFPOOL. At each hierarchical layer, we run a GNN model to obtain embeddings of nodes. We then use these learned embeddings to cluster nodes together and run another GNN layer on this coarsened graph. This whole process is repeated for L layers and we use the final output representation to classify the graph.

DIFFPOOL coarsens the input graph more and more, and DIFFPOOL is able to generate a hierarchical representation of any input graph after training. We show that DIFFPOOL can be combined with various GNN approaches, resulting in an average 7% gain in accuracy and a new state of the art on four out of five benchmark graph classification tasks. Finally, we show that DIFFPOOL can learn interpretable hierarchical clusters that correspond to well-defined communities in the input graphs.

4.2 Related Work

Our work builds upon a rich line of recent research on graph neural networks and graph classification.

General graph neural networks. A wide variety of graph neural network (GNN) models have been proposed in recent years, including methods inspired by convolutional neural networks (Bruna et al., 2014a; Defferrard et al., 2016a; Duvenaud et al., 2015b; Hamilton et al., 2017c; Kipf and Welling, 2017; Lei et al., 2017; Niepert et al., 2016), recurrent neural networks (Li et al., 2016), recursive neural networks (Bianchini et al., 2001; Scarselli et al., 2009a) and loopy belief propagation (Dai et al., 2016a). Most of these approaches fit within the framework of "neural message passing" proposed by Gilmer *et al.* (Gilmer et al., 2017). In the message passing framework, a GNN is viewed as a message passing algorithm where node representations are iteratively computed from the features of their neighbor nodes using a differentiable aggregation function. Hamilton *et al.* (Hamilton et al., 2017b) provides a conceptual review of recent advancements in this area, and Bronstein *et al.* (Bronstein et al., 2017) outlines connections to spectral graph convolutions.

Graph classification with graph neural networks. GNNs have been applied to a wide variety of tasks, including node classification (Hamilton et al., 2017c; Kipf and Welling, 2017), link prediction (Schlichtkrull et al., 2018b), graph classification (Dai et al., 2016a; Duvenaud et al., 2015b; Zhang et al., 2018b), and chemoinformatics (Merkwirth and Lengauer, 2005; Lusci et al., 2013; Fout et al., 2017; Jin et al., 2017; Schütt et al., 2017). In the context of graph classification—the task that we study here—a major challenge in applying GNNs is going from node embeddings, which are the output of GNNs, to a representation of the entire graph. Common approaches to this problem include simply summing up or averaging all the node embeddings in a final layer (Duvenaud et al., 2015b), introducing a "virtual node" that is connected to all the nodes in the graph (Li et al., 2016), or aggregating the node embeddings using a deep learning architecture that operates over sets (Gilmer et al., 2017). However, all of these methods have the limitation that they do not learn hierarchical representations (i.e., all the node embeddings are globally pooled together in a single layer), and thus are unable to capture the natural structures of many real-world graphs. Some recent approaches have also proposed applying CNN architectures to the concatenation of all the node embeddings (Niepert et al., 2016; Zhang et al., 2018b), but this requires a specifying (or learning) a canonical ordering over nodes, which is in general very difficult and equivalent to solving graph isomorphism.

Lastly, there are some recent works that learn hierarchical graph representations by combining GNNs with deterministic graph clustering algorithms (Defferrard et al., 2016a; Simonovsky and Komodakis, 2017; Fey et al., 2018), following a two-stage approach. However, unlike these previous approaches, we seek to *learn* the hierarchical structure in an end-to-end fashion, rather than relying on a deterministic graph clustering subroutine.

4.3 Proposed Method

The key idea of DIFFPOOL is that it enables the construction of deep, multi-layer GNN models by providing a differentiable module to hierarchically pool graph nodes. In this section, we outline the DIFFPOOL module and show how it is applied in a deep GNN architecture.

4.3.1 Preliminaries

We represent a graph G as (A, F), where $A \in \{0, 1\}^{n \times n}$ is the adjacency matrix, and $F \in \mathbb{R}^{n \times d}$ is the node feature matrix assuming each node has d features.[1] Given a set of labeled graphs

[1] We do not consider edge features, although one can easily extend the algorithm to support edge features using techniques introduced in (Simonovsky and Komodakis, 2017).

$\mathcal{D} = \{(G_1, y_1), (G_2, y_2), ...\}$ where $y_i \in \mathcal{Y}$ is the label corresponding to graph $G_i \in \mathcal{G}$, the goal of graph classification is to learn a mapping $f : \mathcal{G} \rightarrow \mathcal{Y}$ that maps graphs to the set of labels. The challenge—compared to standard supervised machine learning setup—is that we need a way to extract useful feature vectors from these input graphs. That is, in order to apply standard machine learning methods for classification, e.g., neural networks, we need a procedure to convert each graph to an finite dimensional vector in $\mathbb{R}^D$.

Graph neural networks. In this work, we build upon graph neural networks in order to learn useful representations for graph classification in an end-to-end fashion. In particular, we consider GNNs that employ the following general "message-passing" architecture:

$$H^{(k)} = M(A, H^{(k-1)}; \theta^{(k)}), \tag{4.1}$$

where $H^{(k)} \in \mathbb{R}^{n \times d}$ are the node embeddings (i.e., "messages") computed after k steps of the GNN and M is the message propagation function, which depends on the adjacency matrix, trainable parameters $\theta^{(k)}$, and the node embeddings $H^{(k-1)}$ generated from the previous message-passing step.[2] The input node embeddings $H^{(0)}$ at the initial message-passing iteration $(k = 1)$, are initialized using the node features on the graph, $H^{(0)} = F$.

There are many possible implementations of the propagation function M (Gilmer et al., 2017; Hamilton et al., 2017c). For example, one popular variant of GNNs—Kipf's *et al*. (Kipf and Welling, 2017) Graph Convolutional Networks (GCNs)—implements M using a combination of linear transformations and ReLU non-linearities:

$$H^{(k)} = M(A, H^{(k-1)}; W^{(k)}) = \text{ReLU}(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(k-1)} W^{(k-1)}), \tag{4.2}$$

where $\tilde{A} = A + I$, $\tilde{D} = \sum_j \tilde{A}_{ij}$ and $W^{(k)} \in \mathbb{R}^{d \times d}$ is a trainable weight matrix. The differentiable pooling model we propose can be applied to any GNN model implementing Equation (4.1), and is agnostic with regards to the specifics of how M is implemented.

A full GNN module will run K iterations of Equation (4.1) to generate the final output node embeddings $Z = H^{(K)} \in \mathbb{R}^{n \times d}$, where K is usually in the range 2-6. For simplicity, in the following sections we will abstract away the internal structure of the GNNs and use $Z = \text{GNN}(A, X)$ to denote an arbitrary GNN module implementing K iterations of message passing according to some adjacency matrix A and initial input node features X.

Stacking GNNs and pooling layers. GNNs implementing Equation (4.1) are inherently flat, as

[2]For notational convenience, we assume that the embedding dimension is d for all $H^{(k)}$; however, in general this restriction is not necessary.

they only propagate information across edges of a graph. The goal of this work is to define a general, end-to-end differentiable strategy that allows one to *stack* multiple GNN modules in a hierarchical fashion. Formally, given $Z = \text{GNN}(A, X)$, the output of a GNN module, and a graph adjacency matrix $A \in \mathbb{R}^{n \times n}$, we seek to define a strategy to output a new coarsened graph containing $m < n$ nodes, with weighted adjacency matrix $A^{'} \in \mathbb{R}^{m \times m}$ and node embeddings $Z^{'} \in \mathbb{R}^{m \times d}$. This new coarsened graph can then be used as input to another GNN layer, and this whole process can be repeated L times, generating a model with L GNN layers that operate on a series of coarser and coarser versions of the input graph (Figure 4.1). Thus, our goal is to learn how to cluster or pool together nodes using the output of a GNN, so that we can use this coarsened graph as input to another GNN layer. What makes designing such a pooling layer for GNNs especially challenging—compared to the usual graph coarsening task—is that our goal is not to simply cluster the nodes in one graph, but to provide a general recipe to hierarchically pool nodes across a broad set of input graphs. That is, we need our model to learn a pooling strategy that will generalize across graphs with different nodes, edges, and that can adapt to the various graph structures during inference.

4.3.2 Differentiable Pooling via Learned Assignments

Our proposed approach, DIFFPOOL, addresses the above challenges by learning a cluster assignment matrix over the nodes using the output of a GNN model. The key intuition is that we stack L GNN modules and learn to assign nodes to clusters at layer l in an end-to-end fashion, using embeddings generated from a GNN at layer $l - 1$. Thus, we are using GNNs to both extract node embeddings that are useful for graph classification, as well to extract node embeddings that are useful for hierarchical pooling. Using this construction, the GNNs in DIFFPOOL learn to encode a general pooling strategy that is useful for a large set of training graphs. We first describe how the DIFFPOOL module pools nodes at each layer given an assignment matrix; following this, we discuss how we generate the assignment matrix using a GNN architecture.

Pooling with an assignment matrix. We denote the learned cluster assignment matrix at layer l as $S^{(l)} \in \mathbb{R}^{n_l \times n_{l+1}}$. Each row of $S^{(l)}$ corresponds to one of the n_l nodes (or clusters) at layer l, and each column of $S^{(l)}$ corresponds to one of the n_{l+1} clusters at the next layer $l + 1$. Intuitively, $S^{(l)}$ provides a soft assignment of each node at layer l to a cluster in the next coarsened layer $l + 1$.

Suppose that $S^{(l)}$ has already been computed, i.e., that we have computed the assignment matrix at the l-th layer of our model. We denote the input adjacency matrix at this layer as $A^{(l)}$ and denote the input node embedding matrix at this layer as $Z^{(l)}$. Given these inputs, the DIFFPOOL layer $(A^{(l+1)}, X^{(l+1)}) = \text{DIFFPOOL}(A^{(l)}, Z^{(l)})$ coarsens the input graph, generating a new coarsened

adjacency matrix $A^{(l+1)}$ and a new matrix of embeddings $X^{(l+1)}$ for each of the nodes/clusters in this coarsened graph. In particular, we apply the two following equations:

$$X^{(l+1)} = {S^{(l)}}^T Z^{(l)} \in \mathbb{R}^{n_{l+1} \times d}, \tag{4.3}$$

$$A^{(l+1)} = {S^{(l)}}^T A^{(l)} S^{(l)} \in \mathbb{R}^{n_{l+1} \times n_{l+1}}. \tag{4.4}$$

Equation (4.3) takes the node embeddings $Z^{(l)}$ and aggregates these embeddings according to the cluster assignments $S^{(l)}$, generating embeddings for each of the n_{l+1} clusters. Similarly, Equation (4.4) takes the adjacency matrix $A^{(l)}$ and generates a coarsened adjacency matrix denoting the connectivity strength between each pair of clusters.

Through Equations (4.3) and (4.4), the DIFFPOOL layer coarsens the graph: the next layer adjacency matrix $A^{(l+1)}$ represents a coarsened graph with n_{l+1} nodes or *cluster nodes*, where each individual cluster node in the new coarsened graph corresponds to a cluster of nodes in the graph at layer l. Note that $A^{(l+1)}$ is a real matrix and represents a fully connected edge-weighted graph; each entry $A_{ij}^{(l+1)}$ can be viewed as the connectivity strength between cluster i and cluster j. Similarly, the i-th row of $X^{(l+1)}$ corresponds to the embedding of cluster i. Together, the coarsened adjacency matrix $A^{(l+1)}$ and cluster embeddings $X^{(l+1)}$ can be used as input to another GNN layer, a process which we describe in detail below.

Learning the assignment matrix. In the following we describe the architecture of DIFFPOOL, i.e., how DIFFPOOL generates the assignment matrix $S^{(l)}$ and embedding matrices $Z^{(l)}$ that are used in Equations (4.3) and (4.4). We generate these two matrices using two separate GNNs that are both applied to the input cluster node features $X^{(l)}$ and coarsened adjacency matrix $A^{(l)}$. The *embedding GNN* at layer l is a standard GNN module applied to these inputs:

$$Z^{(l)} = \text{GNN}_{l,\text{embed}}(A^{(l)}, X^{(l)}), \tag{4.5}$$

i.e., we take the adjacency matrix between the cluster nodes at layer l (from Equation 4.4) and the pooled features for the clusters (from Equation 4.3) and pass these matrices through a standard GNN to get new embeddings $Z^{(l)}$ for the cluster nodes. In contrast, the *pooling GNN* at layer l, uses the input cluster features $X^{(l)}$ and cluster adjacency matrix $A^{(l)}$ to generate an assignment matrix:

$$S^{(l)} = \text{softmax}\left(\text{GNN}_{l,\text{pool}}(A^{(l)}, X^{(l)})\right), \tag{4.6}$$

where the softmax function is applied in a row-wise fashion. The output dimension of $\text{GNN}_{l,\text{pool}}$

corresponds to a pre-defined maximum number of clusters in layer l, and is a hyperparameter of the model.

Note that these two GNNs consume the same input data but have distinct parameterizations and play separate roles: The embedding GNN generates new embeddings for the input nodes at this layer, while the pooling GNN generates a probabilistic assignment of the input nodes to n_{l+1} clusters.

In the base case, the inputs to Equations (4.5) and Equations (4.6) at layer $l = 0$ are simply the input adjacency matrix A and the node features F for the original graph. At the penultimate layer $L - 1$ of a deep GNN model using DIFFPOOL, we set the assignment matrix $S^{(L-1)}$ be a vector of 1's, i.e., all nodes at the final layer L are assigned to a single cluster, generating a final embedding vector corresponding to the entire graph. This final output embedding can then be used as feature input to a differentiable classifier (e.g., a softmax layer), and the entire system can be trained end-to-end using stochastic gradient descent.

Permutation invariance. Note that in order to be useful for graph classification, the pooling layer should be invariant under node permutations. For DIFFPOOL we get the following positive result, which shows that any deep GNN model based on DIFFPOOL is permutation invariant, as long as the component GNNs are permutation invariant.

Proposition 1 *Let $P \in \{0,1\}^{n \times n}$ be any permutation matrix, then* DIFFPOOL$(A, Z) =$ DIFFPOOL(PAP^T, PX) *as long as GNN$(A, X) =$ GNN(PAP^T, X) (i.e., as long as the GNN method used is permutation invariant).*

Proof 1 *Equations (4.5) and (4.6) are permutation invariant by the assumption that the GNN module is permutation invariant. And since any permutation matrix is orthogonal, applying $P^T P = I$ to Equation (4.3) and (4.4) finishes the proof.*

4.3.3 Auxiliary Link Prediction Objective and Entropy Regularization

In practice, it can be difficult to train the pooling GNN (Equation 4.4) using only gradient signal from the graph classification task. Intuitively, we have a non-convex optimization problem and it can be difficult to push the pooling GNN away from spurious local minima early in training. To alleviate this issue, we train the pooling GNN with an auxiliary link prediction objective, which encodes the intuition that nearby nodes should be pooled together. In particular, at each layer l, we minimize $L_{\text{LP}} = ||A^{(l)}, S^{(l)} S^{(l)^T}||_F$, where $|| \cdot ||_F$ denotes the Frobenius norm. Note that the adjacency matrix $A^{(l)}$ at deeper layers is a function of lower level assignment matrices, and changes during training.

Another important characteristic of the pooling GNN (Equation 4.4) is that the output cluster assignment for each node should generally be close to a one-hot vector, so that the membership for each cluster or subgraph is clearly defined. We therefore regularize the entropy of the cluster assignment by minimizing $L_{\text{E}} = \frac{1}{n}\sum_{i=1}^{n} H(S_i)$, where H denotes the entropy function, and S_i is the i-th row of S.

During training, L_{LP} and L_{E} from each layer are added to the classification loss. In practice we observe that training with the side objective takes longer to converge, but nevertheless achieves better performance and more interpretable cluster assignments.

4.4 Experiments

We evaluate the benefits of DIFFPOOL against a number of state-of-the-art graph classification approaches, with the goal of answering the following questions:

Q1 How does DIFFPOOL compare to other pooling methods proposed for GNNs (e.g., using sort pooling (Zhang et al., 2018b) or the SET2SET method (Gilmer et al., 2017))?

Q2 How does DIFFPOOL combined with GNNs compare to the state-of-the-art for graph classification task, including both GNNs and kernel-based methods?

Q3 Does DIFFPOOL compute meaningful and interpretable clusters on the input graphs?

Data sets. To probe the ability of DIFFPOOL to learn complex hierarchical structures from graphs in different domains, we evaluate on a variety of relatively large graph data sets chosen from benchmarks commonly used in graph classification (Kersting et al., 2016). We use protein data sets including ENZYMES, PROTEINS (Borgwardt et al., 2005; Feragen et al., 2013), D&D (Dobson and Doig, 2003), the social network data set REDDIT-MULTI-12K (Yanardag and Vishwanathan, 2015a), and the scientific collaboration data set COLLAB (Yanardag and Vishwanathan, 2015a). See Appendix A for statistics and properties. For all these data sets, we perform 10-fold cross-validation to evaluate model performance, and report the accuracy averaged over 10 folds.

Model configurations. In our experiments, the GNN model used for DIFFPOOL is built on top of the GRAPHSAGE architecture, as we found this architecture to have superior performance compared to the standard GCN approach as introduced in (Kipf and Welling, 2017). We use the "mean" variant of GRAPHSAGE (Hamilton et al., 2017c) and apply a DIFFPOOL layer after every two GRAPHSAGE layers in our architecture. A total of 2 DIFFPOOL layers are used for the datasets. For small datasets such as ENZYMES, PROTEINS and COLLAB, 1 DIFFPOOL layer can achieve similar performance. After each DIFFPOOL layer, 3 layers of graph convolutions are performed, before the next DIFFPOOL layer, or the readout layer. The embedding matrix and the assignment

matrix are computed by two separate GRAPHSAGE models respectively. In the 2 DIFFPOOL layer architecture, the number of clusters is set as 25% of the number of nodes before applying DIFFPOOL, while in the 1 DIFFPOOL layer architecture, the number of clusters is set as 10%. Batch normalization (Ioffe and Szegedy, 2015) is applied after every layer of GRAPHSAGE. We also found that adding an ℓ_2 normalization to the node embeddings at each layer made the training more stable. In Section 4.4.2, we also test an analogous variant of DIFFPOOL on the STRUCTURE2VEC (Dai et al., 2016a) architecture, in order to demonstrate how DIFFPOOL can be applied on top of other GNN models. All models are trained for 3 000 epochs with early stopping applied when the validation loss starts to drop. We also evaluate two simplified versions of DIFFPOOL:

- DIFFPOOL-DET, is a DIFFPOOL model where assignment matrices are generated using a deterministic graph clustering algorithm (Dhillon et al., 2007).
- DIFFPOOL-NOLP is a variant of DIFFPOOL where the link prediction side objective is turned off.

4.4.1 Baseline Methods

In the performance comparison on graph classification, we consider baselines based upon GNNs (combined with different pooling methods) as well as state-of-the-art kernel-based approaches.

GNN-based methods.

- GRAPHSAGE with global mean-pooling (Hamilton et al., 2017c). Other GNN variants such as those proposed in (Kipf and Welling, 2017) are omitted as empirically GraphSAGE obtained higher performance in the task.
- STRUCTURE2VEC (S2V) (Dai et al., 2016a) is a state-of-the-art graph representation learning algorithm that combines a latent variable model with GNNs. It uses global mean pooling.
- Edge-conditioned filters in CNN for graphs (ECC) (Simonovsky and Komodakis, 2017) incorporates edge information into the GCN model and performs pooling using a graph coarsening algorithm.
- PATCHYSAN (Niepert et al., 2016) defines a receptive field (neighborhood) for each node, and using a canonical node ordering, applies convolutions on linear sequences of node embeddings.
- SET2SET replaces the global mean-pooling in the traditional GNN architectures by the aggregation used in SET2SET (Vinyals et al., 2015). Set2Set aggregation has been shown to perform better than mean pooling in previous work (Gilmer et al., 2017). We use GRAPHSAGE as the base GNN model.
- SORTPOOL (Zhang et al., 2018b) applies a GNN architecture and then performs a single layer of soft pooling followed by 1D convolution on sorted node embeddings.

For all the GNN baselines, we use 10-fold cross validation numbers reported by the original authors when possible. For the GRAPHSAGE and SET2SET baselines, we use the base implementation and hyperparameter sweeps as in our DIFFPOOL approach. When baseline approaches did not have the necessary published numbers, we contacted the original authors and used their code (if available) to run the model, performing a hyperparameter search based on the original author's guidelines.

Kernel-based algorithms. We use the GRAPHLET (Shervashidze et al., 2009), the SHORTEST-PATH (Borgwardt and Kriegel, 2005), WEISFEILER-LEHMAN kernel (WL) (Shervashidze et al., 2011a), and WEISFEILER-LEHMAN OPTIMAL ASSIGNMENT KERNEL (WL-OA) (Kriege et al., 2016) as kernel baselines. For each kernel, we computed the normalized gram matrix. We computed the classification accuracies using the C-SVM implementation of LIBSVM (Chang and Lin, 2011), using 10-fold cross validation. The C parameter was selected from $\{10^{-3}, 10^{-2}, \ldots, 10^{2}, 10^{3}\}$ by 10-fold cross validation on the training folds. Moreover, for WL and WL-OA we additionally selected the number of iteration from $\{0, \ldots, 5\}$.

4.4.2 Results for Graph Classification

Table 4.1 compares the performance of DIFFPOOL to these state-of-the-art graph classification baselines. These results provide positive answers to our motivating questions **Q1** and **Q2**: We observe that our DIFFPOOL approach obtains the highest average performance among all pooling approaches for GNNs, improves upon the base GRAPHSAGE architecture by an average of 6.27%, and achieves state-of-the-art results on 4 out of 5 benchmarks. Interestingly, our simplified model variant, DIFFPOOL-DET, achieves state-of-the-art performance on the COLLAB benchmark. This is because many collaboration graphs in COLLAB show only single-layer community structures, which can be captured well with pre-computed graph clustering algorithm (Dhillon et al., 2007). One observation is that despite significant performance improvement, DIFFPOOL could be unstable to train, and there is significant variation in accuracy across different runs, even with the same hyperparameter setting. It is observed that adding the link predictioin objective makes training more stable, and reduces the standard deviation of accuracy across different runs.

Differentiable Pooling on STRUCTURE2VEC. DIFFPOOL can be applied to other GNN architectures besides GRAPHSAGE to capture hierarchical structure in the graph data. To further support answering **Q1**, we also applied DIFFPOOL on Structure2Vec (S2V). We ran experiments using S2V with three layer architecture, as reported in (Dai et al., 2016a). In the first variant, one DIFFPOOL layer is applied after the first layer of S2V, and two more S2V layers are stacked on top of the output of DIFFPOOL. The second variant applies one DIFFPOOL layer after the first and

Table 4.1: Classification accuracies in percent. The far-right column gives the relative increase in accuracy compared to the baseline GRAPHSAGE approach.

	Method	Data Set					
		ENZYMES	D&D	REDDIT-MULTI-12K	COLLAB	PROTEINS	Gain
Kernel	GRAPHLET	41.03	74.85	21.73	64.66	72.91	
	SHORTEST-PATH	42.32	78.86	36.93	59.10	76.43	
	1-WL	53.43	74.02	39.03	78.61	73.76	
	WL-OA	60.13	79.04	44.38	80.74	75.26	
GNN	PATCHYSAN	–	76.27	41.32	72.60	75.00	4.17
	GRAPHSAGE	54.25	75.42	42.24	68.25	70.48	–
	ECC	53.50	74.10	41.73	67.79	72.65	0.11
	SET2SET	60.15	78.12	43.49	71.75	74.29	3.32
	SORTPOOL	57.12	79.37	41.82	73.76	75.54	3.39
	DIFFPOOL-DET	58.33	75.47	46.18	**82.13**	75.62	5.42
	DIFFPOOL-NOLP	61.95	79.98	46.65	75.58	76.22	5.95
	DIFFPOOL	**62.53**	**80.64**	**47.08**	75.48	**76.25**	**6.27**

second layer of S2V respectively. In both variants, S2V model is used to compute the embedding matrix, while GRAPHSAGE model is used to compute the assignment matrix.

Table 4.2: Accuracy results of applying DIFFPOOL to S2V.

Data Set	Method		
	S2V	S2V WITH 1 DIFFPOOL	S2V WITH 2 DIFFPOOL
ENZYMES	61.10	62.86	**63.33**
D&D	78.92	80.75	**82.07**

The results in terms of classification accuracy are summarized in Table 4.2. We observe that DIFFPOOL significantly improves the performance of S2V on both ENZYMES and D&D data sets. Similar performance trends are also observed on other data sets. The results demonstrate that DIFFPOOL is a general strategy to pool over hierarchical structure that can benefit different GNN architectures.

Running time. Although applying DIFFPOOL requires additional computation of an assignment matrix, we observed that DIFFPOOL did not incur substantial additional running time in practice. This is because each DIFFPOOL layer reduces the size of graphs by extracting a coarser representation of the graph, which speeds up the graph convolution operation in the next layer. Concretely, we found that GRAPHSAGE with DIFFPOOL was 12$\times$ faster than the GRAPHSAGE model with SET2SET pooling, while still achieving significantly higher accuracy on all benchmarks.

4.4.3 Analysis of Cluster Assignment in DIFFPOOL

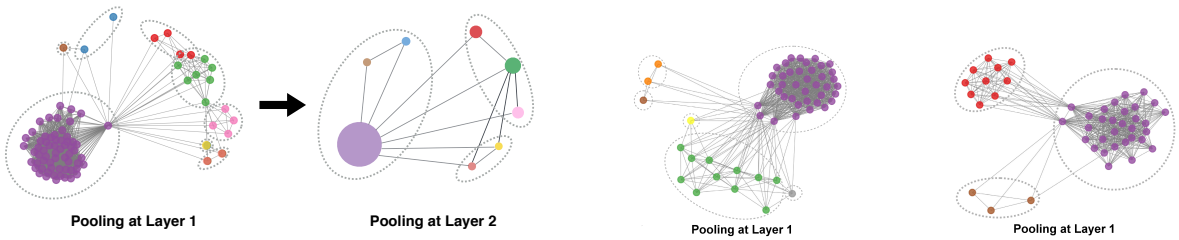
Hierarchical cluster structure. To address **Q3**, we investigated the extent to which DIFFPOOL learns meaningful node clusters by visualizing the cluster assignments in different layers. Figure 4.2 shows such a visualization of node assignments in the first and second layers on a graph from COLLAB data set, where node color indicates its cluster membership. Node cluster membership is determined by taking the argmax of its cluster assignment probabilities. We observe that even when learning cluster assignment based solely on the graph classification objective, DIFFPOOL can still capture the hierarchical community structure. We also observe significant improvement in membership assignment quality with link prediction auxiliary objectives.

Dense vs. sparse subgraph structure. In addition, we observe that DIFFPOOL learns to collapse nodes into soft clusters in a non-uniform way, with a tendency to collapse densely-connected subgraphs into clusters. Since GNNs can efficiently perform message-passing on dense, clique-like subgraphs (due to their small diameters) (Liao et al., 2018), pooling together nodes in such a dense subgraph is not likely to lead to any loss of structural information. This intuitively explains why collapsing dense subgraphs is a useful pooling strategy for DIFFPOOL. In contrast, sparse subgraphs may contain many interesting structures, including path-, cycle- and tree-like structures, and given the high-diameter induced by sparsity, GNN message-passing may fail to capture these structures. Thus, by separately pooling distinct parts of a sparse subgraph, DIFFPOOL can learn to capture the meaningful structures present in sparse graph regions (e.g., as in Figure 4.2).

Assignment for nodes with similar representations. Since the assignment network computes the soft cluster assignment based on features of input nodes and their neighbors, nodes with both similar input features and neighborhood structure will have similar cluster assignment. In fact, one can construct synthetic cases where 2 nodes, although far away, have exactly the same neighborhood structure and features for self and all neighbors. In this case the pooling network is forced to assign them into the same cluster, which is different from the concept of pooling in other architectures such as image ConvNets. In some cases we do observe that disconnected nodes are pooled together.

In practice we rely on the identifiability assumption similar to Theorem 1 in GraphSAGE (Hamilton et al., 2017c), where nodes are identifiable via their features. This holds in many real datasets [3]. The auxiliary link prediction objective is observed to also help discouraging nodes that are far away to be pooled together. Furthermore, it is possible to use more sophisticated GNN aggregation function such as high-order moments (Verma and Zhang, 2018) to distinguish nodes

[3]However, some chemistry molecular graph datasets contain many nodes that are structurally similar, and assignment network is observed to pool together nodes that are far away.



(a) (b) (c)

Figure 4.2: Visualization of hierarchical cluster assignment in DIFFPOOL, using example graphs from COLLAB. The left figure (a) shows hierarchical clustering over two layers, where nodes in the second layer correspond to clusters in the first layer. (Colors are used to connect the nodes/clusters across the layers, and dotted lines are used to indicate clusters.) The right two plots (b and c) show two more examples first-layer clusters in different graphs. Note that although we globally set the number of clusters to be 25% of the nodes, the assignment GNN automatically learns the appropriate number of meaningful clusters to assign for these different graphs.

that are similar in structure and feature space. The overall framework remains unchanged.

Sensitivity of the Pre-defined Maximum Number of Clusters. We found that the assignment varies according to the depth of the network and C, the maximum number of clusters. With larger C, the pooling GNN can model more complex hierarchical structure. The trade-off is that very large C results in more noise and less efficiency. Although the value of C is a pre-defined parameter, the pooling net learns to use the appropriate number of clusters by end-to-end training. In particular, some clusters might not be used by the assignment matrix. Column corresponding to unused cluster has low values for all nodes. This is observed in Figure 4.2(c), where nodes are assigned predominantly into 3 clusters.

Chapter 5

Hyperbolic Graph Convolutional Networks

In Chater 4, DiffPool leverages the inductive bias of hierarchy in graphs to design a pooling strategy for graph embeddings. In this chapter, let us consider a different concept of hierarchies in graphs, as a tree-like structure: in many real-world graphs, there are a few high-hierarchy nodes, which are connected to exponentially many lower-level nodes, forming a tree-like structure. Examples include knowledge graphs such as WordNet and biological graphs such as gene ontology. The proposed method of hyperbolic graph convolutional networks (HGCN) tackles prediction tasks on tree-like graphs by embedding nodes into a hyperbolic space through graph neural networks.

5.1 Introduction

Graph Convolutional Neural Networks (GCNs) are state-of-the-art models for representation learning in graphs, where nodes of the graph are embedded into points in Euclidean space (Hamilton et al., 2017c; Kipf and Welling, 2016a; Veličković et al., 2018a; Xu et al., 2019). However, many real-world graphs, such as protein interaction networks and social networks, often exhibit scale-free or hierarchical structure (Clauset et al., 2008; Zitnik et al., 2019) and Euclidean embeddings, used by existing GCNs, have a high distortion when embedding such graphs (Chen et al., 2013; Ravasz and Barabási, 2003). In particular, scale-free graphs have tree-like structure and in such graphs the graph volume, defined as the number of nodes within some radius to a center node, grows exponentially as a function of radius. However, the volume of balls in Euclidean space only grows polynomially with respect to the radius, which leads to high distortion embeddings (Sala et al., 2018; Sarkar, 2011), while in hyperbolic space, this volume grows exponentially.

Hyperbolic geometry offers an exciting alternative as it enables embeddings with much smaller

distortion when embedding scale-free and hierarchical graphs. However, current hyperbolic embedding techniques only account for the graph structure and do not leverage rich node features. For instance, Poincaré embeddings (Nickel and Kiela, 2017) capture the hyperbolic properties of real graphs by learning shallow embeddings with hyperbolic distance metric and Riemannian optimization. Compared to deep alternatives such as GCNs, shallow embeddings do not take into account features of nodes, lack scalability, and lack inductive capability. Furthermore, in practice, optimization in hyperbolic space is challenging.

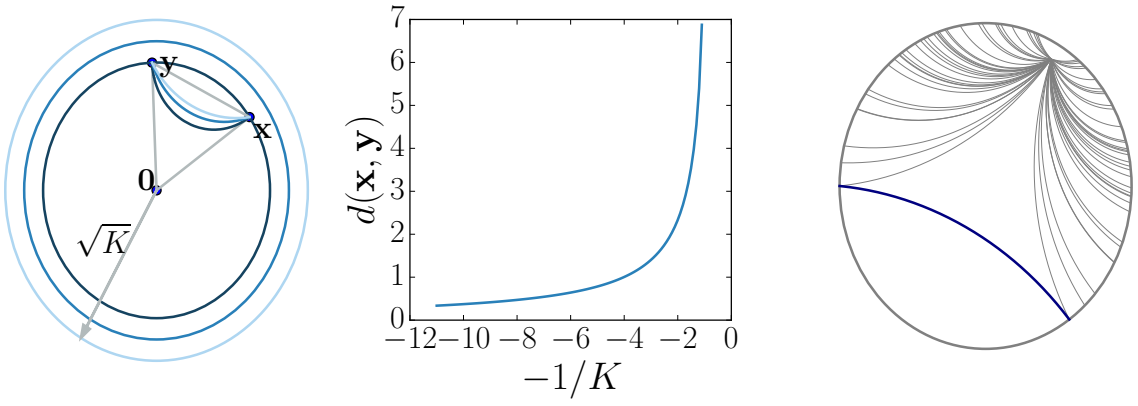
While extending GCNs to hyperbolic geometry has the potential to lead to more faithful embeddings and accurate models, it also poses many hard challenges: (1) Input node features are usually Euclidean, and it is not clear how to optimally use them as inputs to hyperbolic neural networks; (2) It is not clear how to perform set aggregation, a key step in message passing, in hyperbolic space; And (3) one needs to choose hyperbolic spaces with the right curvature at every layer of the GCN.

Here we solve the above challenges and propose *Hyperbolic Graph Convolutional Networks (*HGCN*)*[1], a class of graph representation learning models that combines the expressiveness of GCNs and hyperbolic geometry to learn improved representations for real-world hierarchical and scale-free graphs in inductive settings: (1) We derive the core operations of GCNs in the hyperboloid model of hyperbolic space to transform input features which lie in Euclidean space into hyperbolic embeddings; (2) We introduce a hyperbolic attention-based aggregation scheme that captures hierarchical structure of networks; (3) At different layers of HGCN we apply feature transformations in hyperbolic spaces of different trainable curvatures to learn low-distortion hyperbolic embeddings.

The transformation between different hyperbolic spaces at different layers allows HGCN to find the best geometry of hidden layers to achieve low distortion and high separation of class labels. Our approach jointly trains the weights for hyperbolic graph convolution operators, layer-wise curvatures and hyperbolic attention to learn inductive embeddings that reflect hierarchies in graphs.

Compared to Euclidean GCNs, HGCN offers improved expressiveness for hierarchical graph data. We demonstrate the efficacy of HGCN in link prediction and node classification tasks on a wide range of open graph datasets which exhibit different extent of hierarchical structure. Experiments show that HGCN significantly outperforms Euclidean-based state-of-the-art graph neural networks on scale-free graphs and reduces error from 11.5% up to 47.5% on node classification tasks and from 28.2% up to 63.1% on link prediction tasks. Furthermore, HGCN achieves new

[1]Project website with code and data: http://snap.stanford.edu/hgcn



(a) (b) (c)

Figure 5.1: (a) Poincaré disk geodesics (shortest path) connecting $\mathbf{x}$ and $\mathbf{y}$ for different curvatures. As curvature $(-1/K)$ decreases, the distance between $\mathbf{x}$ and $\mathbf{y}$ increases, and the geodesics lines get closer to the origin. Center: Hyperbolic distance vs curvature. (b) Hyperbolic distance vs curvature. (c) Poincaré geodesic lines.

state-of-the-art results on the standard PUBMED benchmark. Finally, we analyze the notion of hierarchy learned by HGCN and show how the embedding geometry transforms from Euclidean features to hyperbolic embeddings.

5.2 Related Works to Hyperbolic Embeddings

The problem of graph representation learning belongs to the field of geometric deep learning. There exist two major types of approaches: transductive shallow embeddings and inductive GCNs. **Transductive, shallow embeddings**. The first type of approach attempts to optimize node embeddings as parameters by minimizing a reconstruction error. In other words, the mapping from nodes in a graph to embeddings is an embedding look-up. Examples include matrix factorization (Belkin and Niyogi, 2002; Kruskal, 1964) and random walk methods (Grover and Leskovec, 2016; Perozzi et al., 2014). Shallow embedding methods have also been developed in hyperbolic geometry (Nickel and Kiela, 2017, 2018) for reconstructing trees (Sarkar, 2011) and graphs (Chamberlain et al., 2017; Gu et al., 2019; Kleinberg, 2007), or embedding text (Tifrea et al., 2019). However, shallow (Euclidean and hyperbolic) embedding methods have three major downsides: (1) They fail to leverage rich node feature information, which can be crucial in tasks such as node classification. (2) These methods are transductive, and therefore cannot be used for inference on unseen graphs. And, (3) they scale poorly as the number of model parameters grows linearly with the number of nodes.

Hyperbolic Neural Networks. Hyperbolic geometry has been applied to neural networks, to problems of computer vision or natural language processing (Dhingra et al., 2018; Gulcehre et al., 2019; Khrulkov et al., 2019; Tay et al., 2018). More recently, hyperbolic neural networks (Ganea et al., 2018) were proposed, where core neural network operations are in hyperbolic space. In contrast to previous work, we derive the core neural network operations in a more stable model of hyperbolic space, and propose new operations for set aggregation, which enables HGCN to perform graph convolutions with attention in hyperbolic space with trainable curvature. After NeurIPS 2019 announced accepted papers, we also became aware of the concurrently developed HGNN model (Liu et al., 2019a) for learning GNNs in hyperbolic space. The main difference with our work is how our HGCN defines the architecture for neighborhood aggregation and uses a learnable curvature. Additionally, while (Liu et al., 2019a) demonstrates strong performance on graph classification tasks and provides an elegant extension to dynamic graph embeddings, we focus on link prediction and node classification.

5.3 Background of Hyperbolic Embeddings

Problem setting. Without loss of generality we describe graph representation learning on a single graph. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph with vertex set $\mathcal{V}$ and edge set $\mathcal{E}$, and let $(\mathbf{x}_i^{0,E})_{i\in\mathcal{V}}$ be d-dimensional input node features where ^{0} indicates the first layer. We use the superscript ^{E} to indicate that node features lie in a Euclidean space and use ^{H} to denote hyperbolic features. The goal in graph representation learning is to learn a mapping f which maps nodes to embedding vectors:

$$f : (\mathcal{V}, \mathcal{E}, (\mathbf{x}_i^{0,E})_{i\in\mathcal{V}}) \to Z \in \mathbb{R}^{|\mathcal{V}|\times d'},$$

where $d' \ll |\mathcal{V}|$. These embeddings should capture both structural and semantic information and can then be used as input for downstream tasks such as node classification and link prediction.

Graph Convolutional Neural Networks (GCNs). Let $\mathcal{N}(i) = \{j : (i, j) \in \mathcal{E}\}$ denote a set of neighbors of $i \in \mathcal{V}$, (W^ℓ, $\mathbf{b}^\ell$) be weights and bias parameters for layer ℓ, and $\sigma(\cdot)$ be a non-linear activation function. General GCN message passing rule at layer ℓ for node i then consists of:

$$\mathbf{h}_i^{\ell,E} = W^\ell \mathbf{x}_i^{\ell-1,E} + \mathbf{b}^\ell \qquad \text{(feature transform)} \tag{5.1}$$

$$\mathbf{x}_i^{\ell,E} = \sigma(\mathbf{h}_i^{\ell,E} + \sum_{j\in\mathcal{N}(i)} w_{ij}\mathbf{h}_j^{\ell,E}) \qquad \text{(neighborhood aggregation)} \tag{5.2}$$

where aggregation weights w_{ij} can be computed using different mechanisms (Hamilton et al., 2017c; Kipf and Welling, 2016a; Veličković et al., 2018a). Message passing is then performed

for multiple layers to propagate messages over network neighborhoods. Unlike shallow methods, GCNs leverage node features and can be applied to unseen nodes/graphs in inductive settings.

The hyperboloid model of hyperbolic space. We review basic concepts of hyperbolic geometry that serve as building blocks for HGCN. Hyperbolic geometry is a non-Euclidean geometry with a constant negative curvature, where curvature measures how a geometric object deviates from a flat plane (*cf.* (Robbin and Salamon, 2011) for an introduction to differential geometry). Here, we work with the hyperboloid model for its simplicity and its numerical stability (Nickel and Kiela, 2018). We review results for any constant negative curvature, as this allows us to learn curvature as a model parameter, leading to more stable optimization (*cf.* Section 5.4.5 for more details).

Hyperboloid manifold. We first introduce our notation for the hyperboloid model of hyperbolic space. Let $\langle ., . \rangle_{\mathcal{L}} : \mathbb{R}^{d+1} \times \mathbb{R}^{d+1} \to \mathbb{R}$ denote the Minkowski inner product, $\langle \mathbf{x}, \mathbf{y} \rangle_{\mathcal{L}} \coloneqq -x_0 y_0 + x_1 y_1 + \ldots + x_d y_d$. We denote $\mathbb{H}^{d,K}$ as the hyperboloid manifold in d dimensions with constant negative **curvature** $-1/K$ $(K > 0)$, and $\mathcal{T}_{\mathbf{x}}\mathbb{H}^{d,K}$ the (Euclidean) **tangent space** centered at point $\mathbf{x}$

$$\mathbb{H}^{d,K} \coloneqq \{\mathbf{x} \in \mathbb{R}^{d+1} : \langle \mathbf{x}, \mathbf{x} \rangle_{\mathcal{L}} = -K, x_0 > 0\} \quad \mathcal{T}_{\mathbf{x}}\mathbb{H}^{d,K} \coloneqq \{\mathbf{v} \in \mathbb{R}^{d+1} : \langle \mathbf{v}, \mathbf{x} \rangle_{\mathcal{L}} = 0\}. \tag{5.3}$$

Now for $\mathbf{v}$ and $\mathbf{w}$ in $\mathcal{T}_{\mathbf{x}}\mathbb{H}^{d,K}$, $g^K_{\mathbf{x}}(\mathbf{v}, \mathbf{w}) \coloneqq \langle \mathbf{v}, \mathbf{w} \rangle_{\mathcal{L}}$ is a Riemannian metric tensor (Robbin and Salamon, 2011) and $(\mathbb{H}^{d,K}, g^K_{\mathbf{x}})$ is a Riemannian manifold with negative curvature $-1/K$. $\mathcal{T}_{\mathbf{x}}\mathbb{H}^{d,K}$ is a local, first-order approximation of the hyperbolic manifold at $\mathbf{x}$ and the restriction of the Minkowski inner product to $\mathcal{T}_{\mathbf{x}}\mathbb{H}^{d,K}$ is positive definite. $\mathcal{T}_{\mathbf{x}}\mathbb{H}^{d,K}$ is useful to perform Euclidean operations undefined in hyperbolic space and we denote $||\mathbf{v}||_{\mathcal{L}} = \sqrt{\langle \mathbf{v}, \mathbf{v} \rangle_{\mathcal{L}}}$ as the norm of $\mathbf{v} \in \mathcal{T}_{\mathbf{x}}\mathbb{H}^{d,K}$.

Geodesics and induced distances. Next, we introduce the notion of geodesics and distances in manifolds, which are generalizations of shortest paths in graphs or straight lines in Euclidean geometry (Figure 5.1). Geodesics and distance functions are particularly important in graph embedding algorithms, as a common optimization objective is to minimize geodesic distances between connected nodes. Let $\mathbf{x} \in \mathbb{H}^{d,K}$ and $\mathbf{u} \in \mathcal{T}_{\mathbf{x}}\mathbb{H}^{d,K}$, and assume that $\mathbf{u}$ is unit-speed, *i.e.* $\langle \mathbf{u}, \mathbf{u} \rangle_{\mathcal{L}} = 1$, then we have the following result: Let $\mathbf{x} \in \mathbb{H}^{d,K}$, $\mathbf{u} \in \mathcal{T}_{\mathbf{x}}\mathbb{H}^{d,K}$ be unit-speed. The unique unit-speed geodesic $\gamma_{\mathbf{x} \to \mathbf{u}}(\cdot)$ such that $\gamma_{\mathbf{x} \to \mathbf{u}}(0) = \mathbf{x}$, $\dot{\gamma}_{\mathbf{x} \to \mathbf{u}}(0) = \mathbf{u}$ is $\gamma^K_{\mathbf{x} \to \mathbf{u}}(t) = \cosh\left(\frac{t}{\sqrt{K}}\right)\mathbf{x} + \sqrt{K}\sinh\left(\frac{t}{\sqrt{K}}\right)\mathbf{u}$, and the intrinsic distance function between two points $\mathbf{x}, \mathbf{y}$ in $\mathbb{H}^{d,K}$ is then:

$$d^K_{\mathcal{L}}(\mathbf{x}, \mathbf{y}) = \sqrt{K}\operatorname{arcosh}(-\langle \mathbf{x}, \mathbf{y} \rangle_{\mathcal{L}}/K). \tag{5.4}$$

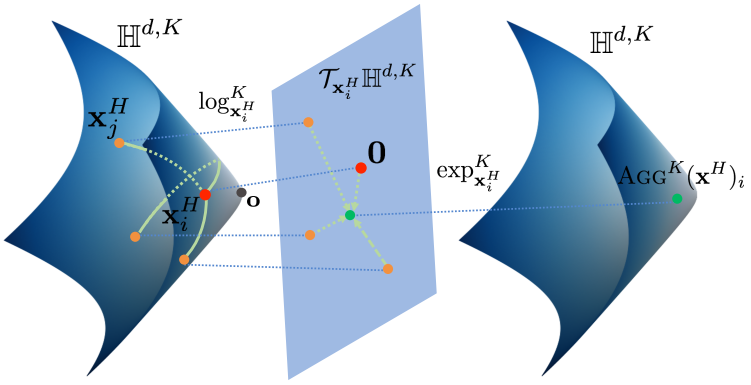


Figure 5.2: HGCN neighborhood aggregation (Eq. 5.9) first maps messages/embeddings to the tangent space, performs the aggregation in the tangent space, and then maps back to the hyperbolic space.

Exponential and logarithmic maps. Mapping between tangent space and hyperbolic space is done by exponential and logarithmic maps. Given $\mathbf{x} \in \mathbb{H}^{d,K}$ and a tangent vector $\mathbf{v} \in \mathcal{T}_{\mathbf{x}}\mathbb{H}^{d,K}$, the exponential map $\exp^K_{\mathbf{x}} : \mathcal{T}_{\mathbf{x}}\mathbb{H}^{d,K} \to \mathbb{H}^{d,K}$ assigns to $\mathbf{v}$ the point $\exp^K_{\mathbf{x}}(\mathbf{v}) \coloneqq \gamma(1)$, where γ is the unique geodesic satisfying $\gamma(0) = \mathbf{x}$ and $\dot{\gamma}(0) = \mathbf{v}$. The logarithmic map is the reverse map that maps back to the tangent space at $\mathbf{x}$ such that $\log^K_{\mathbf{x}}(\exp^K_{\mathbf{x}}(\mathbf{v})) = \mathbf{v}$. In general Riemannian manifolds, these operations are only defined locally but in the hyperbolic space, they form a bijection between the hyperbolic space and the tangent space at a point. We have the following direct expressions of the exponential and the logarithmic maps, which allow us to perform operations on points on the hyperboloid manifold by mapping them to tangent spaces and vice-versa: For $\mathbf{x} \in \mathbb{H}^{d,K}$, $\mathbf{v} \in \mathcal{T}_{\mathbf{x}}\mathbb{H}^{d,K}$ and $\mathbf{y} \in \mathbb{H}^{d,K}$ such that $\mathbf{v} \neq \mathbf{0}$ and $\mathbf{y} \neq \mathbf{x}$, the exponential and logarithmic maps of the hyperboloid model are given by:

$$\exp^K_{\mathbf{x}}(\mathbf{v}) = \cosh\left(\frac{||\mathbf{v}||_{\mathcal{L}}}{\sqrt{K}}\right)\mathbf{x} + \sqrt{K}\sinh\left(\frac{||\mathbf{v}||_{\mathcal{L}}}{\sqrt{K}}\right)\frac{\mathbf{v}}{||\mathbf{v}||_{\mathcal{L}}}, \quad \log^K_{\mathbf{x}}(\mathbf{y}) = d^K_{\mathcal{L}}(\mathbf{x},\mathbf{y})\frac{\mathbf{y} + \frac{1}{K}\langle \mathbf{x}, \mathbf{y}\rangle_{\mathcal{L}}\mathbf{x}}{||\mathbf{y} + \frac{1}{K}\langle \mathbf{x}, \mathbf{y}\rangle_{\mathcal{L}}\mathbf{x}||_{\mathcal{L}}}.$$

5.4 Hyperbolic GCN Architecture

Here we introduce HGCN, a generalization of inductive GCNs in hyperbolic geometry that benefits from the expressiveness of both graph neural networks and hyperbolic embeddings. First, since input features are often Euclidean, we derive a mapping from Euclidean features to hyperbolic space. Next, we derive two components of graph convolution: The analogs of Euclidean feature

transformation and feature aggregation (Equations 5.1, 5.2) in the hyperboloid model. Finally, we introduce the HGCN algorithm with trainable curvature.

5.4.1 Mapping from Euclidean to hyperbolic spaces

HGCN first maps input features to the hyperboloid manifold via the $\exp$ map. Let $\mathbf{x}^{0,E} \in \mathbb{R}^d$ denote input Euclidean features. For instance, these features could be produced by pre-trained Euclidean neural networks. Let $\mathbf{o} := \{\sqrt{K}, 0, \ldots, 0\} \in \mathbb{H}^{d,K}$ denote the north pole (origin) in $\mathbb{H}^{d,K}$, which we use as a reference point to perform tangent space operations. We have $\langle (0, \mathbf{x}^{0,E}), \mathbf{o} \rangle = 0$. Therefore, we interpret $(0, \mathbf{x}^{0,E})$ as a point in $\mathcal{T}_{\mathbf{o}}\mathbb{H}^{d,K}$ and use Proposition 5.3 to map it to $\mathbb{H}^{d,K}$ with:

$$\mathbf{x}^{0,H} = \exp_{\mathbf{o}}^{K}((0, \mathbf{x}^{0,E})) = \left(\sqrt{K}\cosh\left(\frac{||\mathbf{x}^{0,E}||_2}{\sqrt{K}}\right), \sqrt{K}\sinh\left(\frac{||\mathbf{x}^{0,E}||_2}{\sqrt{K}}\right)\frac{\mathbf{x}^{0,E}}{||\mathbf{x}^{0,E}||_2} \right). \tag{5.5}$$

5.4.2 Feature transform in hyperbolic space

The feature transform in Equation 5.1 is used in GCN to map the embedding space of one layer to the next layer embedding space and capture large neighborhood structures. We now want to learn transformations of points on the hyperboloid manifold. However, there is no notion of vector space structure in hyperbolic space. We build upon Hyperbolic Neural Network (HNN) (Ganea et al., 2018) and derive transformations in the hyperboloid model. The main idea is to leverage the $\exp$ and $\log$ maps in Proposition 5.3 so that we can use the tangent space $\mathcal{T}_{\mathbf{o}}\mathbb{H}^{d,K}$ to perform Euclidean transformations.

Hyperboloid linear transform. Linear transformation requires multiplication of the embedding vector by a weight matrix, followed by bias translation. To compute matrix vector multiplication, we first use the logarithmic map to project hyperbolic points $\mathbf{x}^H$ to $\mathcal{T}_{\mathbf{o}}\mathbb{H}^{d,K}$. Thus the matrix representing the transform is defined on the tangent space, which is Euclidean and isomorphic to $\mathbb{R}^d$. We then project the vector in the tangent space back to the manifold using the exponential map. Let W be a $d' \times d$ weight matrix. We define the hyperboloid matrix multiplication as:

$$W \otimes^K \mathbf{x}^H := \exp_{\mathbf{o}}^K(W \log_{\mathbf{o}}^K(\mathbf{x}^H)), \tag{5.6}$$

where $\log_{\mathbf{o}}^K(\cdot)$ is on $\mathbb{H}^{d,K}$ and $\exp_{\mathbf{o}}^K(\cdot)$ maps to $\mathbb{H}^{d',K}$.

In order to perform bias addition, we use a result from the HNN model and define $\mathbf{b}$ as an Euclidean vector located at $\mathcal{T}_{\mathbf{o}}\mathbb{H}^{d,K}$. We then parallel transport $\mathbf{b}$ to the tangent space of the hyperbolic point of interest and map it to the manifold. If $P^K_{\mathbf{o} \to \mathbf{x}^H}(\cdot)$ is the parallel transport from

$\mathcal{T}_{\mathbf{o}}\mathbb{H}^{d',K}$ to $\mathcal{T}_{\mathbf{x}^H}\mathbb{H}^{d',K}$ described by the Levi-Cevita Connection, the hyperboloid bias addition is then defined as:

$$\mathbf{x}^H \oplus^K \mathbf{b} \coloneqq \exp^K_{\mathbf{x^H}}(P^K_{\mathbf{o}\to\mathbf{x}^H}(\mathbf{b})). \tag{5.7}$$

5.4.3 Neighborhood aggregation on the hyperboloid manifold

Aggregation (Equation 5.2) is a crucial step in GCNs as it captures neighborhood structures and features. Suppose that $\mathbf{x}_i$ aggregates information from its neighbors $(\mathbf{x}_j)_{j\in\mathcal{N}(i)}$ with weights $(w_j)_{j\in\mathcal{N}(i)}$. Mean aggregation in Euclidean GCN computes the weighted average $\sum_{j\in\mathcal{N}(i)} w_j\mathbf{x}_j$. An analog of mean aggregation in hyperbolic space is the Fréchet mean (Fréchet, 1948), which, however, has no closed form solution. Instead, we propose to perform aggregation in tangent spaces using hyperbolic attention.

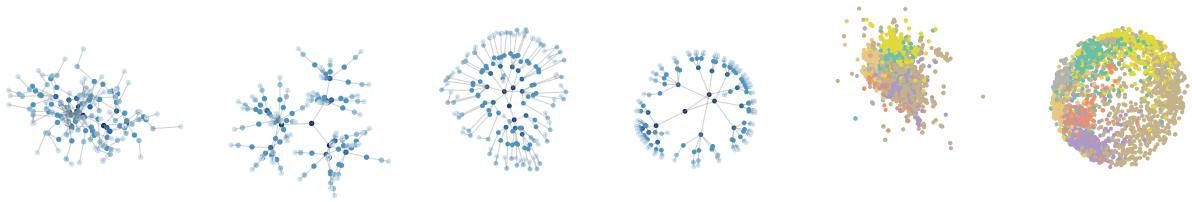
Attention based aggregation. Attention in GCNs learns a notion of neighbors' importance and aggregates neighbors' messages according to their importance to the center node. However, attention on Euclidean embeddings does not take into account the hierarchical nature of many real-world networks. Thus, we further propose hyperbolic attention-based aggregation. Given hyperbolic embeddings $(\mathbf{x}^H_i, \mathbf{x}^H_j)$, we first map $\mathbf{x}^H_i$ and $\mathbf{x}^H_j$ to the tangent space of the origin to compute attention weights w_{ij} with concatenation and Euclidean Multi-layer Percerptron (MLP). We then propose a hyperbolic aggregation to average nodes' representations:

$$w_{ij} = \textsc{Softmax}_{j\in\mathcal{N}(i)}(\text{MLP}(\log^K_{\mathbf{o}}(\mathbf{x}^H_i)||\log^K_{\mathbf{o}}(\mathbf{x}^H_j))) \tag{5.8}$$

$$\text{AGG}^K(\mathbf{x}^H)_i = \exp^K_{\mathbf{x}^H_i}\left(\sum_{j\in\mathcal{N}(i)} w_{ij}\log^K_{\mathbf{x}^H_i}(\mathbf{x}^H_j)\right). \tag{5.9}$$

Note that our proposed aggregation is directly performed in the tangent space of each center point $\mathbf{x}^H_i$, as this is where the Euclidean approximation is best (*cf.* Figure 5.2). We show in our ablation experiments (*cf.* Table 5.1) that this local aggregation outperforms aggregation in tangent space at the origin ($\text{AGG}_{\mathbf{o}}$), due to the fact that relative distances have lower distortion in our approach.

Non-linear activation with different curvatures. Analogous to Euclidean aggregation (Equation 5.2), HGCN uses a non-linear activation function, $\sigma(\cdot)$ such that $\sigma(0) = 0$, to learn non-linear transformations. Given hyperbolic curvatures $-1/K_{\ell-1}, -1/K_\ell$ at layer $\ell-1$ and ℓ respectively, we introduce a hyperbolic non-linear activation $\sigma^{\otimes^{K_{\ell-1},K_\ell}}$ with different curvatures. This step is crucial as it allows us to smoothly vary curvature at each layer. More concretely, HGCN applies



(a) GCN layers. (b) HGCN layers. (c) GCN (left), HGCN (right).

Figure 5.3: Visualization of embeddings for LP on DISEASE and NC on CORA (visualization on the Poincaré disk for HGCN). (a) GCN embeddings in first and last layers for DISEASE LP hardly capture hierarchy (depth indicated by color). (b) In contrast, HGCN preserves node hierarchies. (c) On CORA NC, HGCN leads to better class separation (indicated by different colors).

the Euclidean non-linear activation in $\mathcal{T}_{\mathbf{o}}\mathbb{H}^{d,K_{\ell-1}}$ and then maps back to $\mathbb{H}^{d,K_\ell}$:

$$\sigma^{\otimes^{K_{\ell-1},K_\ell}}(\mathbf{x}^H) = \exp_{\mathbf{o}}^{K_\ell}(\sigma(\log_{\mathbf{o}}^{K_{\ell-1}}(\mathbf{x}^H))). \tag{5.10}$$

Note that in order to apply the exponential map, points must be located in the tangent space at the north pole. Fortunately, tangent spaces of the north pole are shared across hyperboloid manifolds of the same dimension that have different curvatures, making Equation 5.10 mathematically correct.

5.4.4 HGCN architecture

Having introduced all the building blocks of HGCN, we now summarize the model architecture. Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and input Euclidean features $(\mathbf{x}^{0,E})_{i\in\mathcal{V}}$, the first layer of HGCN maps from Euclidean to hyperbolic space as detailed in Section 5.4.1. HGCN then stacks multiple hyperbolic graph convolution layers. At each layer HGCN transforms and aggregates neighbour's embeddings in the tangent space of the center node and projects the result to a hyperbolic space with different curvature. Hence the message passing in a HGCN layer is:

$$\mathbf{h}_i^{\ell,H} = (W^\ell \otimes^{K_{\ell-1}} \mathbf{x}_i^{\ell-1,H}) \oplus^{K_{\ell-1}} \mathbf{b}^\ell \qquad \text{(hyperbolic feature transform)} \tag{5.11}$$

$$\mathbf{y}_i^{\ell,H} = \text{AGG}^{K_{\ell-1}}(\mathbf{h}^{\ell,H})_i \qquad \text{(attention-based neighborhood aggregation)} \tag{5.12}$$

$$\mathbf{x}_i^{\ell,H} = \sigma^{\otimes^{K_{\ell-1},K_\ell}}(\mathbf{y}_i^{\ell,H}) \qquad \text{(non-linear activation with different curvatures)} \tag{5.13}$$

where $-1/K_{\ell-1}$ and $-1/K_\ell$ are the hyperbolic curvatures at layer $\ell-1$ and ℓ respectively. Hyperbolic embeddings $(\mathbf{x}^{L,H})_{i\in\mathcal{V}}$ at the last layer can then be used to predict node attributes or links.

For link prediction, we use the Fermi-Dirac decoder (Krioukov et al., 2010; Nickel and Kiela,

2017), a generalization of sigmoid, to compute probability scores for edges:

$$p((i,j) \in \mathcal{E} | \mathbf{x}_i^{L,H}, \mathbf{x}_j^{L,H}) = \left[e^{(d_{\mathcal{L}}^{K_L}(\mathbf{x}_i^{L,H}, \mathbf{x}_j^{L,H})^2 - r)/t} + 1 \right]^{-1}, \tag{5.14}$$

where $d_{\mathcal{L}}^{K_L}(\cdot,\cdot)$ is the hyperbolic distance and r and t are hyper-parameters. We then train HGCN by minimizing the cross-entropy loss using negative sampling.

For node classification, we map the output of the last HGCN layer to the tangent space of the origin with the logarithmic map $\log_{\mathbf{o}}^{K_L}(\cdot)$ and then perform Euclidean multinomial logistic regression. Note that another possibility is to directly classify points on the hyperboloid manifold using the hyperbolic multinomial logistic loss (Ganea et al., 2018). This method performs similarly to Euclidean classification (*cf.* (Ganea et al., 2018) for an empirical comparison). Finally, we also add a link prediction regularization objective in node classification tasks, to encourage embeddings at the last layer to preserve the graph structure.

5.4.5 Trainable curvature

We further analyze the effect of trainable curvatures in HGCN. Theorem 5.4.5 (proof in Appendix B) shows that assuming infinite precision, for the link prediction task, we can achieve the same performance for varying curvatures with an affine invariant decoder by scaling embeddings. For any hyperbolic curvatures $-1/K, -1/K' < 0$, for any node embeddings $H = \{\mathbf{h}_i\} \subset \mathbb{H}^{d,K}$ of a graph G, we can find $H' \subset \mathbb{H}^{d,K'}$, $H' = \{\mathbf{h}_i' | \mathbf{h}_i' = \sqrt{\frac{K'}{K}}\mathbf{h}_i\}$, such that the reconstructed graph from H' via the Fermi-Dirac decoder is the same as the reconstructed graph from H, with different decoder parameters (r,t) and (r',t'). However, despite the same expressive power, adjusting curvature at every layer is important for good performance in practice due to factors of limited machine precision and normalization. First, with very low or very high curvatures, the scaling factor $\frac{K'}{K}$ in Theorem 5.4.5 becomes close to 0 or very large, and limited machine precision results in large error due to rounding. This is supported by Figure 5.4 and Table 5.1 where adjusting and training curvature lead to significant performance gain. Second, the norms of hidden layers that achieve the same local minimum in training also vary by a factor of $\sqrt{K}$. In practice, however, optimization is much more stable when the values are normalized (Ioffe and Szegedy, 2015). In the context of HGCN, trainable curvature provides a natural way to learn embeddings of the right scale at each layer, improving optimization. Figure 5.4 shows the effect of decreasing curvature ($K = +\infty$ is the Euclidean case) on link prediction performance.

5.5 Hyperbolic GCN Performance Evaluation

We comprehensively evaluate our method on a variety of networks, on both node classification (NC) and link prediction (LP) tasks, in transductive and inductive settings. We compare performance of HGCN against a variety of shallow and GNN-based baselines. We further use visualizations to investigate the expressiveness of HGCN in link prediction tasks, and also demonstrate its ability to learn embeddings that capture the hierarchical structure of many real-world networks.

5.5.1 Experimental setup

Datasets. We use a variety of open transductive and inductive datasets that we detail below (more details in Appendix). We compute Gromov's $\delta-$hyperbolicity (Adcock et al., 2013; Narayan and Saniee, 2011; Jonckheere et al., 2008), a notion from group theory that measures how tree-like a graph is. The lower δ, the more hyperbolic is the graph dataset, and $\delta = 0$ for trees. We conjecture that HGCN works better on graphs with small δ-hyperbolicity.

1. **Citation networks**. CORA (Sen et al., 2008) and PUBMED (Namata et al., 2012) are standard benchmarks describing citation networks where nodes represent scientific papers, edges are citations between them, and node labels are academic (sub)areas. CORA contains 2,708 machine learning papers divided into 7 classes while PUBMED has 19,717 publications in the area of medicine grouped in 3 classes.
2. **Disease propagation tree**. We simulate the SIR disease spreading model (Anderson and May, 1992), where the label of a node is whether the node was infected or not. Based on the model, we build tree networks, where node features indicate the susceptibility to the disease. We build transductive and inductive variants of this dataset, namely DISEASE and DISEASE-M (which contains multiple tree components).
3. **Protein-protein interactions (PPI) networks**. PPI is a dataset of human PPI networks (Szklarczyk et al., 2016). Each human tissue has a PPI network, and the dataset is a union of PPI networks for human tissues. Each protein has a label indicating the stem cell growth rate after 19 days (van de Leemput et al., 2014), which we use for the node classification task. The 16-dimensional feature for each node represents the RNA expression levels of the corresponding proteins, and we perform log transform on the features.
4. **Flight networks**. AIRPORT is a transductive dataset where nodes represent airports and edges represent the airline routes as from OpenFlights.org. Compared to previous compilations (Zhang and Chen, 2018), our dataset has larger size (2,236 nodes). We also augment the graph with geographic information (longitude, latitude and altitude), and GDP of the country where the

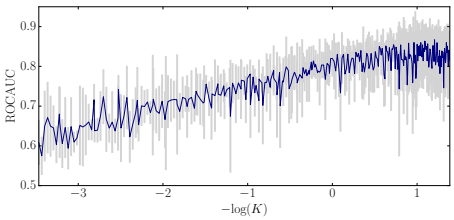


Figure 5.4: Decreasing curvature ($-1/K$) improves link prediction performance on DISEASE.

Method	DISEASE	AIRPORT
HGCN	78.4 ± 0.3	91.8 ± 0.3
HGCN-ATT$_0$	80.9 ± 0.4	92.3 ± 0.3
HGCN-ATT	82.0 ± 0.2	92.5 ± 0.2
HGCN-C	89.1 ± 0.2	94.9 ± 0.3
HGCN-ATT-C	**90.8** ± 0.3	**96.4** ± 0.1

Table 5.1: ROC AUC for link prediction on AIRPORT and DISEASE datasets.

airport belongs to. We use the population of the country where the airport belongs to as the label for node classification.

Baselines. For shallow methods, we consider Euclidean embeddings (EUC) and Poincaré embeddings (HYP) (Nickel and Kiela, 2017). We conjecture that HYP will outperform EUC on hierarchical graphs. For a fair comparison with HGCN which leverages node features, we also consider EUC-MIXED and HYP-MIXED baselines, where we concatenate the corresponding shallow embeddings with node features, followed by a MLP to predict node labels or links. For state-of-the-art Euclidean GNN models, we consider GCN (Kipf and Welling, 2016a), GraphSAGE (SAGE) (Hamilton et al., 2017c), Graph Attention Networks (GAT) (Veličković et al., 2018a) and Simplified Graph Convolution (SGC) (Wu et al., 2019)[2]. We also consider feature-based approaches: MLP and its hyperbolic variant (HNN) (Ganea et al., 2018), which does not utilize the graph structure.

Training. For all methods, we perform a hyper-parameter search on a validation set over initial learning rate, weight decay, dropout[3], number of layers, and activation functions. We measure performance on the final test set over 10 random parameter initializations. For fairness, we also control the number of dimensions to be the same (16) for all methods. We optimize all models with Adam (Kingma and Ba, 2015), except Poincaré embeddings which are optimized with RiemannianSGD (Bonnabel, 2013; Zhang et al., 2016). Further details can be found in Appendix. We open source our implementation[4] of HGCN and baselines.

Evaluation metric. In transductive LP tasks, we randomly split edges into $85/5/10\%$ for training, validation and test sets. For transductive NC, we use $70/15/15\%$ splits for AIRPORT, $30/10/60\%$ splits for DISEASE, and we use standard splits (Kipf and Welling, 2016a; Yang et al., 2016) with

[2]The equivalent of GCN in link prediction is GAE (Kipf and Welling, 2016b). We did not compare link prediction GNNs based on shallow embeddings such as (Zhang and Chen, 2018) since they are not inductive.

[3]HGCN uses DropConnect (Wan et al., 2013), as described in Appendix C.

[4]Code available at http://snap.stanford.edu/hgcn. We provide HGCN implementations for hyperboloid and Poincaré models. Empirically, both models give similar performance but hyperboloid model offers more stable optimization, because Poincaré distance is numerically unstable (Nickel and Kiela, 2018).

20 train examples per class for CORA and PUBMED. One of the main advantages of HGCN over related hyperbolic graph embedding is its inductive capability. For inductive tasks, the split is performed across graphs. All nodes/edges in training graphs are considered the training set, and the model is asked to predict node class or unseen links for test graphs. Following previous works, we evaluate link prediction by measuring area under the ROC curve on the test set and evaluate node classification by measuring F1 score, except for CORA and PUBMED, where we report accuracy as is standard in the literature.

5.5.2 Results

Table 4.1 reports the performance of HGCN in comparison to baseline methods. HGCN works best in inductive scenarios where both node features and network topology play an important role. The performance gain of HGCN with respect to Euclidean GNN models is correlated with graph hyperbolicity. HGCN achieves an average of 45.4% (LP) and 12.3% (NC) error reduction compared to the best deep baselines for graphs with high hyperbolicity (low δ), suggesting that GNNs can significantly benefit from hyperbolic geometry, especially in link prediction tasks. Furthermore, the performance gap between HGCN and HNN suggests that neighborhood aggregation has been effective in learning node representations in graphs. For example, in disease spread datasets, both Euclidean attention and hyperbolic geometry lead to significant improvement of HGCN over other baselines. This can be explained by the fact that in disease spread trees, parent nodes contaminate their children. HGCN can successfully model these asymmetric and hierarchical relationships with hyperbolic attention and improves performance over all baselines.

On the CORA dataset with low hyperbolicity, HGCN does not outperform Euclidean GNNs, suggesting that Euclidean geometry is better for its underlying graph structure. However, for small dimensions, HGCN is still significantly more effective than GCN even for CORA. Figure 5.3c shows 2D HGCN and GCN embeddings trained with LP objective, where colors denote the label class. HGCN achieves much better label class separation.

5.5.3 Analysis

Ablations. We further analyze the effect of proposed components in HGCN, namely hyperbolic attention (ATT) and trainable curvature (C) on AIRPORT and DISEASE datasets in Table 5.1. We observe that both attention and trainable curvature lead to performance gains over HGCN with fixed curvature and no attention. Furthermore, our attention model ATT outperforms $\text{ATT}_{\mathbf{o}}$ (aggregation in tangent space at $\mathbf{o}$), and we conjecture that this is because the local Euclidean average

is a better approximation near the center point rather than near $\mathbf{o}$. Finally, the addition of both ATT and C improves performance even further, suggesting that both components are important in HGCN.

Visualizations. We visualize the GCN and HGCN embeddings at the first and last layers in Figure 5.3. We train HGCN with 3-dimensional hyperbolic embeddings and map them to the Poincaré disk which is better for visualization. In contrast to GCN, tree structure is preserved in HGCN, where nodes close to the center are higher in the hierarchy of the tree. This way HGCN smoothly transforms Euclidean features to Hyperbolic embeddings that preserve node hierarchy.

Chapter 6

Multi-hop Attention-based Graph Neural Networks

In this chapter, inspired by the attention mechanism and graph diffusion, I further design a new GNN architecture which performs diffusion to obtain attention values for nodes multiple hops away, thus dramatically increases the receptive field of each GNN layer, and achieves state-of-the-art performance on a variety of standard graph benchmarks as of April 2021. The MAGNA extension improves over GraphSAGE by utilizing an attention-based multi-hop neighborhood definition, and aggregate the multi-hop neighborhood within each layer of MAGNA.

6.1 Introduction to Multi-hop GNNs

Self-attention (Bahdanau et al., 2015; Vaswani et al., 2017) has pushed the state-of-the-art in many domains including graph presentation learning (Devlin et al., 2019). Graph Attention Network (GAT) (Veličković et al., 2018b) and related models (Li et al., 2018a; Wang et al., 2019a; Liu et al., 2019b; Oono and Suzuki, 2020) developed attention mechanism for Graph Neural Networks (GNNs), which compute attention scores between nodes connected by an edge, allowing the model to attend to messages of node's neighbors.

However, such attention computation on pairs of nodes connected by edges implies that a node can only attend to its immediate neighbors to compute its (next layer) representation. This implies that receptive field of a single GNN layer is restricted to one-hop network neighborhoods. Although stacking multiple GAT layers could in principle enlarge the receptive field and learn non-neighboring interactions, such deep GAT architectures suffer from the oversmoothing problem (Wang et al., 2019a; Liu et al., 2019b; Oono and Suzuki, 2020) and do not perform well.

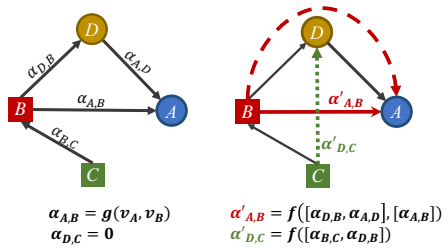


Figure 6.1: **Multi-hop attention diffusion.** Consider making a prediction at nodes A and D. **Left**: A single GAT layer computes attention scores $\boldsymbol{\alpha}$ between directly connected pairs of nodes (i.e., edges) and thus $\boldsymbol{\alpha}_{D,C} = 0$. Furthermore, the attention $\boldsymbol{\alpha}_{A,B}$ between A and B only depends on node representations of A and B. **Right**: A single MAGNA layer: (1) captures the information of D's two-hop neighbor node C via multi-hop attention $\boldsymbol{\alpha}'_{D,C}$; and (2) enhances graph structure learning by considering all paths between nodes via diffused attention, which is based on powers of graph adjacency matrix. MAGNA makes use of node D's features for attention computation between A and B. This means that two-hop attention in MAGNA is context (node D) dependent.

Furthermore, edge attention in a single GAT layer is based solely on representations of the two nodes at the edge endpoints, and does not depend on their graph neighborhood context. In other words, the one-hop attention mechanism in GATs limits their ability to explore the relationship between the broader graph structure. While previous works (Xu et al., 2018b; Klicpera et al., 2019c) have shown advantages in performing multi-hop message-passing in a single layer, these approaches are not graph-attention based. Therefore, incorporating multi-hop neighboring context into the attention computation in graph neural networks remains to be explored.

Here we present *Multi-hop Attention Graph Neural Network (*MAGNA*)*, an effective multi-hop self-attention mechanism for graph structured data. MAGNA uses a novel graph attention diffusion layer (Figure 6.1), where we first compute attention weights on edges (represented by solid arrows), and then compute self-attention weights (dotted arrows) between disconnected pairs of nodes through an attention diffusion process using the attention weights on the edges.

Our model has two main advantages: (1) MAGNA captures long-range interactions between nodes that are not directly connected but may be multiple hops away. Thus the model enables effective long-range message passing, from important nodes multiple hops away. (2) The attention computation in MAGNA is context-dependent. The attention value in GATs (Veličković et al., 2018b) only depends on node representations of the previous layer, and is zero between non-connected pairs of nodes. In contrast, for any pair of nodes within a chosen multi-hop neighborhood, MAGNA computes attention by aggregating the attention scores over all the possible paths (length ≥ 1) connecting the two nodes.

Mathematically we show that MAGNA places a Personalized Page Rank (PPR) prior on the attention values. We further apply spectral graph analysis to show that MAGNA emphasizes on

large-scale graph structure and lowering high-frequency noise in graphs. Specifically, MAGNA enlarges the lower Laplacian eigen-values, which correspond to the large-scale structure in the graph, and suppresses the higher Laplacian eigen-values which correspond to more noisy and fine-grained information in the graph.

We experiment on standard datasets for semi-supervised node classification as well as knowledge graph completion. Experiments show that MAGNA achieves state-of-the-art results: MAGNA achieves up to 5.7% relative error reduction over previous state-of-the-art on Cora, Citeseer, and Pubmed. MAGNA also obtains better performance on a large-scale Open Graph Benchmark dataset. On knowledge graph completion, MAGNA advances state-of-the-art on WN18RR and FB15k-237 across four metrics, with the largest gain of 7.1% in the metric of Hit at 1.

Furthermore, we show that MAGNA with just 3 layers and 6 hop wide attention per layer significantly out-performs GAT with 18 layers, even though both architectures have the same receptive field. Moreover, our ablation study reveals the synergistic effect of the essential components of MAGNA, including layer normalization and multi-hop diffused attention. We further observe that compared to GAT, the attention values learned by MAGNA have higher diversity, indicating the ability to better pay attention to important nodes.

6.2 MAGNA Architecture

We first discuss the background and explain the novel multi-hop attention diffusion module and the MAGNA architecture.

6.2.1 Preliminaries

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a given graph, where $\mathcal{V}$ is the set of N_n nodes, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of N_e edges connecting M pairs of nodes in $\mathcal{V}$. Each node $v \in \mathcal{V}$ and each edge $e \in \mathcal{E}$ are associated with their type mapping functions: $\phi : \mathcal{V} \rightarrow \mathcal{T}$ and $\psi : \mathcal{E} \rightarrow \mathcal{R}$. Here $\mathcal{T}$ and $\mathcal{R}$ denote the sets of node types (labels) and edge/relation types. Our framework supports learning on heterogeneous graphs with multiple elements in $\mathcal{R}$.

A general Graph Neural Network (GNN) approach learns an embedding that maps nodes and/or edge types into a continuous vector space. Let $\boldsymbol{X} \in \mathbb{R}^{N_n \times d_n}$ and $\boldsymbol{R} \in \mathbb{R}^{N_r \times d_r}$ be the node embedding and edge/relation type embedding, where $N_n = |\mathcal{V}|$, $N_r = |\mathcal{R}|$, d_n and d_r represent the embedding dimension of node and edge/relation types, each row $\boldsymbol{x}_i = \boldsymbol{X}[i\ :]$ represents the embedding of node v_i ($1 \leq i \leq N_n$), and $\boldsymbol{r}_j = \boldsymbol{R}[j\ :]$ represents the embedding of relation r_j

($1 \leq j \leq N_r$). MAGNA builds on GNNs, while bringing together the benefits of Graph Attention and Diffusion techniques.

6.2.2 Multi-hop Attention Diffusion

We first introduce attention diffusion to compute the multi-hop attention directly, which operates on the MAGNA's attention scores at each layer. The input to the attention diffusion operator is a set of triples (v_i, r_k, v_j), where v_i, v_j are nodes and r_k is the edge type. MAGNA first computes the attention scores on all edges. The attention diffusion module then computes the attention values between pairs of nodes that are not directly connected by an edge, based on the edge attention scores, via a diffusion process. The attention diffusion module can then be used as a component in MAGNA architecture, which we will further elaborate in Section 7.3.2.

Edge Attention Computation. At each layer l, a vector message is computed for each triple (v_i, r_k, v_j). To compute the representation of v_j at layer $l+1$, all messages from triples incident to v_j are aggregated into a single message, which is then used to update v_j^{l+1}.

In the first stage, the attention score s of an edge (v_i, r_k, v_j) is computed by the following:

$$s_{i,k,j}^{(l)} = \delta(\boldsymbol{v}_a^{(l)} \tanh(\boldsymbol{W}_h^{(l)} \boldsymbol{h}_{\boldsymbol{i}}^{(\boldsymbol{l})} \| \boldsymbol{W}_t^{(l)} \boldsymbol{h}_{\boldsymbol{j}}^{(\boldsymbol{l})} \| \boldsymbol{W}_r^{(l)} \boldsymbol{r}_k)) \tag{6.1}$$

where $\delta = \text{LeakyReLU}$, $\boldsymbol{W}_h^{(l)}, \boldsymbol{W}_t^{(l)} \in \mathbb{R}^{d^{(l)} \times d^{(l)}}$, $\boldsymbol{W}_r^{(l)} \in \mathbb{R}^{d^{(l)} \times d_r}$ and $\boldsymbol{v}_a^{(l)} \in \mathbb{R}^{1 \times 3d^{(l)}}$ are the trainable weights shared by l-th layer. $\boldsymbol{h}_i^{(l)} \in \mathbb{R}^{d^{(l)}}$ represents the embedding of node i at l-th layer, and $\boldsymbol{h}_i^{(0)} = \boldsymbol{x}_i$. $\boldsymbol{r}_k$ is the trainable relation embedding of the k-th relation type r_k ($1 \leq k \leq N_r$), and $a \| b$ denotes concatenation of embedding vectors a and b. For graphs with no relation type, we treat as a degenerate categorical distribution with 1 category[1].

Applying Eq. 6.1 on each edge of the graph $\mathcal{G}$, we obtain an attention score matrix $\boldsymbol{S}^{(l)}$:

$$\boldsymbol{S}_{i,j}^{(l)} = \begin{cases} s_{i,k,j}^{(l)}, & \text{if } (v_i, r_k, v_j) \text{ appears in } \mathcal{G} \\ -\infty, & \text{otherwise} \end{cases} \tag{6.2}$$

Subsequently we obtain the attention matrix $\boldsymbol{A}^{(l)}$ by performing row-wised softmax over the score matrix $\boldsymbol{S}^{(l)}$: $\boldsymbol{A}^{(l)} = \text{softmax}(\boldsymbol{S}^{(\boldsymbol{l})})$. $\boldsymbol{A}_{ij}^{(l)}$ denotes the attention value at layer l when aggregating message from node j to node i.

Attention Diffusion for Multi-hop Neighbors. In the second stage, we further enable attention between nodes that are not directly connected in the network. We achieve this via the following

[1] In this case, we can view that there is only one "pseudo" relation type (category), i.e., $N_r = 1$

attention diffusion procedure. The procedure computes the attention scores of multi-hop neighbors via graph diffusion based on the powers of the 1-hop attention matrix $\boldsymbol{A}$:

$$\mathcal{A} = \sum_{i=0}^{\infty} \theta_i \boldsymbol{A}^i \text{ where } \sum_{i=0}^{\infty} \theta_i = 1 \text{ and } \theta_i > 0 \tag{6.3}$$

where θ_i is the attention decay factor and $\theta_i > \theta_{i+1}$. The powers of attention matrix, A^i, give us the number of relation paths from node h to node t of length up to i, increasing the receptive field of the attention (Figure 6.1). Importantly, the mechanism allows the attention between two nodes to not only depend on their previous layer representations, but also taking into account of the paths between the nodes, effectively creating attention shortcuts between nodes that are not directly connected (Figure 6.1). Attention through each path is also weighted differently, depending on θ and the path length.

In our implementation we utilize the geometric distribution $\theta_i = \alpha(1-\alpha)^i$, where $\alpha \in (0, 1]$. The choice is based on the inductive bias that nodes further away should be weighted less in message aggregation, and nodes with different relation path lengths to the target node are sequentially weighted in an independent manner. In addition, notice that if we define $\theta_0 = \alpha \in (0, 1]$, $\boldsymbol{A}^0 = \boldsymbol{I}$, then Eq. 6.3 gives the Personalized Page Rank (PPR) procedure on the graph with the attention matrix $\boldsymbol{A}$ and teleport probability α. Hence the diffused attention weights, $\mathcal{A}_{ij}$, can be seen as the influence of node j to node i. We further elaborate the significance of this observation in Section 6.3.3.

We can also view $\mathcal{A}_{ij}$ as the attention value of node j to i since $\sum_{j=1}^{N_n} \mathcal{A}_{ij} = 1$.[2] We then define the *graph attention diffusion* based feature aggregation as

$$\text{AttDiff}(\mathcal{G}, \boldsymbol{H}^{(l)}, \Theta) = \mathcal{A}\boldsymbol{H}^{(l)}, \tag{6.4}$$

where Θ is the set of parameters for computing attention. Thanks to the diffusion process defined in Eq. 6.3, MAGNA uses the same number of parameters as if we were only computing attention between nodes connected via edges. This ensures runtime efficiency (refer to Appendix[3] A for complexity analysis) and model generalization.

Approximate Computation for Attention Diffusion. For large graphs computing the exact attention diffusion matrix $\mathcal{A}$ using Eq. 6.3 may be prohibitively expensive, due to computing the powers of the attention matrix (Klicpera et al., 2019a). To resolve this bottleneck, we proceed as follows: Let $\boldsymbol{H}^{(l)}$ be the input entity embedding of the l-th layer ($\boldsymbol{H}^{(0)} = \boldsymbol{X}$) and $\theta_i = \alpha(1-\alpha)^i$.

[2] Obtained by the definition $\boldsymbol{A}^{(l)} = \text{softmax}(\boldsymbol{S}^{(l)})$ and Eq. 6.3.

[3] Appendix can be found via https://arxiv.org/abs/2009.14332

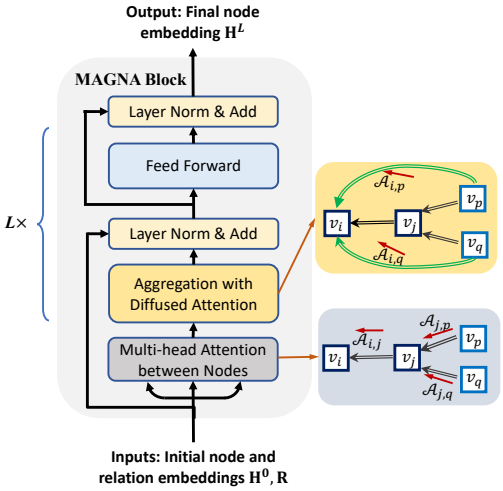


Figure 6.2: **MAGNA Architecture**. Each MAGNA block consists of attention computation, attention diffusion, layer normalization, feed forward layers, and 2 residual connections for each block. MAGNA blocks can be stacked to constitute a deep model. As illustrated on the right, context-dependent attention is achieved via the attention diffusion process. Here $v_i, v_j, v_p, v_q \in \mathcal{V}$ are nodes in the graph.

Since MAGNA only requires aggregation via $\mathcal{A}\boldsymbol{H}^{(l)}$, we can approximate $\mathcal{A}\boldsymbol{H}^{(l)}$ by defining a sequence $Z^{(K)}$ which converges to the true value of $\mathcal{A}\boldsymbol{H}^{(l)}$ (Proposition 2) as $K \to \infty$:

$$\boldsymbol{Z}^{(0)} = \boldsymbol{H}^{(l)}, \boldsymbol{Z}^{(k+1)} = (1-\alpha)\boldsymbol{A}\boldsymbol{Z}^{(k)} + \alpha\boldsymbol{Z}^{(0)}, \tag{6.5}$$

where $0 \leq k < K$.

Proposition 2 $\lim_{K\to\infty} Z^{(K)} = \mathcal{A}\boldsymbol{H}^{(l)}$

In the Appendix we give the proof which relies on the expansion of Eq. 6.5.

Using the above approximation, the complexity of attention computation with diffusion is still $O(|E|)$, with a constant factor corresponding to the number of hops K. In practice, we find that choosing the values of K such that $3 \leq K \leq 10$ results in good model performance. Many real-world graphs exhibit small-world property, in which case even a smaller value of K is sufficient. For graphs with larger diameter, we choose larger K, and lower the value of α.

6.2.3 Multi-hop Attention based GNN Architecture

Figure 6.2 provides an architecture overview of the MAGNA Block that can be stacked multiple times.

Multi-head Graph Attention Diffusion Layer. Multi-head attention (Vaswani et al., 2017; Veličković et al., 2018b) is used to allow the model to jointly attend to information from different representation sub-spaces at different viewpoints. In Eq. 6.6, the attention diffusion for each head i is computed separately with Eq. 6.4, and aggregated:

$$\begin{aligned} \hat{\boldsymbol{H}}^{(l)} &= \text{MultiHead}(\mathcal{G}, \tilde{\boldsymbol{H}}^{(l)}) = \left(\Big\|_{i=1}^{M} \text{head}_i \right) \boldsymbol{W}_o \\ \text{head}_i &= \text{AttDiff}(\mathcal{G}, \tilde{\boldsymbol{H}}^{(l)}, \Theta_i), \tilde{\boldsymbol{H}}^{(l)} = \text{LN}(\boldsymbol{H}^{(l)}), \end{aligned} \tag{6.6}$$

where $\|$ denotes concatenation and Θ_i are the parameters in Eq. 6.1 for the i-th head ($1 \leq i \leq M$), $\boldsymbol{W}_o$ represents a parameter matrix, and LN = LayerNorm. Since we calculate the attention diffusion in a recursive way in Eq. 6.5, we add layer normalization which helpful to stabilize the recurrent computation procedure (Ba et al., 2016).

Deep Aggregation. Moreover our MAGNA block contains a fully connected feed-forward sub-layer, which consists of a two-layer feed-forward network. We also add the layer normalization and residual connection in both sub-layers, allowing for a more expressive aggregation step for each block (Xiong et al., 2020):

$$\begin{aligned} \hat{\boldsymbol{H}}^{(l+1)} &= \hat{\boldsymbol{H}}^{(l)} + \boldsymbol{H}^{(l)} \\ \boldsymbol{H}^{(l+1)} &= \boldsymbol{W}_2^{(l)} \text{ReLU}\left(\boldsymbol{W}_1^{(l)} \text{LN}(\hat{\boldsymbol{H}}^{(l+1)}) \right) + \hat{\boldsymbol{H}}^{(l+1)} \end{aligned} \tag{6.7}$$

MAGNA generalizes GAT. MAGNA extends GAT via the diffusion process. The feature aggregation in GAT is $\boldsymbol{H}^{(l+1)} = \sigma(\boldsymbol{A}\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)})$, where σ represents the activation function. We can divide GAT layer into two components as follows:

$$\boldsymbol{H}^{(l+1)} = \underbrace{\sigma}_{(2)}(\underbrace{\boldsymbol{A}\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}}_{(1)}). \tag{6.8}$$

In component (1), MAGNA removes the restriction of attending to direct neighbors, without requiring additional parameters as $\mathcal{A}$ is induced from $\boldsymbol{A}$. For component (2) MAGNA uses layer normalization and deep aggregation to achieve higher expressive power compared to elu nonlinearity in GAT.

6.3 MAGNA Performance on Standard Graph Benchmarks

We evaluate MAGNA on two classical tasks[4]: (1) on node classification we achieve an average of 5.7% relative error reduction; (2) on knowledge graph completion we achieve 7.1% relative improvement in the Hit@1 metric.[5]

6.3.1 Task 1: Node Classification

Datasets. We employ four benchmark datasets for node classification: (1) standard citation network benchmarks Cora, Citeseer and Pubmed (Sen et al., 2008; Kipf and Welling, 2016a); and (2) a benchmark dataset ogbn-arxiv on 170k nodes and 1.2m edges from the Open Graph Benchmark (Weihua Hu, 2020). We follow the standard data splits for all datasets. Further information about these datasets is summarized in the Appendix.

Baselines. We compare against a comprehensive suite of state-of-the-art GNN methods including: GCNs (Kipf and Welling, 2016a), Chebyshev filter based GCNs (Defferrard et al., 2016b), DualGCN (Zhuang and Ma, 2018), JKNet (Xu et al., 2018b), LGCN (Gao et al., 2018), Diffusion-GCN (Diff-GCN) (Klicpera et al., 2019c), APPNP (Klicpera et al., 2019a), Graph U-Nets (g-U-Nets) (Gao and Ji, 2019), and GAT (Veličković et al., 2018b).

Experimental Setup. For datasets Cora, Citeseer and Pubmed, we use 6 MAGNA blocks with hidden dimension 512 and 8 attention heads. For the large-scale ogbn-arxiv dataset, we use 2 MAGNA blocks with hidden dimension 128 and 8 attention heads. Refer to Appendix for detailed description of all hyper-parameters and evaluation settings.

Results. MAGNA achieves the best on all datasets (Tables 6.1 and 6.2) [6], out-performing mutlihop baselines such as Diffusion GCN, APPNP and JKNet. The baseline performance and their embedding dimensions are from the previous papers. Appendix Table 6 further demonstrates that large 512 dimension embedding only benefits the expressive MAGNA, whereas GAT and Diffusion GCN performance degrades.

Ablation study. We report (Table 6.1) the model performance after removing each component of MAGNA (layer normalization, attention diffusion and feed forward layers) from every MAGNA layer. Note that the model is equivalent to GAT without these three components. We observe that diffusion and layer normalization play a crucial role in improving the node classification performance for all datasets. Since MAGNA computes attention diffusion in a recursive manner, layer

[4]All datasets are public. And our implementation is available at https://github.com/xjtuwgt/GNN-MAGNA

[5]Please see the definitions of these two tasks in Appendix.

[6]We also compared to GAT and Diffusion-GCN (with LayerNorm and feed-forward Layer) over random splits in Appendix.

	Models	Cora	Citeseer	Pubmed
Baselines	GCN (Kipf and Welling, 2016a)	81.5	70.3	79.0
	Cheby (Defferrard et al., 2016b)	81.2	69.8	74.4
	DualGCN (Zhuang and Ma, 2018)	83.5	72.6	80.0
	JKNet (Xu et al., 2018b)*	81.1	69.8	78.1
	LGCN (Gao et al., 2018)	83.3 ± 0.5	73.0 ± 0.6	79.5 ± 0.2
	Diff-GCN (Klicpera et al., 2019c)	83.6 ± 0.2	73.4 ± 0.3	79.6 ± 0.4
	APPNP (Klicpera et al., 2019a)	84.3 ± 0.2	71.1 ± 0.4	79.7 ± 0.3
	g-U-Nets (Gao and Ji, 2019)	84.4 ± 0.6	73.2 ± 0.5	79.6 ± 0.2
	GAT (Veličković et al., 2018b)	83.0 ± 0.7	72.5 ± 0.7	79.0 ± 0.3
Abl.	No LayerNorm	83.8 ± 0.6	71.1 ± 0.5	79.8 ± 0.2
	No Diffusion	83.0 ± 0.4	71.6 ± 0.4	79.3 ± 0.3
	No Feed-Forward$^\diamond$	84.9 ± 0.4	72.2 ± 0.3	80.9 ± 0.3
	No (LayerNorm + Feed-Forward)	84.3 ± 0.6	72.6 ± 0.4	79.6 ± 0.4
	MAGNA	**85.4** ± 0.6	**73.7** ± 0.5	**81.4** ± 0.2

* : based on the implementation in https://github.com/DropEdge/DropEdge;
$^\diamond$: replace the feed forward layer with *elu* used in GAT.

Table 6.1: Node classification accuracy on Cora, Citeseer, Pubmed. MAGNA achieves state-of-the-art.

Data	GCN (Kipf and Welling, 2016a)	GraphSAGE (Hamilton et al., 2017c)	JKNet (Xu et al., 2018b)	DAGNN (Liu et al., 2020)	GaAN (Zhang et al., 2018a)	**MAGNA**
ogbn-arxiv	71.74 ± 0.29	71.49 ± 0.27	72.19 ± 0.21	72.09 ± 0.25	71.97 ± 0.24	**72.76** ± 0.14

Table 6.2: Node classification accuracy on the OGB Arxiv dataset.

normalization is crucial in ensuring training stability (Ba et al., 2016). Meanwhile, comparing to GAT (see the next-to-last row of Table 6.1), attention diffusion allows multi-hop attention in every layer to benefit node classification.

6.3.2 Task 2: Knowledge Graph Completion

Datasets. We evaluate MAGNA on standard benchmark knowledge graphs: WN18RR (Dettmers et al., 2018) and FB15K-237 (Toutanova and Chen, 2015). See the statistics of these KGs in Appendix.

Baselines. We compare MAGNA with state-of-the-art baselines, including (1) translational distance based models: TransE (Bordes et al., 2013) and its latest extension RotatE (Sun et al., 2019b), OTE (Tang et al., 2020) and ROTH (Chami et al., 2020); (2) semantic matching based models: ComplEx (Trouillon et al., 2016), QuatE (Zhang et al., 2019), CoKE (Wang et al., 2019b), ConvE (Dettmers et al., 2018), DistMult (Yang et al., 2015), TuckER (Balazevic et al., 2019) and AutoSF (Zhang et al., 2020); (3) GNN-based models: R-GCN (Schlichtkrull et al., 2018a), SACN (Shang et al., 2019) and A2N (Bansal et al., 2019).

Models	WN18RR					FB15k-237				
	MR	MRR	H@1	H@3	H@10	MR	MRR	H@1	H@3	H@10
TransE (Bordes et al., 2013)	3384	.226	-	-	.501	357	.294	-	-	.465
RotatE (Sun et al., 2019b)	3340	.476	.428	.492	.571	177	.338	.241	.375	.533
OTE (Tang et al., 2020)	-	.491	.442	.511	.583	-	.361	.267	.396	.550
ROTH (Chami et al., 2020)	-	.496	.449	.514	.586	-	.344	.246	.380	.535
ComplEx (Trouillon et al., 2016)	5261	.44	.41	.46	.51	339	.247	.158	.275	.428
QuatE (Zhang et al., 2019)	2314	.488	.438	.508	.582	-	.366	.271	.401	.556
CoKE (Wang et al., 2019b)	-	.475	.437	.490	.552	-	.361	.269	.398	.547
ConvE (Dettmers et al., 2018)	4187	.43	.40	.44	.52	244	.325	.237	.356	.501
DistMult (Yang et al., 2015)	5110	.43	.39	.44	.49	254	.241	.155	.263	.419
TuckER (Balazevic et al., 2019)	-	.470	.443	.482	.526	-	.358	.266	.392	.544
AutoSF (Zhang et al., 2020)	-	.490	.451	-	.567	-	.360	.267	-	.552
R-GCN (Schlichtkrull et al., 2018a)	-	-	-	-	-	-	.249	.151	.264	.417
SACN (Shang et al., 2019)	-	.47	.43	.48	.54	-	.35	.26	.39	.54
A2N (Bansal et al., 2019)	-	.45	.42	.46	.51	-	.317	.232	.348	.486
MAGNA + DistMult	2545	.502	.459	.519	.589	138	.369	.275	.409	.563

Table 6.3: KG Completion on WN18RR and FB15k-237. MAGNA achieves state of the art.

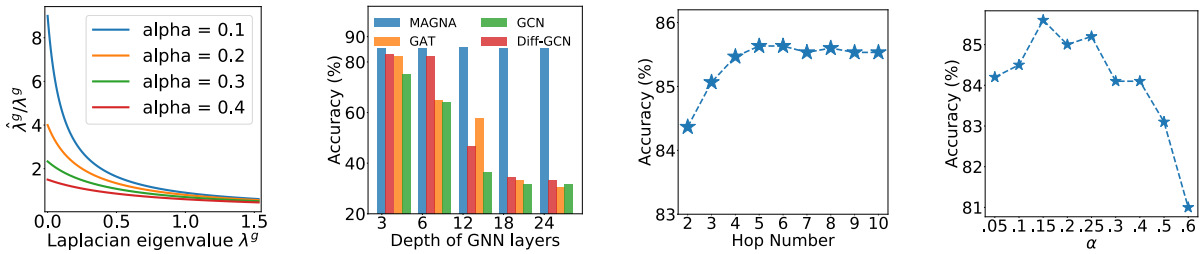


Figure 6.3: Analysis of MAGNA on Cora. (a) Influence of MAGNA on Laplacian eigenvalues. (b) Effect of depth on performance. (c) Effect of hop number K on performance. (d) Effect of teleport probability α.

Experimental Setup. We use the multi-layer MAGNA as encoder for both FB15k-237 and WN18RR. We randomly initialize the entity embedding and relation embedding as the input of the encoders, and set the dimensionality of the initialized entity/relation vector as 100 used in DistMult (Yang et al., 2015). We select other MAGNA model hype-parameters, including number of layers, hidden dimension, head number, top-k, learning rate, hop number, teleport probability α and dropout ratios (see the settings of these parameter in Appendix), by a random search during the training.

Training procedure. We use the standard training procedure used in previous KG embedding models (Balazevic et al., 2019; Dettmers et al., 2018) (Appendix for details). We follow an encoder-decoder framework: The encoder applies the proposed MAGNA model to compute the entity embeddings. The decoder makes link prediction given the embeddings. To show the power of MAGNA, we employ a simple decoder DistMult (Yang et al., 2015).

Evaluation. We use the standard split for the benchmarks, and the standard testing procedure of predicting tail (head) entity given the head (tail) entity and relation type. We exactly follow the evaluation used by all previous works, namely the Mean Reciprocal Rank (MRR), Mean Rank (MR), and hit rate at K (H@K). See Appendix for a detailed description of this standard setup.
Results. MAGNA achieves new state-of-the-art in knowledge graph completion on all four metrics (Table 6.3). MAGNA compares favourably to both the most recent shallow embedding methods (QuatE), and deep embedding methods (SACN). Note that with the same decoder (DistMult), MAGNA using its own embeddings achieves drastic improvements over using the corresponding DistMult embeddings.

6.3.3 MAGNA Model Analysis

Here we present (1) spectral analysis results, (2) robustness to hyper-parameter changes, and (3) attention distribution analysis to show the strengths of MAGNA.
Spectral Analysis: Why MAGNA works for node classification. We compute the eigenvalues of the graph Laplacian of the attention matrix $\mathbf{A}$, $\hat{\lambda}_i^g$, and compare to that of the diffused matrix $\mathcal{A}$, λ_i^g. Figure 6.3 (a) shows the ratio $\hat{\lambda}_i^g/\lambda_i^g$ on the Cora dataset. Low eigenvalues corresponding to large-scale structure in the graph are amplified (up to a factor of 8), while high eigenvalues corresponding to eigenvectors with noisy information are suppressed (Klicpera et al., 2019c).
MAGNA Model Depth. Here we conduct experiments by varying the number of GCN, Diffusion-GCN (PPR based) GAT and our MAGNA layers to be 3, 6, 12, 18 and 24 for node classification on Cora. Results in Fig. 6.3 (b) show that deep GCN, Diffusion-GCN and GAT (even with residual connection) suffer from degrading performance, due to the over-smoothing problem (Li et al., 2018a; Wang et al., 2019a). In contrast, the MAGNA model achieves consistent best results even with 18 layers, making deep MAGNA model robust and expressive. Notice that GAT with 18 layers cannot out-perform MAGNA with 3 layers and K=6 hops, although they have the same receptive field.
Effect of K and α. Figs. 6.3 (c) and (d) report the effect of hop number K and teleport probability α on model performance. We observe significant increase in performance when considering multi-hop neighbors information ($K > 1$). However, increasing the hop number K has a diminishing returns, for $K \geq 6$. Moreover, we find that the optimal K is correlated with the largest node average shortest path distance (e.g., 5.27 for Cora). This provides a guideline for choosing the best K.

We also observe that the accuracy drops significantly for larger $\alpha > 0.25$. This is because small α increases the low-pass effect (Fig. 6.3 (a)). However, α being too small causes the model to only

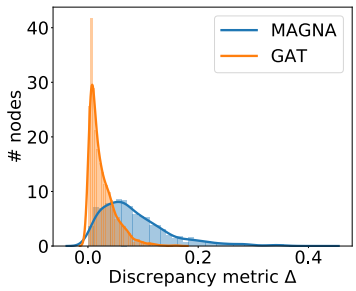


Figure 6.4: Attention weight distribution on Cora.

focus on the most large-scale graph structure and have lower performance.

Attention Distribution. Last we also analyze the learned attention scores of GAT and MAGNA. We first define a discrepancy metric over the attention matrix $\boldsymbol{A}$ for node v_i as $\Delta_i = \frac{\|\boldsymbol{A}_{[i,:]} - U_i\|}{\text{degree}(v_i)}$ (Shanthamallu et al., 2020), where U_i is the uniform distribution score for the node v_i. Δ_i gives a measure of how much the learnt attention deviates from an uninformative uniform distribution. Large Δ_i indicates more meaningful attention scores. Fig. 6.4 shows the distribution of the discrepancy metric of the attention matrix of the 1st head w.r.t. the first layer of MAGNA and GAT. Observe that attention scores learned in MAGNA have much larger discrepancy. This shows that MAGNA is more powerful than GAT in distinguishing important nodes and assigns attention scores accordingly.

Part III

Scalable Graph Neural Networks: Applications

Chapter 7

GNN for Web-Scale Recommender Systems

In Part III, the GraphSAGE framework and its architectural improvements is demonstrated in challenging real-world applications. These applications range from diverse domains including social networks, physical simulations and biology, and are characterized by the large-scale dataset, rich node and graph features, hierarchical structures, as well as complex interactions.

In this chapter, we showcase PinSage, a GNN model for recommendation systems with hundreds of millions of users and billions of items. Furthermore, the learned representation of PinSage is proved to be useful many other applications such as user behavior prediction and item embedding service. The system is the first web-scale GNN that was deployed in industry, and has inspired multiple large-scale GNN systems for commercial applications.

7.1 Introduction

Deep learning methods have an increasingly critical role in recommender system applications, being used to learn useful low-dimensional embeddings of images, text, and even individual users (Covington et al., 2016; den Oord et al., 2013). The representations learned using deep models can be used to complement, or even replace, traditional recommendation algorithms like collaborative filtering. and these learned representations have high utility because they can be re-used in various recommendation tasks. For example, item embeddings learned using a deep model can be used for item-item recommendation and also to recommended themed collections (*e.g.*, playlists, or "feed" content).

Recent years have seen significant developments in this space—especially the development of new deep learning methods that are capable of learning on graph-structured data, which is fundamental for recommendation applications (*e.g.*, to exploit user-to-item interaction graphs as

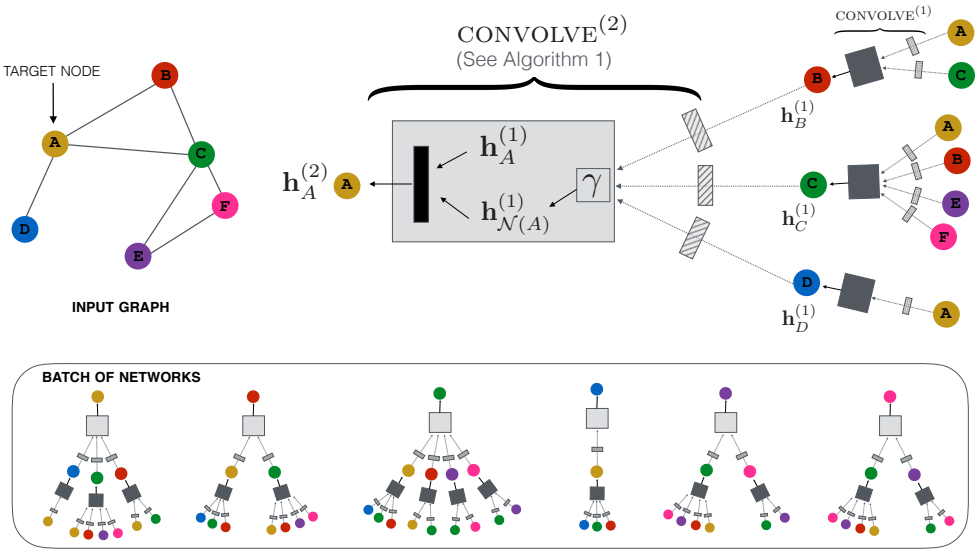


Figure 7.1: Overview of our model architecture using depth-2 convolutions (best viewed in color). Left: A small example input graph. Right: The 2-layer neural network that computes the embedding $\mathbf{h}_A^{(2)}$ of node A using the previous-layer representation, $\mathbf{h}_A^{(1)}$, of node A and that of its neighborhood $\mathcal{N}(A)$ (nodes B, C, D). (However, the notion of neighborhood is general and not all neighbors need to be included (Section 7.3.2).) Bottom: The neural networks that compute embeddings of each node of the input graph. While neural networks differ from node to node they all share the same set of parameters (*i.e.*, the parameters of the CONVOLVE$^{(1)}$ and CONVOLVE$^{(2)}$ functions; Algorithm 1). Boxes with the same shading patterns share parameters; γ denotes an importance pooling function; and thin rectangular boxes denote densely-connected multi-layer neural networks.

well as social graphs) (van den Berg et al., 2017; Bronstein et al., 2017; Hamilton et al., 2017b; Kipf and Welling, 2016a; Monti et al., 2017; You et al., 2018b).

Most prominent among these recent advancements is the success of deep learning architectures known as Graph Convolutional Networks (GCNs) (van den Berg et al., 2017; Hamilton et al., 2017b; Kipf and Welling, 2016a; Monti et al., 2017). The core idea behind GCNs is to learn how to iteratively aggregate feature information from local graph neighborhoods using neural networks (Figure 7.1). Here a single "convolution" operation transforms and aggregates feature information from a node's one-hop graph neighborhood, and by stacking multiple such convolutions information can be propagated across far reaches of a graph. Unlike purely content-based deep models (*e.g.*, recurrent neural networks (Bansal et al., 2016)), GCNs leverage both content information as well as graph structure. GCN-based methods have set a new standard on countless recommender system benchmarks (see (Hamilton et al., 2017b) for a survey). However, these gains on benchmark tasks have yet to be translated to gains in real-world production environments.

The main challenge is to scale both the training as well as inference of GCN-based node embeddings to graphs with billions of nodes and tens of billions of edges. Scaling up GCNs is difficult

because many of the core assumptions underlying their design are violated when working in a big data environment. For example, all existing GCN-based recommender systems require operating on the full graph Laplacian during training—an assumption that is infeasible when the underlying graph has billions of nodes and whose structure is constantly evolving.

Present work. Here we present a highly-scalable GCN framework that we have developed and deployed in production at Pinterest. Our framework, a random-walk-based GCN named PinSage, operates on a massive graph with 3 billion nodes and 18 billion edges—a graph that is $10,000\times$ larger than typical applications of GCNs. PinSage leverages several key insights to drastically improve the scalability of GCNs:

- **On-the-fly convolutions:** Traditional GCN algorithms perform graph convolutions by multiplying feature matrices by powers of the full graph Laplacian. In contrast, our PinSage algorithm performs efficient, localized convolutions by sampling the neighborhood around a node and dynamically constructing a computation graph from this sampled neighborhood. These dynamically constructed computation graphs (Fig. 7.1) specify how to perform a localized convolution around a particular node, and alleviate the need to operate on the entire graph during training.
- **Producer-consumer minibatch construction:** We develop a producer-consumer architecture for constructing minibatches that ensures maximal GPU utilization during model training. A large-memory, CPU-bound producer efficiently samples node network neighborhoods and fetches the necessary features to define local convolutions, while a GPU-bound TensorFlow model consumes these pre-defined computation graphs to efficiently run stochastic gradient decent.
- **Efficient MapReduce inference:** Given a fully-trained GCN model, we design an efficient MapReduce pipeline that can distribute the trained model to generate embeddings for billions of nodes, while minimizing repeated computations.

In addition to these fundamental advancements in scalability, we also introduce new training techniques and algorithmic innovations. These innovations improve the quality of the representations learned by PinSage, leading significant performance gains in downstream recommender system tasks:

- **Constructing convolutions via random walks:** Taking full neighborhoods of nodes to perform convolutions (Fig. 7.1) would result in huge computation graphs, so we resort to sampling. However, random sampling is suboptimal, and we develop a new technique using short random walks to sample the computation graph. An additional benefit is that each node now has an importance score, which we use in the pooling/aggregation step.
- **Importance pooling:** A core component of graph convolutions is the aggregation of feature

information from local neighborhoods in the graph. We introduce a method to weigh the importance of node features in this aggregation based upon random-walk similarity measures, leading to a 46% performance gain in offline evaluation metrics.

- **Curriculum training:** We design a curriculum training scheme, where the algorithm is fed harder-and-harder examples during training, resulting in a 12% performance gain.

We have deployed PinSage for a variety of recommendation tasks at Pinterest, a popular content discovery and curation application where users interact with *pins*, which are visual bookmarks to online content (*e.g.*, recipes they want to cook, or clothes they want to purchase). Users organize these pins into *boards*, which contain collections of similar pins. Altogether, Pinterest is the world's largest user-curated graph of images, with over 2 billion unique pins collected into over 1 billion boards.

Through extensive offline metrics, controlled user studies, and A/B tests, we show that our approach achieves state-of-the-art performance compared to other scalable deep content-based recommendation algorithms, in both an item-item recommendation task (*i.e.*, related-pin recommendation), as well as a "homefeed" recommendation task. In offline ranking metrics we improve over the best performing baseline by more than 40%, in head-to-head human evaluations our recommendations are preferred about 60% of the time, and the A/B tests show 30% to 100% improvements in user engagement across various settings.

To our knowledge, this is the largest-ever application of deep graph embeddings and paves the way for new generation of recommendation systems based on graph convolutional architectures.

7.2 Related work

Our work builds upon a number of recent advancements in deep learning methods for graph-structured data.

The notion of neural networks for graph data was first outlined in Gori et al. (2005) (Gori et al., 2005) and further elaborated on in Scarselli et al. (2009) (Scarselli et al., 2009b). However, these initial approaches to deep learning on graphs required running expensive neural "message-passing" algorithms to convergence and were prohibitively expensive on large graphs. Some limitations were addressed by Gated Graph Sequence Neural Networks (Li et al., 2015)—which employs modern recurrent neural architectures—but the approach remains computationally expensive and has mainly been used on graphs with $<10,000$ nodes.

More recently, there has been a surge of methods that rely on the notion of "graph convolutions" or Graph Convolutional Networks (GCNs). This approach originated with the work of Bruna et

al. (2013), which developed a version of graph convolutions based on spectral graph thery (Bruna et al., 2014b). Following on this work, a number of authors proposed improvements, extensions, and approximations of these spectral convolutions (van den Berg et al., 2017; Bronstein et al., 2017; Dai et al., 2016a; Defferrard et al., 2016b; Duvenaud et al., 2015c; Hamilton et al., 2017c; Kipf and Welling, 2016a; Monti et al., 2017; Zitnik et al., 2018b), leading to new state-of-the-art results on benchmarks such as node classification, link prediction, as well as recommender system tasks (*e.g.*, the MovieLens benchmark (Monti et al., 2017)). These approaches have consistently outperformed techniques based upon matrix factorization or random walks (*e.g.*, node2vec (Grover and Leskovec, 2016) and DeepWalk (Perozzi et al., 2014)), and their success has led to a surge of interest in applying GCN-based methods to applications ranging from recommender systems (Monti et al., 2017) to drug design (Kearnes et al., 2016; Zitnik et al., 2018b). Hamilton et al. (2017b) (Hamilton et al., 2017b) and Bronstein et al. (2017) (Bronstein et al., 2017) provide comprehensive surveys of recent advancements.

However, despite the successes of GCN algorithms, no previous works have managed to apply them to production-scale data with billions of nodes and edges—a limitation that is primarily due to the fact that traditional GCN methods require operating on the entire graph Laplacian during training. Here we fill this gap and show that GCNs can be scaled to operate in a production-scale recommender system setting involving billions of nodes/items. Our work also demonstrates the substantial impact that GCNs have on recommendation performance in a real-world environment.

In terms of algorithm design, our work is most closely related to Hamilton et al. (2017a)'s GraphSAGE algorithm (Hamilton et al., 2017c) and the closely related follow-up work of Chen et al. (2018) (Chen et al., 2018a). GraphSAGE is an inductive variant of GCNs that we modify to avoid operating on the entire graph Laplacian. We fundamentally improve upon GraphSAGE by removing the limitation that the whole graph be stored in GPU memory, using low-latency random walks to sample graph neighborhoods in a producer-consumer architecture. We also introduce a number of new training techniques to improve performance and a MapReduce inference pipeline to scale up to graphs with billions of nodes.

Lastly, also note that graph embedding methods like node2vec (Grover and Leskovec, 2016) and DeepWalk (Perozzi et al., 2014) cannot be applied here. First, these are unsupervised methods. Second, they cannot include node feature information. Third, they directly learn embeddings of nodes and thus the number of model parameters is linear with the size of the graph, which is prohibitive for our setting.

7.3 Method

In this section, we describe the technical details of the PinSage architecture and training, as well as a MapReduce pipeline to efficiently generate embeddings using a trained PinSage model.

The key computational workhorse of our approach is the notion of localized graph convolutions.[1] To generate the embedding for a node (*i.e.*, an item), we apply multiple convolutional modules that aggregate feature information (*e.g.*, visual, textual features) from the node's local graph neighborhood (Figure 7.1). Each module learns how to aggregate information from a small graph neighborhood, and by stacking multiple such modules, our approach can gain information about the local network topology. Importantly, parameters of these localized convolutional modules are shared across all nodes, making the parameter complexity of our approach independent of the input graph size.

7.3.1 Problem Setup

Pinterest is a content discovery application where users interact with *pins*, which are visual bookmarks to online content (*e.g.*, recipes they want to cook, or clothes they want to purchase). Users organize these pins into *boards*, which contain collections of pins that the user deems to be thematically related. Altogether, the Pinterest graph contains 2 billion pins, 1 billion boards, and over 18 billion edges (*i.e.*, memberships of pins to their corresponding boards).

Our task is to generate high-quality embeddings or representations of pins that can be used for recommendation (*e.g.*, via nearest-neighbor lookup for related pin recommendation, or for use in a downstream re-ranking system). In order to learn these embeddings, we model the Pinterest environment as a bipartite graph consisting of nodes in two disjoint sets, $\mathcal{I}$ (containing pins) and $\mathcal{C}$ (containing boards). Note, however, that our approach is also naturally generalizable, with $\mathcal{I}$ being viewed as a set of items and $\mathcal{C}$ as a set of user-defined contexts or collections.

In addition to the graph structure, we also assume that the pins/items $u \in \mathcal{I}$ are associated with real-valued attributes, $x_u \in \mathbb{R}^d$. In general, these attributes may specify metadata or content information about an item, and in the case of Pinterest, we have that pins are associated with both rich text and image features. Our goal is to leverage both these input attributes as well as the structure of the bipartite graph to generate high-quality embeddings. These embeddings are then used for recommender system candidate generation via nearest neighbor lookup (*i.e.*, given a pin,

[1] Following a number of recent works (*e.g.*, (Duvenaud et al., 2015c; Kearnes et al., 2016)) we use the term "convolutional" to refer to a module that aggregates information from a local graph region and to denote the fact that parameters are shared between spatially distinct applications of this module; however, the architecture we employ does not directly approximate a spectral graph convolution (though they are intimately related) (Bronstein et al., 2017).

find related pins) or as features in machine learning systems for ranking the candidates.

For notational convenience and generality, when we describe the PinSage algorithm, we simply refer to the node set of the full graph with $\mathcal{V} = \mathcal{I} \cup \mathcal{C}$ and do not explicitly distinguish between pin and board nodes (unless strictly necessary), using the more general term "node" whenever possible.

7.3.2 Model Architecture

We use localized convolutional modules to generate embeddings for nodes. We start with input node features and then learn neural networks that transform and aggregate features over the graph to compute the node embeddings (Figure 7.1).

Forward propagation algorithm. We consider the task of generating an embedding, $\mathbf{z}_u$ for a node u, which depends on the node's input features and the graph structure around this node.

Algorithm 2: CONVOLVE

Input : Current embedding $\mathbf{z}_u$ for node u; set of neighbor embeddings $\{\mathbf{z}_v | v \in \mathcal{N}(u)\}$, set of neighbor weights $\boldsymbol{\alpha}$; symmetric vector function $\gamma(\cdot)$

Output: New embedding $\mathbf{z}_u^{\text{NEW}}$ for node u

1 $\mathbf{n}_u \leftarrow \gamma\left(\{\text{ReLU}\left(\mathbf{Q}\mathbf{h}_v + \mathbf{q}\right) \mid v \in \mathcal{N}(u)\}, \boldsymbol{\alpha}\right)$;

2 $\mathbf{z}_u^{\text{NEW}} \leftarrow \text{ReLU}\left(\mathbf{W} \cdot \text{CONCAT}(\mathbf{z}_u, \mathbf{n}_u) + \mathbf{w}\right)$;

3 $\mathbf{z}_u^{\text{NEW}} \leftarrow \mathbf{z}_u^{\text{NEW}} / \|\mathbf{z}_u^{\text{NEW}}\|_2$

The core of our PinSage algorithm is a localized convolution operation, where we learn how to aggregate information from u's neighborhood (Figure 7.1). This procedure is detailed in Algorithm 2 CONVOLVE. The basic idea is that we transform the representations $\mathbf{z}_v, \forall v \in \mathcal{N}(u)$ of u's neighbors through a dense neural network and then apply a aggregator/pooling fuction (*e.g.*, a element-wise mean or weighted sum, denoted as γ) on the resulting set of vectors (Line 1). This aggregation step provides a vector representation, $\mathbf{n}_u$, of u's local neighborhood, $\mathcal{N}(u)$. We then concatenate the aggregated neighborhood vector $\mathbf{n}_u$ with u's current representation $\mathbf{h}_u$ and transform the concatenated vector through another dense neural network layer (Line 2). Empirically we observe significant performance gains when using concatenation operation instead of the average operation as in (Kipf and Welling, 2016a). Additionally, the normalization in Line 3 makes training more stable, and it is more efficient to perform approximate nearest neighbor search for normalized embeddings (Section 7.3.5). The output of the algorithm is a representation of u that incorporates both information about itself and its local graph neighborhood.

Importance-based neighborhoods. An important innovation in our approach is how we define

node neighborhoods $\mathcal{N}(u)$, *i.e.*, how we select the set of neighbors to convolve over in Algorithm 2. Whereas previous GCN approaches simply examine k-hop graph neighborhoods, in PinSage we define importance-based neighborhoods, where the neighborhood of a node u is defined as the T nodes that exert the most influence on node u. Concretely, we simulate random walks starting from node u and compute the L_1-normalized visit count of nodes visited by the random walk (Eksombatchai et al., 2018).[2] The neighborhood of u is then defined as the top T nodes with the highest normalized visit counts with respect to node u.

The advantages of this importance-based neighborhood definition are two-fold. First, selecting a fixed number of nodes to aggregate from allows us to control the memory footprint of the algorithm during training (Hamilton et al., 2017c). Second, it allows Algorithm 2 to take into account the importance of neighbors when aggregating the vector representations of neighbors. In particular, we implement γ in Algorithm 2 as a weighted-mean, with weights defined according to the L_1 normalized visit counts. We refer to this new approach as *importance pooling*.

Stacking convolutions. Each time we apply the CONVOLVE operation (Algorithm 2) we get a new representation for a node, and we can stack multiple such convolutions on top of each other in order to gain more information about the local graph structure around node u. In particular, we use multiple layers of convolutions, where the inputs to the convolutions at layer k depend on the representations output from layer $k-1$ (Figure 7.1) and where the initial (*i.e.*, "layer 0") representations are equal to the input node features. Note that the model parameters in Algorithm 2 ($\mathbf{Q}$, $\mathbf{q}$, $\mathbf{W}$, and $\mathbf{w}$) are shared across the nodes but differ between layers.

Algorithm 3 details how stacked convolutions generate embeddings for a minibatch set of nodes, $\mathcal{M}$. We first compute the neighborhoods of each node and then apply K convolutional iterations to generate the layer-K representations of the target nodes. The output of the final convolutional layer is then fed through a fully-connected neural network to generate the final output embeddings $\mathbf{z}_u, \forall u \in \mathcal{M}$.

The full set of parameters of our model which we then learn is: the weight and bias parameters for each convolutional layer ($\mathbf{Q}^{(k)}, \mathbf{q}^{(k)}, \mathbf{W}^{(k)}, \mathbf{w}^{(k)}, \forall k \in \{1, ..., K\}$) as well as the parameters of the final dense neural network layer, $\mathbf{G}_1$, $\mathbf{G}_2$, and $\mathbf{g}$. The output dimension of Line 1 in Algorithm 2 (*i.e.*, the column-space dimension of $\mathbf{Q}$) is set to be m at all layers. For simplicity, we set the output dimension of all convolutional layers (*i.e.*, the output at Line 3 of Algorithm 2) to be equal, and we denote this size parameter by d. The final output dimension of the model (after applying line 18 of Algorithm 3) is also set to be d.

[2] In the limit of infinite simulations, the normalized counts approximate the Personalized PageRank scores with respect to u.

Algorithm 3: MINIBATCH

Input : Set of nodes $\mathcal{M} \subset \mathcal{V}$; depth parameter K; neighborhood function $\mathcal{N} : \mathcal{V} \rightarrow 2^{\mathcal{V}}$
Output: Embeddings $\mathbf{z}_u, \forall u \in \mathcal{M}$

/* Sampling neighborhoods of minibatch nodes. */
$\mathcal{S}^{(K)} \leftarrow \mathcal{M}$;
for $k = K, ..., 1$ **do**
  $\mathcal{S}^{(k-1)} \leftarrow \mathcal{S}^{(k)}$;
  for $u \in S^{(k)}$ **do**
    $\mathcal{S}^{(k-1)} \leftarrow \mathcal{S}^{(k-1)} \cup \mathcal{N}(u)$;
  end
end
/* Generating embeddings */
$\mathbf{h}_u^{(0)} \leftarrow \mathbf{x}_u, \forall u \in \mathcal{S}^{(0)}$;
for $k = 1, ..., K$ **do**
  for $u \in \mathcal{S}^{(k)}$ **do**
    $\mathcal{H} \leftarrow \left\{\mathbf{h}_v^{(k-1)}, \forall v \in \mathcal{N}(u)\right\}$;
    $\mathbf{h}_u^{(k)} \leftarrow \text{CONVOLVE}^{(k)}\left(\mathbf{h}_u^{(k-1)}, \mathcal{H}\right)$
  end
end
for $u \in \mathcal{M}$ **do**
  $\mathbf{z}_u \leftarrow \mathbf{G}_2 \cdot \text{ReLU}\left(\mathbf{G}_1 \mathbf{h}_u^{(K)} + \mathbf{g}\right)$
end

7.3.3 Model Training

We train PinSage in a supervised fashion using a max-margin ranking loss. In this setup, we assume that we have access to a set of labeled pairs of items $\mathcal{L}$, where the pairs in the set, $(q, i) \in \mathcal{L}$, are assumed to be related—*i.e.*, we assume that if $(q, i) \in \mathcal{L}$ then item i is a good recommendation candidate for query item q. The goal of the training phase is to optimize the PinSage parameters so that the output embeddings of pairs $(q, i) \in \mathcal{L}$ in the labeled set are close together.

We first describe our margin-based loss function in detail. Following this, we give an overview of several techniques we developed that lead to the computation efficiency and fast convergence rate of PinSage, allowing us to train on billion node graphs and billions training examples. And finally, we describe our curriculum-training scheme, which improves the overall quality of the recommendations.

Loss function. In order to train the parameters of the model, we use a max-margin-based loss function. The basic idea is that we want to maximize the inner product of positive examples, *i.e.*,

the embedding of the query item and the corresponding related item. At the same time we want to ensure that the inner product of negative examples—*i.e.*, the inner product between the embedding of the query item and an unrelated item—is smaller than that of the positive sample by some pre-defined margin. The loss function for a single pair of node embeddings $(\mathbf{z}_q, \mathbf{z}_i) : (q, i) \in \mathcal{L}$ is thus

$$J_{\mathcal{G}}(\mathbf{z}_q \mathbf{z}_i) = \mathbb{E}_{n_k \sim P_n(q)} \max\{0, \mathbf{z}_q \cdot \mathbf{z}_{n_k} - \mathbf{z}_q \cdot \mathbf{z}_i + \Delta\}, \tag{7.1}$$

where $P_n(q)$ denotes the distribution of negative examples for item q, and Δ denotes the margin hyper-parameter. We shall explain the sampling of negative samples below.

Multi-GPU training with large minibatches. To make full use of multiple GPUs on a single machine for training, we run the forward and backward propagation in a multi-tower fashion. With multiple GPUs, we first divide each minibatch (Figure 7.1 bottom) into equal-sized portions. Each GPU takes one portion of the minibatch and performs the computations using the same set of parameters. After backward propagation, the gradients for each parameter across all GPUs are aggregated together, and a single step of synchronous SGD is performed. Due to the need to train on extremely large number of examples (on the scale of billions), we run our system with large batch sizes, ranging from 512 to 4096.

We use techniques similar to those proposed by Goyal *et al.* (Goyal et al., 2017) to ensure fast convergence and maintain training and generalization accuracy when dealing with large batch sizes. We use a gradual warmup procedure that increases learning rate from small to a peak value in the first epoch according to the linear scaling rule. Afterwards the learning rate is decreased exponentially.

Producer-consumer minibatch construction. During training, the adjacency list and the feature matrix for billions of nodes are placed in CPU memory due to their large size. However, during the CONVOLVE step of PinSage, each GPU process needs access to the neighborhood and feature information of nodes in the neighborhood. Accessing the data in CPU memory from GPU is not efficient. To solve this problem, we use a *re-indexing* technique to create a sub-graph $G' = (V', E')$ containing nodes and their neighborhood, which will be involved in the computation of the current minibatch. A small feature matrix containing only node features relevant to computation of the current minibatch is also extracted such that the order is consistent with the index of nodes in G'. The adjacency list of G' and the small feature matrix are fed into GPUs at the start of each minibatch iteration, so that no communication between the GPU and CPU is needed during the CONVOLVE step, greatly improving GPU utilization.

The training procedure has alternating usage of CPUs and GPUs. The model computations are in GPUs, whereas extracting features, re-indexing, and negative sampling are computed on CPUs.

In addition to parallelizing GPU computation with multi-tower training, and CPU computation using OpenMP (OpenMP Architecture Review Board, 2015), we design a producer-consumer pattern to run GPU computation at the current iteration and CPU computation at the next iteration in parallel. This further reduces the training time by almost a half.

Sampling negative items. Negative sampling is used in our loss function (Equation 7.1) as an approximation of the normalization factor of edge likelihood (Mikolov et al., 2013). To improve efficiency when training with large batch sizes, we sample a set of 500 negative items to be shared by all training examples in each minibatch. This drastically saves the number of embeddings that need to be computed during each training step, compared to running negative sampling for each node independently. Empirically, we do not observe a difference between the performance of the two sampling schemes.

In the simplest case, we could just uniformly sample negative examples from the entire set of items. However, ensuring that the inner product of the positive example (pair of items (q, i)) is larger than that of the q and each of the 500 negative items is too "easy" and does not provide fine enough "resolution" for the system to learn. In particular, our recommendation algorithm should be capable of finding 1,000 most relevant items to q among the catalog of over 2 billion items. In other words, our model should be able to distinguish/identify 1 item out of 2 million items. But with 500 random negative items, the model's resolution is only 1 out of 500. Thus, if we sample 500 random negative items out of 2 billion items, the chance of any of these items being even slightly related to the query item is small. Therefore, with large probability the learning will not make good parameter updates and will not be able to differentiate slightly related items from the very related ones.

To solve the above problem, for each positive training example (*i.e.*, item pair (q, i)), we add "hard" negative examples, *i.e.*, items that are somewhat related to the query item q, but not as related as the positive item i. We call these "hard negative items". They are generated by ranking items in a graph according to their Personalized PageRank scores with respect to query item q (Eksombatchai et al., 2018). Items ranked at 2000-5000 are randomly sampled as hard negative items. As illustrated in Figure 7.2, the hard negative examples are more similar to the query than random negative examples, and are thus challenging for the model to rank, forcing the model to learn to distinguish items at a finer granularity.

Using hard negative items throughout the training procedure doubles the number of epochs needed for the training to converge. To help with convergence, we develop a curriculum training scheme (Bengio et al., 2009). In the first epoch of training, no hard negative items are used, so that the algorithm quickly finds an area in the parameter space where the loss is relatively small. We

Figure 7.2: Random negative examples and hard negative examples. Notice that the hard negative example is significantly more similar to the query, than the random negative example, though not as similar as the positive example.

then add hard negative items in subsequent epochs, focusing the model to learn how to distinguish highly related pins from only slightly related ones. At epoch n of the training, we add $n-1$ hard negative items to the set of negative items for each item.

7.3.4 Node Embeddings via MapReduce

After the model is trained, it is still challenging to directly apply the trained model to generate embeddings for all items, including those that were not seen during training. Naively computing embeddings for nodes using Algorithm 3 leads to repeated computations caused by the overlap between K-hop neighborhoods of nodes. As illustrated in Figure 7.1, many nodes are repeatedly computed at multiple layers when generating the embeddings for different target nodes. To ensure efficient inference, we develop a MapReduce approach that runs model inference without repeated computations.

We observe that inference of node embeddings very nicely lends itself to MapReduce computational model. Figure 7.3 details the data flow on the bipartite pin-to-board Pinterest graph, where we assume the input (*i.e.*, "layer-0") nodes are pins/items (and the layer-1 nodes are boards/contexts). The MapReduce pipeline has two key parts:

1. One MapReduce job is used to project all pins to a low-dimensional latent space, where the aggregation operation will be performed (Algorithm 2, Line 1).
2. Another MapReduce job is then used to join the resulting pin representations with the ids of the boards they occur in, and the board embedding is computed by pooling the features of its (sampled) neighbors.

Note that our approach avoids redundant computations and that the latent vector for each node is

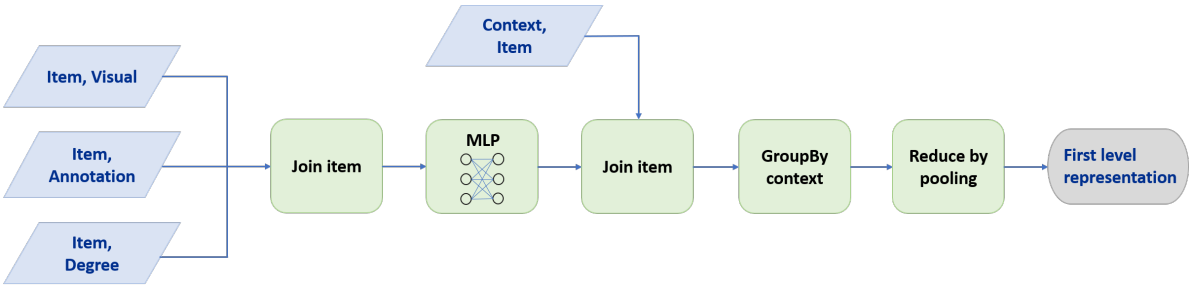


Figure 7.3: Node embedding data flow to compute the first layer representation using MapReduce. The second layer computation follows the same pipeline, except that the inputs are first layer representations, rather than raw item features.

computed only once. After the embeddings of the boards are obtained, we use two more MapReduce jobs to compute the second-layer embeddings of pins, in a similar fashion as above, and this process can be iterated as necessary (up to K convolutional layers).[3]

7.3.5 Efficient nearest-neighbor lookups

The embeddings generated by PinSage can be used for a wide range of downstream recommendation tasks, and in many settings we can directly use these embeddings to make recommendations by performing nearest-neighbor lookups in the learned embedding space. That is, given a query item q, the we can recommend items whose embeddings are the K-nearest neighbors of the query item's embedding. Approximate KNN can be obtained efficiently via locality sensitive hashing (Andoni and Indyk, 2006). After the hash function is computed, retrieval of items can be implemented with a two-level retrieval process based on the Weak AND operator (Broder et al., 2003). Given that the PinSage model is trained offline and all node embeddings are computed via MapReduce and saved in a database, the efficient nearest-neighbor lookup operation enables the system to serve recommendations in an online fashion,

7.4 Experiments

To demonstrate the efficiency of PinSage and the quality of the embeddings it generates, we conduct a comprehensive suite of experiments on the entire Pinterest object graph, including offline experiments, production A/B tests as well as user studies.

[3]Note that since we assume that only pins (and not boards) have features, we must use an even number of convolutional layers.

7.4.1 Experimental Setup

We evaluate the embeddings generated by PinSage in two tasks: recommending related pins and recommending pins in a user's home/news feed. To recommend related pins, we select the K nearest neighbors to the query pin in the embedding space. We evaluate performance on this related-pin recommendation task using both offline ranking measures as well as a controlled user study. For the homefeed recommendation task, we select the pins that are closest in the embedding space to one of the most recently pinned items by the user. We evaluate performance of a fully-deployed production system on this task using A/B tests to measure the overall impact on user engagement.

Training details and data preparation. We define the set, $\mathcal{L}$, of positive training examples (Equation (7.1)) using historical user engagement data. In particular, we use historical user engagement data to identify pairs of pins (q, i), where a user interacted with pin i immediately after she interacted with pin q. We use all other pins as negative items (and sample them as described in Section 7.3.3). Overall, we use 1.2 billion pairs of positive training examples (in addition to 500 negative examples per batch and 6 hard negative examples per pin). Thus in total we use 7.5 billion training examples.

Since PinSage can efficiently generate embeddings for unseen data, we only train on a subset of the Pinterest graph and then generate embeddings for the entire graph using the MapReduce pipeline described in Section 7.3.4. In particular, for training we use a randomly sampled subgraph of the entire graph, containing 20% of all boards (and all the pins touched by those boards) and 70% of the labeled examples. During hyperparameter tuning, a remaining 10% of the labeled examples are used. And, when testing, we run inference on the entire graph to compute embeddings for all 2 billion pins, and the remaining 20% of the labeled examples are used to test the recommendation performance of our PinSage in the offline evaluations. Note that training on a subset of the full graph drastically decreased training time, with a negligible impact on final performance. In total, the full datasets for training and evaluation are approximately 18TB in size with the full output embeddings being 4TB.

Features used for learning. Each pin at Pinterest is associated with an image and a set of textual annotations (title, description). To generate feature representation $\mathbf{x}_q$ for each pin q, we concatenate visual embeddings (4,096 dimensions), textual annotation embeddings (256 dimensions), and the log degree of the node/pin in the graph. The visual embeddings are the 6-th fully connected layer of a classification network using the VGG-16 architecture (Simonyan and Zisserman, 2014). Textual annotation embeddings are trained using a Word2Vec-based model (Mikolov et al., 2013), where the context of an annotation consists of other annotations that are associated with each pin.

Baselines for comparison. We evaluate the performance of PinSage against the following state-of-the-art content-based, graph-based and deep learning baselines that generate embeddings of pins:

1. Visual embeddings (**Visual**): Uses nearest neighbors of deep visual embeddings for recommendations. The visual features are described above.
2. Annotation embeddings (**Annotation**): Recommends based on nearest neighbors in terms of annotation embeddings. The annotation embeddings are described above.
3. Combined embeddings (**Combined**): Recommends based on concatenating visual and annotation embeddings, and using a 2-layer multi-layer perceptron to compute embeddings that capture both visual and annotation features.
4. Graph-based method (**Pixie**): This random-walk-based method (Eksombatchai et al., 2018) uses biased random walks to generate ranking scores by simulating random walks starting at query pin q. Items with top K scores are retrieved as recommendations. While this approach does not generate pin embeddings, it is currently the state-of-the-art at Pinterest for certain recommendation tasks (Eksombatchai et al., 2018) and thus an informative baseline.

The visual and annotation embeddings are state-of-the-art deep learning content-based systems currently deployed at Pinterest to generate representations of pins. Note that we do not compare against other deep learning baselines from the literature simply due to the scale of our problem. We also do not consider non-deep learning approaches for generating item/content embeddings, since other works have already proven state-of-the-art performance of deep learning approaches for generating such embeddings (Covington et al., 2016; Monti et al., 2017; den Oord et al., 2013).

We also conduct ablation studies and consider several variants of PinSage when evaluating performance:

- **max-pooling** uses the element-wise max as a symmetric aggregation function (i.e., $\gamma = \max$) without hard negative samples;
- **mean-pooling** uses the element-wise mean as a symmetric aggregation function (i.e., $\gamma = \text{mean}$);
- **mean-pooling-xent** is the same as mean-pooling but uses the cross-entropy loss introduced in (Hamilton et al., 2017c).
- **mean-pooling-hard** is the same as mean-pooling, except that it incorporates hard negative samples as detailed in Section 7.3.3.
- **PinSage** uses all optimizations presented in this paper, including the use of importance pooling in the convolution step.

The max-pooling and cross-entropy settings are extensions of the best-performing GCN model

from Hamilton et al. (Hamilton et al., 2017c)—other variants (*e.g.*, based on Kipf et al. (Kipf and Welling, 2016a)) performed significantly worse in development tests and are omitted for brevity.[4] For all the above variants, we used $K = 2$, hidden dimension size $m = 2048$, and set the embedding dimension d to be 1024.

Computation resources. Training of PinSage is implemented in TensorFlow (Abadi et al., 2016) and run on a single machine with 32 cores and 16 Tesla K80 GPUs. To ensure fast fetching of item's visual and annotation features, we store them in main memory, together with the graph, using Linux HugePages to increase the size of virtual memory pages from 4KB to 2MB. The total amount of memory used in training is 500GB. Our MapReduce inference pipeline is run on a Hadoop2 cluster with 378 d2.8xlarge Amazon AWS nodes.

7.4.2 Offline Evaluation

To evaluate performance on the related pin recommendation task, we define the notion of *hit-rate*. For each positive pair of pins (q, i) in the test set, we use q as a query pin and then compute its top K nearest neighbors NN_q from a sample of 5 million test pins. We then define the hit-rate as the fraction of queries q where i was ranked among the top K of the test sample (*i.e.*, where $i \in \mathrm{NN}_q$). This metric directly measures the probability that recommendations made by the algorithm contain the items related to the query pin q. In our experiments K is set to be 500.

We also evaluate the methods using Mean Reciprocal Rank (MRR), which takes into account of the rank of the item j among recommended items for query item q:

$$\mathrm{MRR} = \frac{1}{n} \sum_{(q,i) \in \mathcal{L}} \frac{1}{\lceil R_{i,q}/100 \rceil}. \tag{7.2}$$

Due to the large pool of candidates (more than 2 billion), we use a scaled version of the MRR in Equation (7.2), where $R_{i,q}$ is the rank of item i among recommended items for query q, and n is the total number of labeled item pairs. The scaling factor 100 ensures that, for example, the difference between rank at $1,000$ and rank at $2,000$ is still noticeable, instead of being very close to 0.

Table 7.1 compares the performance of the various approaches using the hit rate as well as the MRR.[5] PinSage with our new importance-pooling aggregation and hard negative examples

[4] Note that the recent GCN-based recommender systems of Monti et al. (Monti et al., 2017) and Berg et al. (van den Berg et al., 2017) are not directly comparable because they cannot scale to the Pinterest size data.

[5] Note that we do not include the Pixie baseline in these offline comparisons because the Pixie algorithm runs in production and is "generating" labeled pairs (q, j) for us—*i.e.*, the labeled pairs are obtained from historical user engagement data in which the Pixie algorithm was used as the recommender system. Therefore, the recommended item j is always in the recommendations made by the Pixie algorithm. However, we compare to the Pixie algorithm using human evaluations in Section 7.4.3.

Method	Hit-rate	MRR
Visual	17%	0.23
Annotation	14%	0.19
Combined	27%	0.37
max-pooling	39%	0.37
mean-pooling	41%	0.51
mean-pooling-xent	29%	0.35
mean-pooling-hard	46%	0.56
PinSage	**67%**	**0.59**

Table 7.1: Hit-rate and MRR for PinSage and content-based deep learning baselines. Overall, PinSage gives 150% improvement in hit rate and 60% improvement in MRR over the best baseline.

achieves the best performance at 67% hit-rate and 0.59 MRR, outperforming the top baseline by 40% absolute (150% relative) in terms of the hit rate and also 22% absolute (60% relative) in terms of MRR. We also observe that combining visual and textual information works much better than using either one alone (60% improvement of the combined approach over visual/annotation only).

Embedding similarity distribution. Another indication of the effectiveness of the learned embeddings is that the distances between random pairs of item embeddings are widely distributed. If all items are at about the same distance (*i.e.*, the distances are tightly clustered) then the embedding space does not have enough "resolution" to distinguish between items of different relevance. Figure 7.4 plots the distribution of cosine similarities between pairs of items using annotation, visual, and PinSage embeddings. This distribution of cosine similarity between random pairs of items demonstrates the effectiveness of PinSage, which has the most spread out distribution. In particular, the kurtosis of the cosine similarities of PinSage embeddings is 0.43, compared to 2.49 for annotation embeddings and 1.20 for visual embeddings.

Another important advantage of having such a wide-spread in the embeddings is that it reduces the collision probability of the subsequent LSH algorithm, thus increasing the efficiency of serving the nearest neighbor pins during recommendation.

7.4.3 User Studies

We also investigate the effectiveness of PinSage by performing head-to-head comparison between different learned representations. In the user study, a user is presented with an image of the query pin, together with two pins retrieved by two different recommendation algorithms. The user is then asked to choose which of the two candidate pins is more related to the query pin. Users are instructed to find various correlations between the recommended items and the query item, in aspects such as visual appearance, object category and personal identity. If both recommended

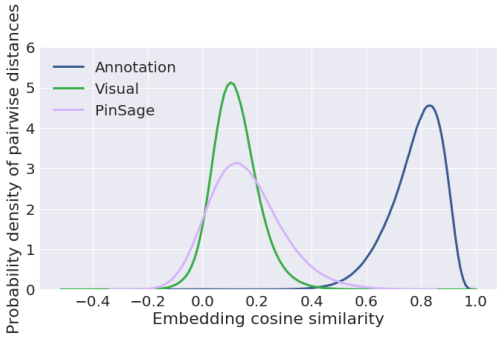


Figure 7.4: Probability density of pairwise cosine similarity for visual embeddings, annotation embeddings, and PinSage embeddings.

items seem equally related, users have the option to choose "equal". If no consensus is reached among $2/3$ of users who rate the same question, we deem the result as inconclusive.

Table 7.2 shows the results of the head-to-head comparison between PinSage and the 4 baselines. Among items for which the user has an opinion of which is more related, around 60% of the preferred items are recommended by PinSage. Figure 7.5 gives examples of recommendations and illustrates strengths and weaknesses of the different methods. The image to the left represents the query item. Each row to the right corresponds to the top recommendations made by the visual embedding baseline, annotation embedding baseline, Pixie, and PinSage. Although visual embeddings generally predict categories and visual similarity well, they occasionally make large mistakes in terms of image semantics. In this example, visual information confused plants with food, and tree logging with war photos, due to similar image style and appearance. The graph-based Pixie method, which uses the graph of pin-to-board relations, correctly understands that the category of query is "plants" and it recommends items in that general category. However, it does not find the most relevant items. Combining both visual/textual and graph information, PinSage is able to find relevant items that are both visually and topically similar to the query item.

In addition, we visualize the embedding space by randomly choosing 1000 items and compute the 2D t-SNE coordinates from the PinSage embedding, as shown in Figure 7.6.[6] We observe that the proximity of the item embeddings corresponds well with the similarity of content, and that items of the same category are embedded into the same part of the space. Note that items that are visually different but have the same theme are also close to each other in the embedding space, as seen by the items depicting different fashion-related items on the bottom side of the plot.

[6]Some items are overlapped and are not visible.

Methods	Win	Lose	Draw	Fraction of wins
PinSage vs. Visual	28.4%	21.9%	49.7%	56.5%
PinSage vs. Annot.	36.9%	14.0%	49.1%	72.5%
PinSage vs. Combined	22.6%	15.1%	57.5%	60.0%
PinSage vs. Pixie	32.5%	19.6%	46.4%	62.4%

Table 7.2: Head-to-head comparison of which image is more relevant to the recommended query image.

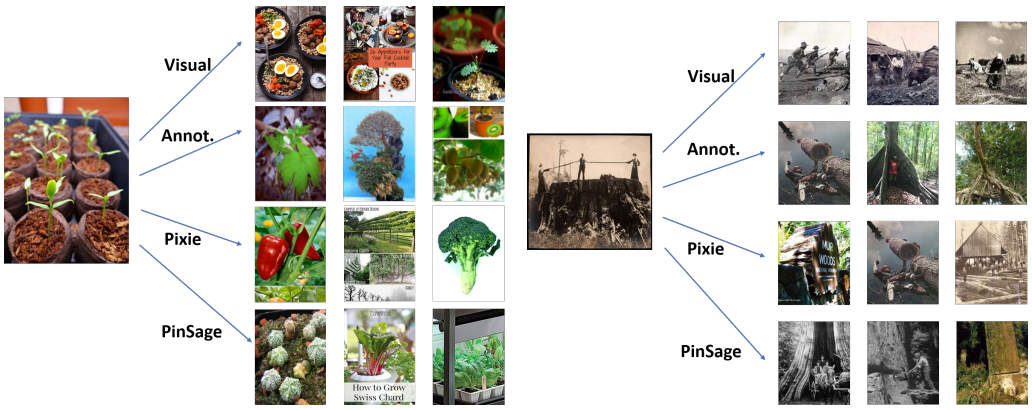


Figure 7.5: Examples of Pinterest pins recommended by different algorithms. The image to the left is the query pin. Recommended items to the right are computed using Visual embeddings, Annotation embeddings, graph-based Pixie, and PinSage.

7.4.4 Production A/B Test

Lastly, we also report on the production A/B test experiments, which compared the performance of PinSage to other deep learning content-based recommender systems at Pinterest on the task of homefeed recommendations. We evaluate the performance by observing the lift in user engagement. The metric of interest is *repin rate*, which measures the percentage of homefeed recommendations that have been saved by the users. A user saving a pin to a board is a high-value action that signifies deep engagement of the user. It means that a given pin presented to a user at a given time was relevant enough for the user to save that pin to one of their boards so that they can retrieve it later.

We find that PinSage consistently recommends pins that are more likely to be re-pinned by the user than the alternative methods. Depending on the particular setting, we observe 10-30% improvements in repin rate over the Annotation and Visual embedding based recommendations.

7.4.5 Training and Inference Runtime Analysis

One advantage of GraphSAGE framework is that the GNNs can be made inductive (Hamilton et al., 2017b): at the inference (*i.e.*, embedding generation) step, we are able to compute embeddings for

Figure 7.6: t-SNE plot of item embeddings in 2 dimensions.

Batch size	Per iteration (ms)	# iterations	Total time (h)
512	590	390k	63.9
1024	870	220k	53.2
2048	1350	130k	48.8
4096	2240	100k	68.4

Table 7.3: Runtime comparisons for different batch sizes.

items that were not in the training set. This allows us to train on a subgraph to obtain model parameters, and then make embed nodes that have not been observed during training. Also note that it is easy to compute embeddings of new nodes that get added into the graph over time. This means that recommendations can be made on the full (and constantly growing) graph. Experiments on development data demonstrated that training on a subgraph containing 300 million items could achieve the best performance in terms of hit-rate (*i.e.*, further increases in the training set size did not seem to help), reducing the runtime by a factor of 6 compared to training on the full graph.

Table 7.3 shows the the effect of batch size of the minibatch SGD on the runtime of PinSage training procedure, using the mean-pooling-hard variant. For varying batch sizes, the table shows: (1) the computation time, in milliseconds, for each minibatch, when varying batch size; (2) the number of iterations needed for the model to converge; and (3) the total estimated time for the training procedure. Experiments show that a batch size of 2048 makes training most efficient.

When training the PinSage variant with importance pooling, another trade-off comes from choosing the size of neighborhood T. Table 7.3 shows the runtime and performance of PinSage

# neighbors	Hit-rate	MRR	Training time (h)
10	60%	0.51	20
20	63%	0.54	33
50	67%	0.59	78

Table 7.4: Performance tradeoffs for importance pooling.

when $T = 10$, 20 and 50. We observe a diminishing return as T increases, and find that a two-layer GCN with neighborhood size 50 can best capture the neighborhood information of nodes, while still being computationally efficient.

After training completes, due to the highly efficient MapReduce inference pipeline, the whole inference procedure to generate embeddings for 3 billion items can finish in less than 24 hours.

Summary. In this chapter, we introduced PinSage, a random-walk GNN. PinSage is a highly-scalable GCN algorithm capable of learning embeddings for nodes in web-scale graphs containing billions of objects. In addition to new techniques that ensure scalability, we introduced the use of importance pooling and curriculum training that drastically improved embedding performance. We deployed PinSage at Pinterest and comprehensively evaluated the quality of the learned embeddings on a number of recommendation tasks, with offline metrics, user studies and A/B tests all demonstrating a substantial improvement in recommendation performance. Our work demonstrates the impact that graph convolutional methods can have in a production recommender system, and we believe that PinSage can be further extended in the future to tackle other graph representation learning problems at large scale, including knowledge graph reasoning and graph clustering.

Chapter 8

Learning to Simulate Complex Physics

In Chapter 8, we investigate another application of GNNs in very different domains, *i.e.* physics and graphics simulation. In collaboration with DeepMind Inc., I proposed a GNN model, GNS, which can predict the future evolution of different materials (*e.g.* water, sand, goop) over a long time range. The key idea is to model the materials in terms of particles [1] which interact with each other through forces. GNS uses graph neural network to predict the interactions between the particles, and thus can iteratively predict particles' future movement given the current particle states.

8.1 Introduction of Learning Simulations

Realistic simulators of complex physics are invaluable to many scientific and engineering disciplines, however traditional simulators can be very expensive to create and use. Building a simulator can entail years of engineering effort, and often must trade off generality for accuracy in a narrow range of settings. High-quality simulators require substantial computational resources, which makes scaling up prohibitive. Even the best are often inaccurate due to insufficient knowledge of, or difficulty in approximating, the underlying physics and parameters. An attractive alternative to traditional simulators is to use machine learning to train simulators directly from observed data, however the large state spaces and complex dynamics have been difficult for standard end-to-end learning approaches to overcome.

Here we present a powerful machine learning framework for learning to simulate complex systems from data—"Graph Network-based Simulators" (GNS). Our framework imposes strong

[1] In graphics, these particles are elements that are used to model the material, while in physics, the particles could correspond to molecule, atom or even smaller particles.

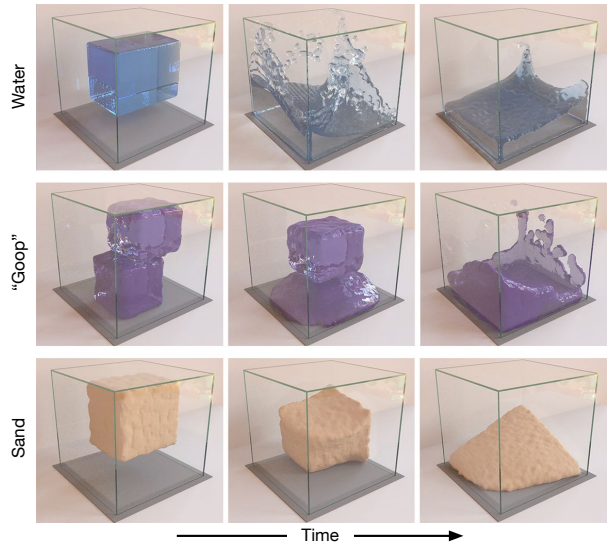


Figure 8.1: Rollouts of our GNS model for our WATER-3D, GOOP-3D and SAND-3D datasets. It learns to simulate rich materials at resolutions sufficient for high-quality rendering [video].

inductive biases, where rich physical states are represented by graphs of interacting particles, and complex dynamics are approximated by learned message-passing among nodes.

We implemented our GNS framework in a single deep learning architecture, and found it could learn to accurately simulate a wide range of physical systems in which fluids, rigid solids, and deformable materials interact with one another. Our model also generalized well to much larger systems and longer time scales than those on which it was trained. While previous learning simulation approaches (Li et al., 2018b; Ummenhofer et al., 2020) have been highly specialized for particular tasks, we found our single GNS model performed well across dozens of experiments and was generally robust to hyperparameter choices. Our analyses showed that performance was determined by a handful of key factors: its ability to compute long-range interactions, inductive biases for spatial invariance, and training procedures which mitigate the accumulation of error over long simulated trajectories.

8.2 Related Work

Our approach focuses on *particle-based* simulation, which is used widely across science and engineering, e.g., computational fluid dynamics, computer graphics. States are represented as a set of particles, which encode mass, material, movement, etc. within local regions of space. Dynamics are computed on the basis of particles' interactions within their local neighborhoods. One popular

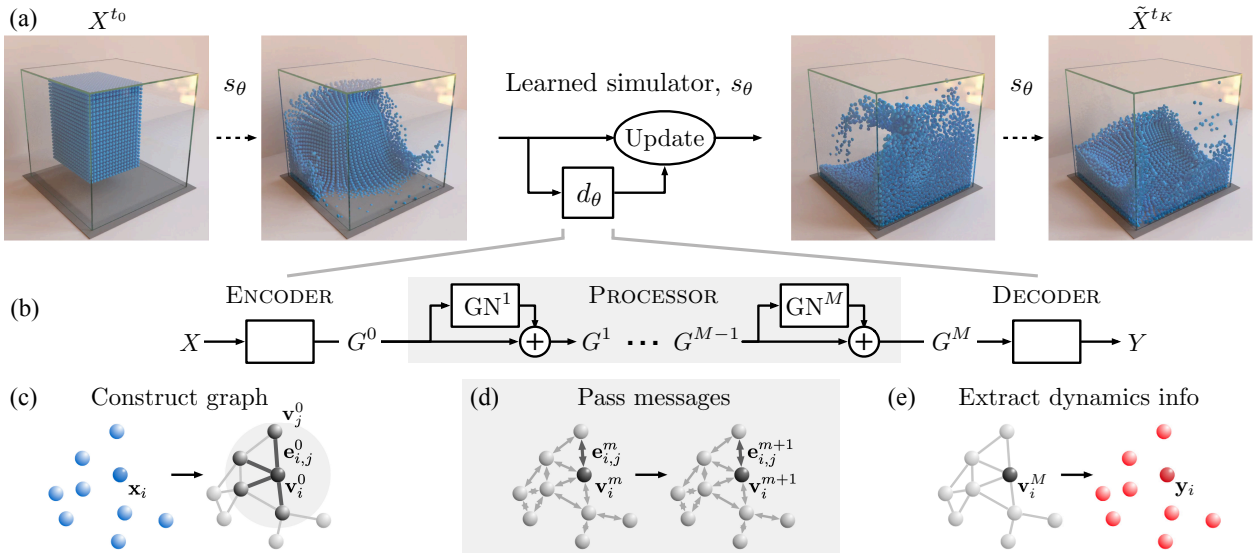


Figure 8.2: **(a)** Our GNS predicts future states represented as particles using its learned dynamics model, d_θ, and a fixed update procedure. **(b)** The d_θ uses an "encode-process-decode" scheme, which computes dynamics information, Y, from input state, X. **(c)** The ENCODER constructs latent graph, G^0, from the input state, X. **(d)** The PROCESSOR performs M rounds of learned message-passing over the latent graphs, $G^0, \ldots, G^M$. **(e)** The DECODER extracts dynamics information, Y, from the final latent graph, G^M.

particle-based method for simulating fluids is "smoothed particle hydrodynamics" (SPH) (Monaghan, 1992), which evaluates pressure and viscosity forces around each particle, and updates particles' velocities and positions accordingly. Other techniques, such as "position-based dynamics" (PBD) (Müller et al., 2007) and "material point method" (MPM) (Sulsky et al., 1995), are more suitable for interacting, deformable materials. In PBD, incompressibility and collision dynamics involve resolving pairwise distance constraints between particles, and directly predicting their position changes. Several differentiable particle-based simulators have recently appeared, e.g., DiffTaichi (Hu et al., 2019), PhiFlow (Holl et al., 2020), and Jax-MD (Schoenholz and Cubuk, 2019), which can backpropagate gradients through the architecture.

Learning simulations from data (Grzeszczuk et al., 1998) has been an important area of study with applications in physics and graphics. Compared to engineered simulators, a learned simulator can be far more efficient for predicting complex phenomena (He et al., 2019); e.g., (Ladickỳ et al., 2015; Wiewel et al., 2019) learn parts of a fluid simulator for faster prediction.

Graph Networks (GN) (Hamrick et al., 2018)—a type of graph neural network (Scarselli et al., 2008)—have recently proven effective at learning forward dynamics in various settings that involve interactions between many entities. A GN maps an input graph to an output graph with the same structure but potentially different node, edge, and graph-level attributes, and can be trained to learn

a form of message-passing (Gilmer et al., 2017), where latent information is propagated between nodes via the edges. GNs and their variants, e.g., "interaction networks", can learn to simulate rigid body, mass-spring, n-body, and robotic control systems (Battaglia et al., 2016; Chang et al., 2016; Sanchez-Gonzalez et al., 2018; Mrowca et al., 2018; Li et al., 2019; Sanchez-Gonzalez et al., 2019), as well as non-physical systems, such as multi-agent dynamics (Tacchetti et al., 2018; Sun et al., 2019a), algorithm execution (Veličković et al., 2020), and other dynamic graph settings (Trivedi et al., 2019, 2017; Yan et al., 2018; Manessi et al., 2020).

Our GNS framework builds on and generalizes several lines of work, especially Sanchez-Gonzalez et al. (2018)'s GN-based model which was applied to various robotic control systems, Li et al. (2018b)'s DPI which was applied to fluid dynamics, and Ummenhofer et al. (2020)'s Continuous Convolution (CConv) which was presented as a non-graph-based method for simulating fluids. Crucially, our GNS framework is a *general* approach to learning simulation, is simpler to implement, and is more accurate across fluid, rigid, and deformable material systems.

8.3 GNS Model Framework

8.3.1 General Learnable Simulation

We assume $X^t \in \mathcal{X}$ is the state of the world at time t. Applying physical dynamics over K timesteps yields a trajectory of states, $\mathbf{X}^{t_{0:K}} = (X^{t_0}, \ldots, X^{t_K})$. A *simulator*, $s : \mathcal{X} \to \mathcal{X}$, models the dynamics by mapping preceding states to causally consequent future states. We denote a simulated "rollout" trajectory as, $\tilde{\mathbf{X}}^{t_{0:K}} = (X^{t_0}, \tilde{X}^{t_1}, \ldots, \tilde{X}^{t_K})$, which is computed iteratively by, $\tilde{X}^{t_{k+1}} = s(\tilde{X}^{t_k})$ for each timestep. Simulators compute dynamics information that reflects how the current state is changing, and use it to update the current state to a predicted future state (see Figure 8.2(a)). An example is a numerical differential equation solver: the equations compute dynamics information, i.e., time derivatives, and the integrator is the update mechanism.

A learnable simulator, s_θ, computes the dynamics information with a parameterized function approximator, $d_\theta : \mathcal{X} \to \mathcal{Y}$, whose parameters, θ, can be optimized for some training objective. The $Y \in \mathcal{Y}$ represents the dynamics information, whose semantics are determined by the update mechanism. The update mechanism can be seen as a function which takes the $\tilde{X}^{t_k}$, and uses d_θ to predict the next state, $\tilde{X}^{t_{k+1}} = \text{Update}(\tilde{X}^{t_k}, d_\theta)$. Here we assume a simple update mechanism—an Euler integrator—and $\mathcal{Y}$ that represents accelerations. However, more sophisticated update procedures which call d_θ more than once can also be used, such as higher-order integrators (e.g., Sanchez-Gonzalez et al. (2019)).

8.3.2 Simulation as Message-Passing on a Graph

Our learnable simulation approach adopts a particle-based representation of the physical system (see Section 9.2), i.e., $X = (\mathbf{x}_0, \ldots, \mathbf{x}_N)$, where each of the N particles' $\mathbf{x}_i$ represents its state. Physical dynamics are approximated by interactions among the particles, e.g., exchanging energy and momentum among their neighbors. The way particle-particle interactions are modeled determines the quality and generality of a simulation method—i.e., the types of effects and materials it can simulate, in which scenarios the method performs well or poorly, etc. We are interested in learning these interactions, which should, in principle, allow learning the dynamics of any system that can be expressed as particle dynamics. So it is crucial that different θ values allow d_θ to span a wide range of particle-particle interaction functions.

Particle-based simulation can be viewed as message-passing on a graph. The nodes correspond to particles, and the edges correspond to pairwise relations among particles, over which interactions are computed. We can understand methods like SPH in this framework—the messages passed between nodes could correspond to, e.g., evaluating pressure using the density kernel.

We capitalize on the correspondence between particle-based simulators and message-passing on graphs to define a general-purpose d_θ based on GNs. Our d_θ has three steps—ENCODER, PROCESSOR, DECODER (Battaglia et al., 2018) (see Figure 8.2(b)).

ENCODER definition. The ENCODER : $\mathcal{X} \to \mathcal{G}$ embeds the particle-based state representation, X, as a latent graph, $G^0 = \text{ENCODER}(X)$, where $G = (V, E, \mathbf{u})$, $\mathbf{v}_i \in V$, and $\mathbf{e}_{i,j} \in E$ (see Figure 8.2(b,c)). The node embeddings, $\mathbf{v}_i = \varepsilon^v(\mathbf{x}_i)$, are learned functions of the particles' states. Directed edges are added to create paths between particle nodes which have some potential interaction. The edge embeddings, $\mathbf{e}_{i,j} = \varepsilon^e(\mathbf{r}_{i,j})$, are learned functions of the pairwise properties of the corresponding particles, $\mathbf{r}_{i,j}$, e.g., displacement between their positions, spring constant, etc. The graph-level embedding, $\mathbf{u}$, can represent global properties such as gravity and magnetic fields (though in our implementation we simply appended those as input node features—see Section 8.4.2 below).

PROCESSOR definition. The PROCESSOR : $\mathcal{G} \to \mathcal{G}$ computes interactions among nodes via M steps of learned message-passing, to generate a sequence of updated latent graphs, $\mathbf{G} = (G^1, ..., G^M)$, where $G^{m+1} = \text{GN}^{m+1}(G^m)$ (see Figure 8.2(b,d)). It returns the final graph, $G^M = \text{PROCESSOR}(G^0)$. Message-passing allows information to propagate and constraints to be respected: the number of message-passing steps required will likely scale with the complexity of the interactions.

DECODER definition. The DECODER : $\mathcal{G} \to \mathcal{Y}$ extracts dynamics information from the nodes of the final latent graph, $\mathbf{y}_i = \delta^v(\mathbf{v}_i^M)$ (see Figure 8.2(b,e)). Learning δ^v should cause the $\mathcal{Y}$ representations to reflect relevant dynamics information, such as acceleration, in order to be semantically

meaningful to the update procedure.

8.4 Experimental Methods

Code and data available at:
github.com/deepmind/deepmind-research/tree/master/learning_to_simulate.

8.4.1 Physical Domains

We explored how our GNS learns to simulate in datasets which contained three diverse, complex physical materials: water as a barely damped fluid, chaotic in nature; sand as a granular material with complex frictional behavior; and "goop" as a viscous, plastically deformable material. These materials have very different behavior, and in most simulators, require implementing separate material models or even entirely different simulation algorithms.

For one domain, we use Li et al. (2018b)'s BOXBATH, which simulates a container of water and a cube floating inside, all represented as particles, using the PBD engine FleX (Macklin et al., 2014).

We also created WATER-3D, a high-resolution 3D water scenario with randomized water position, initial velocity and volume, comparable to Ummenhofer et al. (2020)'s containers of water. We used SPlisHSPlasH (Bender and Koschier, 2015), a SPH-based fluid simulator with strict volume preservation to generate this dataset.

For most of our domains, we use the Taichi-MPM engine (Hu et al., 2018) to simulate a variety of challenging 2D and 3D scenarios. We chose MPM for the simulator because it can simulate a very wide range of materials, and also has some different properties than PBD and SPH, e.g., particles may become compressed over time.

Our datasets typically contained 1000 train, 100 validation and 100 test trajectories, each simulated for 300-2000 timesteps (tailored to the average duration for the various materials to come to a stable equilibrium).

8.4.2 GNS Implementation Details

We implemented the components of the GNS framework using standard deep learning building blocks, and used standard nearest neighbor algorithms (Dong et al., 2011; Chen et al., 2009; Tang et al., 2016) to construct the graph.

Input and output representations. Each particle's input state vector represents position, a sequence of $C = 5$ previous velocities[2], and features that capture static material properties (e.g., water, sand, goop, rigid, boundary particle), $\mathbf{x}_i^{t_k} = [\mathbf{p}_i^{t_k}, \dot{\mathbf{p}}_i^{t_{k-C+1}}, \ldots, \dot{\mathbf{p}}_i^{t_k}, \mathbf{f}_i]$, respectively. The global properties of the system, $\mathbf{g}$, include external forces and global material properties, when applicable. The prediction targets for supervised learning are the per-particle average acceleration, $\ddot{\mathbf{p}}_i$. Note that in our datasets, we only require $\mathbf{p}_i$ vectors: the $\dot{\mathbf{p}}_i$ and $\ddot{\mathbf{p}}_i$ are computed from $\mathbf{p}_i$ using finite differences.

ENCODER details. The ENCODER constructs the graph structure G^0 by assigning a node to each particle and adding edges between particles within a "connectivity radius", R, which reflected local interactions of particles, and which was kept constant for all simulations of the same resolution. For generating rollouts, on each timestep the graph's edges were recomputed by a nearest neighbor algorithm, to reflect the current particle positions.

The ENCODER implements ε^v and ε^e as multilayer perceptrons (MLP), which encode node features and edge features into the latent vectors, $\mathbf{v}_i$ and $\mathbf{e}_{i,j}$, of size 128.

We tested two ENCODER variants, distinguished by whether they use absolute versus relative positional information. For the absolute variant, the input to ε^v was the $\mathbf{x}_i$ described above, with the globals features concatenated to it. The input to ε^e, i.e., $\mathbf{r}_{i,j}$, did not actually carry any information and was discarded, with the $\mathbf{e}_i^0$ in G^0 set to a trainable fixed bias vector. The relative ENCODER variant was designed to impose an inductive bias of invariance to absolute spatial location. The ε^v was forced to ignore $\mathbf{p}_i$ information within $\mathbf{x}_i$ by masking it out. The ε^e was provided with the relative positional displacement, and its magnitude[3], $\mathbf{r}_{i,j} = [(\mathbf{p}_i - \mathbf{p}_j), \|\mathbf{p}_i - \mathbf{p}_j\|]$. Both variants concatenated the global properties $\mathbf{g}$ onto each $\mathbf{x}_i$ before passing it to ε^v.

PROCESSOR details. Our processor uses a stack of M GNs (where M is a hyperparameter) with identical structure, MLPs as internal edge and node update functions, and either shared or unshared parameters (as analyzed in Results Section 8.5.2). We use GNs without global features or global updates (similar to an interaction network)[4], and with a residual connections between the input and output latent node and edge attributes.

DECODER details. Our decoder's learned function, δ^v, is an MLP. After the DECODER, the future position and velocity are updated using an Euler integrator, so the $\mathbf{y}_i$ corresponds to accelerations, $\ddot{\mathbf{p}}_i$, with 2D or 3D dimension, depending on the physical domain. As mentioned above, the supervised training outputs were simply these, $\ddot{\mathbf{p}}_i$ vectors[5].

[2] C is a hyperparameter which we explore in our experiments.

[3] Similarly, relative velocities could be used to enforce invariance to inertial frames of reference.

[4] In preliminary experiments we also attempted using a PROCESSOR with a full GN and a global latent state, for which the global features $\mathbf{g}$ are encoded with a separate ε^g MLP.

[5] Note that in this case optimizing for acceleration is equivalent to optimizing for position, because the acceleration

Neural network parameterizations. All MLPs have two hidden layers (with ReLU activations), followed by a non-activated output layer, each layer with size of 128. All MLPs (except the output decoder) are followed by a LayerNorm (Ba et al., 2016) layer, which we generally found improved training stability.

8.4.3 Training

Software. We implemented our models using TensorFlow 1, Sonnet 1, and the "Graph Nets" library (2018).

Training noise. Modeling a complex and chaotic simulation system requires the model to mitigate error accumulation over long rollouts. Because we train our models on ground-truth one-step data, they are never presented with input data corrupted by this sort of accumulated noise. This means that when we generate a rollout by feeding the model with its own noisy, previous predictions as input, the fact that its inputs are outside the training distribution may lead it to make more substantial errors, and thus rapidly accumulate further error. We use a simple approach to make the model more robust to noisy inputs: at training we corrupt the input velocities of the model with random-walk noise $\mathcal{N}(0, \sigma_v = 0.0003)$ (adjusting input positions), so the training distribution is closer to the distribution generated during rollouts.

Normalization. We normalize all input and target vectors elementwise to zero mean and unit variance, using statistics computed online during training. Preliminary experiments showed that normalization led to faster training, though converged performance was not noticeably improved.

Loss function and optimization procedures. We randomly sampled particle state pairs $(\mathbf{x}_i^{t_k}, \mathbf{x}_i^{t_{k+1}})$ from training trajectories, calculated target accelerations $\ddot{\mathbf{p}}_i^{t_k}$ (subtracting the noise added to the most recent input velocity), and computed the L_2 loss on the predicted per-particle accelerations, i.e., $L(\mathbf{x}_i^{t_k}, \mathbf{x}_i^{t_{k+1}}; \theta) = \|d_\theta(\mathbf{x}_i^{t_k}) - \ddot{\mathbf{p}}_i^{t_k}\|^2$. We optimized the model parameters θ over this loss with the Adam optimizer (Kingma and Ba, 2015), using a nominal[6] mini-batch size of 2. We performed a maximum of 20M gradient update steps, with exponential learning rate decay from 10^{-4} to 10^{-6}. While models can train in significantly less steps, we avoid aggressive learning rates to reduce variance across datasets and make comparisons across settings more fair.

We evaluated our models regularly during training by producing full-length rollouts on 5 held-out validation trajectories, and recorded the associated model parameters for best rollout MSE. We stopped training when we observed negligible decrease in MSE, which, on GPU/TPU hardware, was typically within a few hours for smaller, simpler datasets, and up to a week for the larger, more

is computed as first order finite difference from the position and we use an Euler integrator to update the position.

[6]The actual batch size varies at each step dynamically. See Supplementary Material for more details.

Experimental domain	N	K	**1-step** ($\times 10^{-9}$)	**Rollout** ($\times 10^{-3}$)
WATER-3D (SPH)	13k	800	8.66	10.1
SAND-3D	20k	350	1.42	0.554
GOOP-3D	14k	300	1.32	0.618
WATER-3D-S (SPH)	5.8k	800	9.66	9.52
BOXBATH (PBD)	1k	150	54.5	4.2
WATER	1.9k	1000	2.82	17.4
SAND	2k	320	6.23	2.37
GOOP	1.9k	400	2.91	1.89
MULTIMATERIAL	2k	1000	1.81	16.9
FLUIDSHAKE	1.3k	2000	2.1	20.1
WATERDROP	1k	1000	1.52	7.01
WATERDROP-XL	7.1k	1000	1.23	14.9
WATERRAMPS	2.3k	600	4.91	11.6
SANDRAMPS	3.3k	400	2.77	2.07
RANDOMFLOOR	3.4k	600	2.77	6.72
CONTINUOUS	4.3k	400	2.06	1.06

Table 8.1: List of maximum number of particles N, sequence length K, and quantitative model accuracy (MSE) on the held-out test set. All domain names are also hyperlinks to the video website.

complex datasets.

8.4.4 Evaluation

To report quantitative results, we evaluated our models after training converged by computing one-step and rollout metrics on held-out test trajectories, drawn from the same distribution of initial conditions used for training. We used particle-wise MSE as our main metric between ground truth and predicted data, both for rollout and one-step predictions, averaging across time, particle and spatial axes. We also investigated distributional metrics including optimal transport (OT) (Villani, 2003) (approximated by the Sinkhorn Algorithm (Cuturi, 2013)), and Maximum Mean Discrepancy (MMD) (Gretton et al., 2012). For the generalization experiments we also evaluate our models on a number of initial conditions drawn from distributions different than those seen during training, including, different number of particles, different object shapes, different number of objects, different initial positions and velocities and longer trajectories. See the original paper for full details on metrics and evaluation.

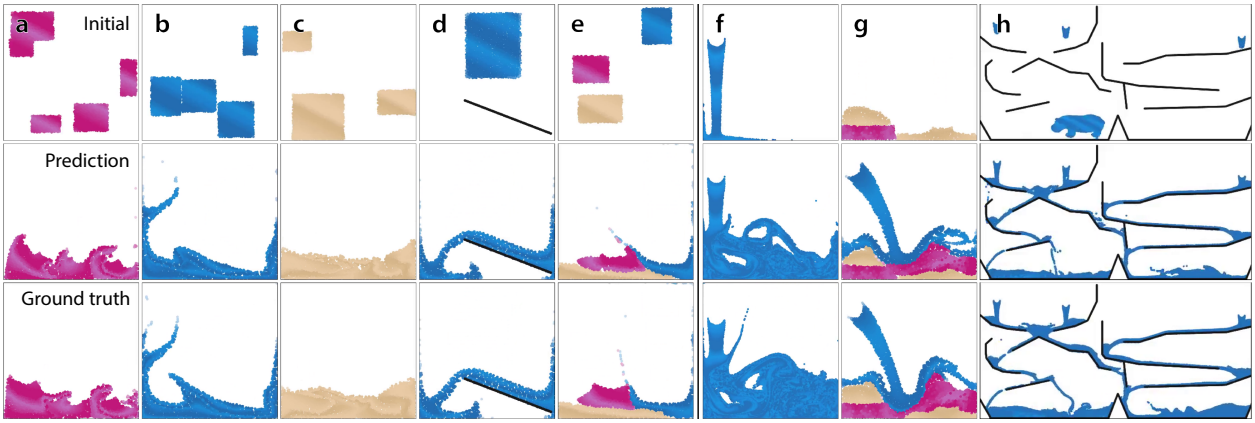


Figure 8.3: We can simulate many materials, from **(a)** GOOP over **(b)** WATER to **(c)** SAND, and **(d)** their interaction with rigid obstacles (WATERRAMPS). We can even train a single model on **(e)** multiple materials and their interaction (MULTIMATERIAL). We applied pre-trained models on several out-of-distribution tasks, involving **(f)** high-res turbulence (trained on WATERRAMPS), **(g)** multi-material interactions with unseen objects (trained on MULTIMATERIAL), and **(h)** generalizing on significantly larger domains (trained on WATERRAMPS). In the two bottom rows, we show a comparison of our model's prediction with the ground truth on the final frame for goop and sand, and on a representative mid-trajectory frame for water.

8.5 Simulation Results Analysis

Our main findings are that our GNS model can learn accurate, high-resolution, long-term simulations of different fluids, deformables, and rigid solids, and it can generalize well beyond training to much longer, larger, and challenging settings. In Section 8.5.3 below, we compare our GNS model to two recent, related approaches, and find our approach was simpler, more generally applicable, and more accurate.

To challenge the robustness of our architecture, we used a single set of model hyperparameters for training across all of our experiments. Our GNS architecture used the relative ENCODER variant, 10 steps of message-passing, with unshared GN parameters in the PROCESSOR. We applied noise with a scale of $3 \cdot 10^{-4}$ to the input states during training.

8.5.1 Simulating Complex Materials

Our GNS model was very effective at learning to simulate different complex materials. Table 8.1 shows the one-step and rollout accuracy, as MSE, for all experiments. For intuition about what these numbers mean, the edge length of the container was approximately 1.0, and Figure 8.3(a-c) shows rendered images of the rollouts of our model, compared to ground truth[7]. Visually, the

[7]All rollout videos can be found here: https://sites.google.com/view/learning-to-simulate

model's rollouts are quite plausible. Though specific model-generated trajectories can be distinguished from ground truth when compared side-by-side, it is difficult to visually classify individual videos as generated from our model versus the ground truth simulator.

Our GNS model scales to large amounts of particles and very long rollouts. With up to 19k particles in our 3D domains—substantially greater than demonstrated in previous methods—GNS can operate at resolutions high enough for practical prediction tasks and high-quality 3D renderings (e.g., Figure 8.1). And although our models were trained to make one-step predictions, the long-term trajectories remain plausible even over thousands of rollout timesteps.

The GNS model could also learn how the materials respond to unpredictable external forces. In the FLUIDSHAKE domain, a container filled with water is being moved side-to-side, causing splashes and irregular waves.

Our model could also simulate fluid interacting with complicated static obstacles, as demonstrated by our WATERRAMPS and SANDRAMPS domains in which water or sand pour over 1-5 obstacles. Figure 8.3(d) depicts comparisons between our model and ground truth, and Table 8.1 shows quantitative performance measures.

We also trained our model on continuously varying material parameters. In the CONTINUOUS domain, we varied the friction angle of a granular material, to yield behavior similar to a liquid ($0°$), sand ($45°$), or gravel ($> 60°$). Our results and videos show that our model can account for these continuous variations, and even interpolate between them: a model trained with the region $[30°, 55°]$ held out in training can accurately predict within that range.

Multiple Interacting Materials. So far we have reported results of training identical GNS architectures separately on different systems and materials. However, we found we could go a step further and train a single architecture with a single set of parameters to simulate all of our different materials, interacting with each other in a single system.

In our MULTIMATERIAL domain, the different materials could interact with each other in complex ways, which means the model had to effectively learn the product space of different interactions (e.g., water-water, sand-sand, water-sand, etc.). The behavior of these systems was often much richer than the single-material domains: the stiffer materials, such as sand and goop, could form temporary semi-rigid obstacles, which the water would then flow around. Figure 8.3(e) and this video shows renderings of such rollouts. Visually, our model's performance in MULTIMATERIAL is comparable to its performance when trained on those materials individually.

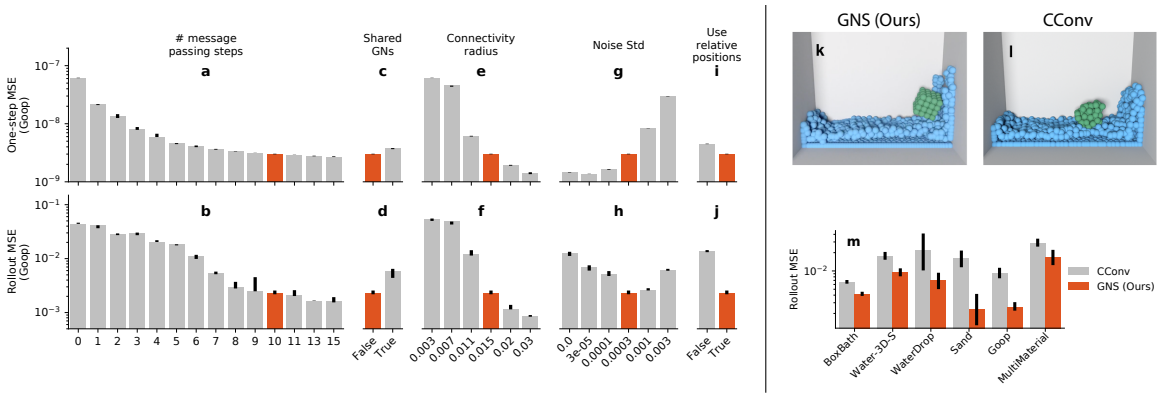


Figure 8.4: (left) Effect of different ablations (grey) against our model (red) on the one-step error **(a,c,e,g,i)** and the rollout error **(b,d,f,h,j)**. Bars show the median seed performance averaged across the entire GOOP test dataset. Error bars display lower and higher quartiles, and are shown for the default parameters. (right) Comparison of average performance of our GNS model to CConv. **(k,l)** Qualitative comparison between GNS **(k)** and CConv **(l)** in BOXBATH after 50 rollout steps (video link). **(m)** Quantitative comparison of our GNS model (red) to the CConv model (grey) across the test set . For our model, we trained one or more seeds using the same set of hyper-parameters and show results for all seeds. For the CConv model we ran several variations including different radius sizes, noise levels, and number of unroll steps during training, and show the result for the best seed. Errors bars show the standard error of the mean across all of the trajectories in the test set (95% confidence level).

8.5.2 Generalization

We found that the GNS generalizes well even beyond its training distributions, which suggests it learns a more general-purpose understanding of the materials and physical processes experienced during training.

To examine its capacity for generalization, we trained a GNS architecture on WATERRAMPS, whose initial conditions involved a square region of water in a container, with 1-5 ramps of random orientation and location. After training, we tested the model on several very different settings. In one generalization condition, rather than all water being present in the initial timestep, we created an "inflow" that continuously added water particles to the scene during the rollout, as shown in Figure 8.3(f). When unrolled for 2500 time steps, the scene contained 28k particles—an order of magnitude more than the 2.5k particles used in training—and the model was able to predict complex, highly chaotic dynamics not experienced during training, as can be seen in this video. The predicted dynamics were visually similar to the ground truth sequence.

Because we used relative displacements between particles as input to our model, in principle the model should handle scenes with much larger spatial extent at test time. We evaluated this on a much larger domain, with several inflows over a complicated arrangement of slides and ramps

(see Figure 8.3(h), video here). The test domain's spatial width $\times$ height were 8.0×4.0, which was 32x larger than the training domain's area; at the end of the rollout, the number of particles was 85k, which was 34x more than during training; we unrolled the model for 5000 steps, which was 8x longer than the training trajectories. We conducted a similar experiment with sand on the SANDRAMPS domain, testing model generalization to hourglass-shaped ramps.

As a final, extreme test of generalization, we applied a model trained on MULTIMATERIAL to a custom test domain with inflows of various materials and shapes (Figure 8.3(g)). The model learned about frictional behavior between different materials (sand on sticky goop, versus slippery floor), and that the model generalized well to unseen shapes, such as hippo-shaped chunks of goop and water, falling from mid-air, as can be observed in this video.

Key Architectural Choices.

We performed a comprehensive analysis of our GNS's architectural choices to discover what influenced performance most heavily. We analyzed a number of hyperparameter choices—e.g., number of MLP layers, linear encoder and decoder functions, global latent state in the PROCESSOR—but found these had minimal impact on performance.

While our GNS model was generally robust to architectural and hyperparameter settings, we also identified several factors which had more substantial impact:

1. the number of message-passing steps,
2. shared vs. unshared PROCESSOR GN parameters,
3. the connectivity radius,
4. the scale of noise added to the inputs during training,
5. relative vs. absolute ENCODER.

We varied these choices systematically for each axis, fixing all other axes with the default architecture's choices, and report their impact on model performance in the GOOP domain (Figure 8.4).

For (1), Figure 8.4(a,b) shows that a greater number of message-passing steps M yielded improved performance in both one-step and rollout accuracy. This is likely because increasing M allows computing longer-range, and more complex, interactions among particles. Because computation time scales linearly with M, in practice it is advisable to use the smallest M that still provides desired performance.

For (2), Figure 8.4(c,d) shows that models with unshared GN parameters in the PROCESSOR yield better accuracy, especially for rollouts. Shared parameters imposes a strong inductive bias that makes the PROCESSOR analogous to a recurrent model, while unshared parameters are more analogous to a deep architecture, which incurs M times more parameters. In practice, we found marginal difference in computational costs or overfitting, so we conclude that using unshared parameters has little downside.

For (3), Figure 8.4(e,f) shows that greater connectivity R values yield lower error. Similar to increasing M, larger neighborhoods allow longer-range communication among nodes. Since the number of edges increases with R, more computation and memory is required, so in practice the minimal R that gives desired performance should be used.

For (4), we observed that rollout accuracy is best for an intermediate noise scale (see Figure 8.4(g,h)), consistent with our motivation for using it (see Section 8.4.3). We also note that one-step accuracy decreases with increasing noise scale. This is not surprising: adding noise makes the training distribution less similar to the uncorrupted distribution used for one-step evaluation.

For (5), Figure 8.4(i,j) shows that the relative ENCODER is clearly better than the absolute version. This is likely because the underlying physical processes that are being learned are invariant to spatial position, and the relative ENCODER's inductive bias is consistent with this invariance.

8.5.3 Comparisons to Previous Models

We compared our approach to two recent papers which explored learned fluid simulators using particle-based approaches. Li et al. (2018b)'s DPI studied four datasets of fluid, deformable, and solid simulations, and presented four different, distinct architectures, which were similar to Sanchez-Gonzalez et al. (2018)'s, with additional features such as as hierarchical latent nodes. When training our GNS model on DPI's BOXBATH domain, we found it could learn to simulate the rigid solid box floating in water, faithfully maintaining the stiff relative displacements among rigid particles, as shown Figure 8.4(k) and this video. Our GNS model did not require any modification—the box particles' material type was simply a feature in the input vector—while DPI required a specialized hierarchical mechanism and forced all box particles to preserve their relative displacements with each other. Presumably the relative ENCODER and training noise alleviated the need for such mechanisms.

Ummenhofer et al. (2020)'s CConv propagates information across particles[8] and uses particle update functions and training procedures which are carefully tailored to modeling fluid dynamics (e.g., an SPH-like local kernel, different sub-networks for fluid and boundary particles, a loss function that weights slow particles with few neighbors more heavily). Ummenhofer et al. (2020) reported CConv outperformed DPI, so we quantitatively compared our GNS model to CConv. We implemented CConv as described in its paper, plus two additional versions which borrowed our noise and multiple input states, and performed hyperparameter sweeps over various CConv parameters. Figure 8.4(m) shows that across all six domains we tested, our GNS model with default

[8]The authors state CConv does not use an explicit graph representation. However we believe their particle update scheme can be interpreted as a special type of message-passing on a graph.

hyperparameters has better rollout accuracy than the best CConv model (among the different versions and hyperparameters) for that domain. In this comparison video, we observe than CConv performs well for domains like water, which it was built for, but struggles with some of our more complex materials. Similarly, in a CConv rollout of the BOXBATH DOMAIN the rigid box loses its shape (Figure 8.4(l)), while our method preserves it.

Chapter 9

Bipartite Dynamic Representations for Abuse Detection

In the final Chapter of Part III, we apply GNNs to the problem of abuse detection on e-commerce platforms. We consider a bipartite graph formed by users who interacts with products through edges. The graph evolves over time as users interact with more products. We formulate abuse detection as a node classification problem on the dynamic graph. The proposed BiDyn algorithm achieved signficant performance gain over traditional abuse detection methods, and is able to scale to real-world data with hundreds of millions of nodes at Amazon.

9.1 Introduction

Detecting abuse is an important problem in online social communities and e-commerce websites. In online social communities (such as Reddit or Wikipedia), propagating misinformation, trolling, and using offensive language are considered as abuse (Cheng et al., 2015; Shah et al., 2016). In e-commerce websites, abuse consists of fradulent activities such as artificially raising the search ranking of an item via fake reviews or purchases (Shah et al., 2016). Such abusive behavior reduces user trust, engagement, and satisfaction.

The online activity of abusive users tends to have significantly different interaction patterns compared to genuine users. For example, in e-commerce websites, abusive customers' engagement with items (e.g., daily clicks or purchases) follows fluctuating patterns (Akoglu et al., 2010). Similar patterns are observed in online forums, where abusive users target a small set of articles by posting harmful comments (Cheng et al., 2015). In the above scenarios, an accurate machine learning approach is required to model the structural and temporal dynamics of users.

There are three major challenges in detecting abuse in real-world applications. (1) Abuse in online communities or e-commerce websites manifests itself over time, giving rise to *dynamic* graph structure. Common abusive behaviors, such as continuously clicking on an item to boost its ranking, and offering such fraudulent services to several abusive items (Shah et al., 2016), involve both temporal and graph-structured aspects of the data. Thus, it is crucial to effectively combine both sources of information. (2) The real-world abuse detection problem in industry requires training and prediction on extremely large-scale graph datasets containing hundreds of millions of nodes and edges (Ying et al., 2018b). (3) It is prohibitively costly to obtain human annotated labels for abusive users to train abuse detection models, and this difficulty is compounded by the rarity of abusive behavior (Adhikari et al., 2021). Hence, the model should be able to learn from a very small amount of annotated data.

There is a plethora of existing work on dynamic graph models (Xu et al., 2020; Kumar et al., 2019; Trivedi et al., 2019; Dai et al., 2016b; Wang et al., 2021). However, the challenge is to combine the information at scale while maintaining model performance. For instance, dynamic graph modeling methods that model the full interrelationships between users and items incur great runtime costs, limiting them to small datasets (Dai et al., 2016b). Another line of work based on graph neural networks (GNNs) (Xu et al., 2020; Adhikari et al., 2021; Rossi et al., 2020a) is limited to shallow networks due to the exponential memory cost of increasing model depth with minibatch training. On the other hand, scalable graph modeling techniques based on random walk propagation (Klicpera et al., 2019b) are able to efficiently handle large-scale graphs, but they do not capture dynamic edge feature information and therefore cannot be extended to dynamic graph settings.

In this paper, we address the above challenges and propose a novel dynamic graph model BiDyn focused on abuse detection.[1] We apply our work to a large e-commerce dataset as well as to public datasets. Our work has the following components:

Joint modeling of time and graph information: We introduce an efficient neural architecture for dynamic graph data on massive graphs called **BiDyn** (Bipartite Dynamic Representations). BiDyn can handle both static and dynamic features via recurrent (RNN) and graph neural network (GNN) modules. We propose compact feature representations and minibatching practices to ensure low memory usage, and set aggregations to handle simultaneous events in a dynamic graph. We outperform the most recent dynamic graph baselines (TGAT and JODIE) by up to **14%** in AUROC on an abuse classification task in both open and proprietary datasets.

Scalable training scheme: We introduce a scalable alternating training scheme for our model

[1]Project website with data and code: http://snap.stanford.edu/bidyn

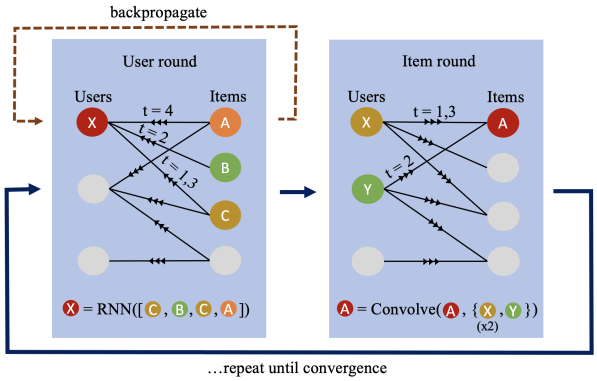


Figure 9.1: BiDyn iteratively learns embeddings through alternating item and user updates. In the item round, a graph convolution is applied to each item's neighbors. In the user round, a recurrent neural network is applied to each user's neighbors. This training scheme improves efficiency by 10x over popular baselines, and allows BiDyn to run on datasets with more than 100M edges.

that is applicable to general node classification on dynamic bipartite graphs (Figure 9.1). The training scheme evolves node embeddings by training on user and item nodes in alternating optimization rounds. Training on one class of nodes at a time allows for training of deep non-linear GNNs without the memory expense of end-to-end training. Our novel training scheme makes our model **10X** more memory-efficient than popular baselines, making training of deeper models feasible, thus leading to significant performance gains.

Self-supervised pretraining framework: We tackle the problem of label sparsity by leveraging inductive biases from domain knowledge. We propose a memory-efficient, self-supervised pretraining task that simultaneously combines sequence and graph autoencoder objectives, allowing the model to generalize from limited training labels. We demonstrate that this task provides an initial separation of abusive and normal nodes before the main training stage, providing a significant performance boost.

The rest of the paper is organized as follows. We summarize the related work in Section 9.2. We define our problem and notation in Section 9.3. We also provide a motivating analysis for our model, via observations from a subsampled e-commerce dataset in this section. Our model along with a scalable training mechanism and pretraining routines is described in Section 9.4. We provide extensive empirical evidence demonstrating the practical use of our method in the presence of a) large datasets and b) label sparsity when compared to several baselines on multiple datasets in Section 9.5. We explain practical deployment strategies in Section 9.6.

9.2 Related Work

Dynamic graph modeling. General machine learning models have been introduced for dynamic graph data (Xu et al., 2020; Kumar et al., 2019; Trivedi et al., 2019; Dai et al., 2016b; Wang et al., 2021; Rossi et al., 2020a). However, due to the high number of events associated with each node compared to static graphs, dynamic graph models suffer from high memory requirements needed to maintain a wide receptive field in both time and graph. Existing models either do not scale to large graphs (Dai et al., 2016b) or make concessions by not propagating gradients back in time (Kumar et al., 2019; Trivedi et al., 2019), or by using a very small receptive field in large graphs (Xu et al., 2020; Wang et al., 2021). As a result, existing models either cannot be applied to large-scale graphs, or are limited to short-term predictions (Kumar et al., 2019; Trivedi et al., 2019). In contrast, BiDyn models node behavior considering a long time range and large number of interactions (edges), crucial for high performance in abuse detection.

Scalable graph neural networks. A number of scalable static graph neural network methods have been proposed (Klicpera et al., 2019b; Wu et al., 2019; Ying et al., 2018b; Wu et al., 2020). Such methods decouple model depth from receptive field size, by either removing nonlinearities between model layers (Wu et al., 2019) or using random walks to propagate messages over long distances (Klicpera et al., 2019b; Ying et al., 2018b). These techniques reduce the runtime and memory usage of the models. However, removing nonlinearities also reduces the expressiveness of the model, potentially reducing performance for complex tasks. Furthermore, unlike previous works, BiDyn extends scalable static GNNs to the dynamic graph setting.

Anomaly detection. Unsupervised approaches have been introduced for detecting anomalous behavior both for static (Shah et al., 2016; Leung and Leckie, 2005; Noble and Cook, 2003; Akoglu et al., 2010) and dynamic (Bhatia et al., 2020; Yoon et al., 2019; Zheng et al., 2019) graphs. A general principle for anomaly detection is to look for nodes whose behavior has low probability compared to normal behavior (Chalapathy and Chawla, 2019). Such methods circumvent the need to collect expensive ground truth labels, but are limited to detecting specific behaviors from prior knowledge. This strategy is fundamentally hard for detecting abuse, since a shrewd abuser will look to "blend in" with normal users (Adhikari et al., 2021), making unsupervised anomaly detection methods perform poorly. BiDyn demonstrates that combining scalable pretraining and supervised fine-tuning effectively incorporates inductive bias from this probabilistic approach while also making use of supervised training labels.

9.3 Abuse Detection Problem

We first introduce our problem setup and the model architecture. We further demonstrate an efficient training algorithm to scale to large-scale graph datasets. Finally, we use pretraining to improve performance in rare-label scenarios, crucial for abuse detection.

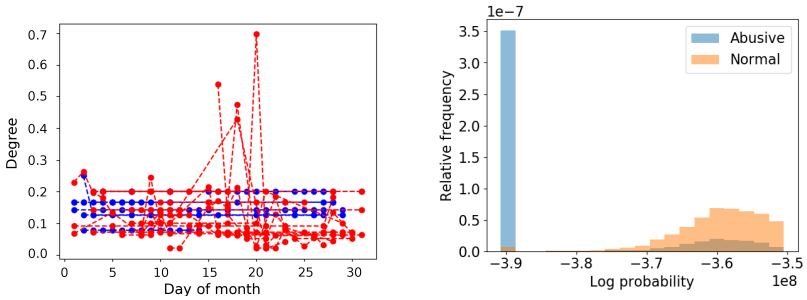
9.3.1 Problem Setup and Notation

Throughout, let $[x_1, \ldots, x_n]$ represent an ordered sequence and ; denote concatenation of vectors. We consider dynamic bipartite graphs $G = (V, E, f, g, h)$ with nodes representing users $A \subset V$ and items $B \subset V$ ($A \cup B = V, A \cap B = \emptyset$), and timestamped edges between them, of the form $(u \in V, v \in V, t \in \mathbb{R}) \in E$, where t is the timestamp, and $u \in A$ iff $v \in B$. Note that E may be a multiset if there are co-occurring events between two nodes. The graph is considered to be undirected ($(u, v, t) \in E$ iff $(v, u, t) \in E$)). We also assume without loss of generality that all timestamps occur in a time interval $[0, T_{max}]$. Let $N(u) = \{(u', v, t) \in E \ \ | u' = u\}$ denote the set of temporal neighbors of node u, also referred to as u's events. Let $w(u, v)$ denote the "weight" of the static edge between u and v, defined by the number of events that occur between them, $w(u, v) = |\{(u', v', t) \in E | u = u', v = v'\}|$. Nodes and edges may contain arbitrary feature vectors; let $f(v) \in \mathbb{R}^d$ denote the features of a user or item node v, and $g(u, v, t) \in \mathbb{R}^{d_e}$ denote the features of edge (u, v, t). The goal is to predict the binary label $y_v \in \{0, 1\}$ of each user $v \in A$. We consider a transductive setup in which a subset of the user labels are visible during training, and the goal is to predict the labels of the remaining nodes in the graph.

9.3.2 Motivating Analysis

Due to the challenges outlined in the previous section for detecting abusive behavior, it is important we understand the ways in which it manifests itself in real-world datasets. We first perform an observational study of such behavior on an anonymized e-commerce network dataset. The analysis provides insights to building an effective model for abuse detection.

Dynamic graph information. Time series information is important in modeling abuse in the e-commerce domain. Figure 9.2(a) shows the degree (a proxy for amount of activity) on each day for both normal (blue) and abusive (red) nodes. We see that the abusive nodes fluctuate much more than normal nodes – this "bursty behavior" distinguishes normal and abusive nodes. Furthermore, we see that it's important to model the whole sequence of events, taking the event ordering into account: in experiment on the Amazon abuse detection dataset, an LSTM model that uses the



(a) Temporal signals of abuse. (b) Graph signals of abuse.

Figure 9.2: (a) Degree of abusive and non-abusive users in the e-commerce network as a function of time. Abusive users (red) have more fluctuating activity levels than normal users (blue), which remain relatively constant. (b) Abusive nodes in the e-commerce network have systematically lower log-probability than normal nodes under a graph autoencoder model.

Graph statistic	Abusive	Normal
% abusive neighbors (users)	~ 1	~ 0.3
% abusive neighbors (items)	~ 1.75	~ 1.5
Degree centrality	$1.4 * 10^{-4}$	$2.1 * 10^{-5}$

Table 9.1: High level statistics of the e-commerce network. We see that there's homophily in the data: abusive nodes tend to associate more with other abusive nodes, and also see more activity.

event sequence to predict the node label outperforms a model that predicts the node label based on statistics about the distribution of purchase counts by 9.5% in AUROC.

Table 9.1 shows that abusive actors tend to interact with each other, suggesting that trustworthiness propagates between nearby nodes. Additionally, abusive nodes also tend to have high activity, with higher centrality in the network. Homophily and structural information about a node's local neighborhood are thus important features of abuse (Adhikari et al., 2021), which can be leveraged via transductive graph models.

Unsupervised modeling of user behavior. Both temporal and graph models of user behavior reveal differences between abusive and non-abusive nodes. In a toy experiment, here we use unsupervised models to provide evidence for the importance of graph structure and temporal information.

- **Abusive nodes have unpredictable event sequences**: We use an RNN autoencoder to reconstruct a given item's purchase counts for each day in the dataset. The reconstruction error (MSE) is significantly higher for abusive items, and predicting the abuse label solely based on the reconstruction error achieves 0.6 AUROC. Hence, the temporal behavior of nodes themselves provides important information about potential abusive behaviors.
- **Abusive nodes have lower likelihood under generative model**: We use a graph variational

autoencoder (Kipf and Welling, 2016b) to reconstruct the (static) graph, ignoring all temporal information. Given the static adjacency matrix M ($M_{ij} = 1$ if $\exists t : (i, j, t) \in E$ else 0), the model learns an embedding vector $z_i \in \mathbb{R}^d$ for each node such that the probability of each user node $u \in A$'s connections is given by:

$$P(u) := \prod_{j \in B} p(M_{ij}|z_i, z_j), \text{ with } p(M_{ij} = 1|z_i, z_j) = \sigma(z_i^T z_j), \tag{9.1}$$

where σ is the logistic sigmoid function. We observe that the probability assigned by the generative model is significantly lower for abusive nodes than regular nodes (p ¡ 0.001), as shown in Figure 9.2(b). Hence, a likelihood model of all nodes in the graph provides a valuable way to distinguish anomalous nodes.

9.3.3 Design Choices

The observations in Section 3.2 provide insights into important components of an abuse detection model, which we take as design choices for our model. The model should (1) combine modules which consider the temporal and the graph nature of the interaction data; (2) use a pretraining objective to model whether nodes are anomalous in their time or graph behavior, so that anomalous nodes can be automatically identified by the model even with limited training data. Next we develop a scalable model that satisfies these conditions.

9.4 Proposed Method: BiDyn (Bipartite Dynamic Representations)

BiDyn can be viewed as a stacked ensemble learning method (Wolpert, 1992) for transductive learning on dynamic graphs, achieving high performance by layering many copies of a core architecture and training them efficiently. The core of the model is a streamlined architecture combining recurrent and graph neural network components to model time and graph information. Importantly, we introduce a scalable training scheme for this model that enables training of the core model on large real-world graphs while maintaining a wide receptive field. Finally, we introduce a pretraining framework within this training scheme that allows incorporation of prior domain knowledge to tackle label sparsity. We develop an efficient dynamic graph modeling task that is particularly suited to detecting abuse and other rare events. The architecture, training method and pretraining framework combine to form a scalable, powerful method for detecting abuse in production-scale

dynamic networks.

9.4.1 Core Model Architecture

BiDyn consists of a Recurrent Neural Network (RNN) phase, followed by a Graph Neural Network (GNN) phase. We process the time and graph information through these separate components in order to allow each component to receive tailored representations of each information source. In particular, in the RNN phase, we aggregate co-occurring events to obtain a sequence of event counts over time, resulting in a highly memory-efficient node feature representation that can also capture bursty behavior. In the GNN phase, we aggregate neighbors of a node regardless of their interaction times, allowing the model to capture homophily effects and enabling memory-efficient batching by subsampling from a node's set of neighbors at every layer.

RNN phase. In the RNN phase, an RNN is applied to each node; a separate RNN is used for each node type. In the bipartite graph setting, we use RNN_A to denote the RNN component for users and RNN_B for items. Each RNN receives as input an encoding of the sequence of neighbors $N(u)$ of the given node u, in time order. In particular, for a given node u, it receives a sequence

$$[f(v_1); g(u, v_1, t_1); \phi(t_1), \ldots, f(v_k); g(u, v_k, t_k); \phi(t_k)] \tag{9.2}$$

for each $(u, v_i, t_i) \in N(u)$, where $t_1 \leq t_2 \leq \ldots \leq t_k$. Here $\phi(t_i) : \mathbb{R} \to \mathbb{R}^{D_{\text{time}}}$ is a sinusoidal encoding of the timestamp (Vaswani et al., 2017), defined per entry by

$$\begin{aligned} \phi_{2j}(t_i) &= \sin(\frac{100}{T_{max}} t_i / 10000^{2i/D_{\text{time}}}), \\ \phi_{2j+1}(t_i) &= \cos(\frac{100}{T_{max}} t_i / 10000^{2i/D_{\text{time}}}), \end{aligned} \tag{9.3}$$

where D_{time} is the dimension of the encoding. The last hidden state of the RNN is concatenated with u's features $f(u)$ to produce an RNN output h_u^0 := (RNN_A(u); f(u)) for users $u \in A$ and (RNN_B(u); f(u)) for items $u \in B$. We use a 2-layer LSTM (Hochreiter and Schmidhuber, 1997) as the RNN architecture due to its best performance in handling long-range dependencies.

GNN phase. In the GNN phase, the per-node outputs from the RNN phase are propagated throughout the *static* network to generate a prediction for each node. To compute the prediction for node u, we first compute its embedding by applying multiple GNN layers to the RNN outputs h_u^0, which are used as the initial node features to the GNN. To get from the embedding h_u^l at GNN layer l to the embedding h_u^{l+1} at layer $l + 1$, we apply a modification of SAGE convolution (Hamilton et al.,

2017c) that incorporates edge weights, setting

$$m_{v,u}^l = \text{ReLU}(W_{msg}^l(h_v^l; w(u,v)))$$
$$h_u^{l+1} = \text{ReLU}\left(W^l \left[h_u^l; \sum_{(u,v,t)\in N(u)} m_{v,u}^l\right]\right). \tag{9.4}$$

The final embeddings h_u^L are then fed through a logistic regression classifier to produce the predicted abuse probability and predicted label $\hat{y}_u$. To enable the model to fit in GPU memory, we use minibatching with random neighbor sampling in the GNN phase: at each layer, instead of considering the full set $N(u)$ of neighbors in the aggregation function, we subsample up to D elements from $N(u)$ uniformly at random.

Extension to coarse-grained timestamps. In many settings, events are timestamped on a coarse-grained scale, such as to the nearest day. Subsequently, many events have identical timestamps. For example, multiple users interact with an item on a single day, and the exact sequence in which these users interacted with the item is not known. BiDyn makes sure that the order in which simultaneous events are fed to the RNN does not affect the final model output.

We propose an extension to our RNN to make our model invariant to reordering of simultaneous events. For a given node u, the RNN will receive a sequence $[\text{AGG}(S_0), \ldots, \text{AGG}(S_{T_{max}})]$ of length T_{max} as input, where AGG is a permutation-invariant function and $S_t = \{v|(u', v, t') \in E, t' = t, u' = u\}$ is the (multi)set of u's neighbors at time t. We use the aggregation function $\text{AGG}(S_t) = (\frac{1}{|S_t|}\sum_{v\in S_t}(f(v); g(u,v,t)); |S_t|)$, concatenating the mean node and edge features with the number of events on each day, since we find that the mean generalizes better than more complex approaches such as DeepSets (Zaheer et al., 2017), while we found that the number of events is a useful feature for detecting burstiness. Note that the choice of AGG is the degree-scaling used in principal neighborhood aggregation (Corso et al., 2020) to better distinguish neighborhoods with similar features. We adopt this aggregation scheme when testing the core architecture on large datasets (e-commerce) in the experiments (RNN-GNN), which proves to be effective and memory efficient.

9.4.2 Scalable Training

While our core architecture is an effective dynamic network model, as we demonstrate in the experiments, it suffers from high memory usage when performing minibatch training on large

Algorithm 4: Alternating training of users and items

Data: $X \in \mathbb{R}^{|V| \times d}$ is node embedding matrix with rows x_u ($\forall u \in V$)

$x_u \leftarrow 0, \forall u \in V$;
while *stopping condition not reached* **do**
  for $u \in A$ ***(User round)*** **do**
    Clear gradients;
    Let $k = |N(u)|$, $\{(u, v_i, t_i)\} = N(u)$ where $t_1 \leq \ldots \leq t_k$;
    $x_u \leftarrow \text{RNN}_1([f(v_1); g(u, v_1, t_1); \phi(t_1); x_{v_1}, \ldots,;$
    $f(v_k); g(u, v_k, t_k); \phi(t_k); x_{v_k}])$;
    $L \leftarrow \text{CrossEntropy}(\text{LogisticRegression}(x_u), y_u)$;
    Backpropagate L;
  end
  for $u \in B$ ***(Item round)*** **do**
    $x_u \leftarrow \text{Convolve}(u, N(u), X)$;
  end
end
return X

datasets[2], since memory usage has complexity $O(D^L)$ with increasing number of GNN layers L, where D is the maximum number of neighbors subsampled at each layer during minibatching. The dynamic graph setting further compounds this memory problem. First, since temporal edges often represent interactions such as edits, clicks or purchases, the fanout D is often very large (order of hundreds). Second, prepending an RNN module to the GNN further exacerbates the memory cost of backpropagation, in practice limiting L to be 1 or 2 in order for the model to fit within a GPU minibatch. These difficulties combine to make training the core model in a standard end-to-end fashion impractical for large-scale, real-world networks.

We tackle this difficulty by introducing an alternative training scheme that iteratively updates node embeddings by stacking multiple copies of the core architecture, circumventing end-to-end training. We (1) extend the core architecture to a deep model that alternates between the RNN and graph convolution layers, and (2) train this model one layer at a time. Although training is no-longer end-to-end, it makes training of a much deeper model possible, which in practice results in significant performance improvement.

Figure 9.1 shows the flow of information through the model. Throughout training, BiDyn maintains a table of current embeddings of all nodes, $x_u \in \mathbb{R}^d$, and alternates between updating user and item embeddings, as shown in Algorithm 4.

User round. In the user round (left component of Figure 9.1), the RNN phase is applied to generate

[2]Existing alternatives to minibatching, such as SIGN (Rossi et al., 2020b), cannot be applied due to the dynamic RNN component.

updated user embeddings, which are saved back to the table and trained on the prediction task to update the RNN and logistic regression weights. With each event fed into the RNN, we concatenate the embedding of the associated item. Hence, the updated embedding of user $u \in A$ is computed as

$$x_u = \text{RNN}_1([f(v_1); g(u, v_1, t_1); \phi(t_1); x_{v_1}, \ldots, f(v_k); g(u, v_k, t_k); \phi(t_k); x_{v_k}]), \tag{9.5}$$

a function of the sequence of node and edge features, time encoding and node embedding for each event. The user embeddings are fed into a logistic regression layer to compute the final abuse prediction for each user. The RNN is trained by backpropagating from the predicted abuse scores (dotted line in Figure 9.1). We apply the RNN in the user round since users were observed to be the source of bursty behavior (Figure 9.2(a)).

Item round. In the item round (right component of Figure 9.1), a graph convolution (i.e. a layer from the GNN phase) is applied to generate updated item embeddings by propagating the neighboring user embeddings. Hence, the updated embedding of item $u \in B$ is computed as

$$x_u = \text{Convolve}(u, N(u), X) \tag{9.6}$$

for some graph convolution layer Convolve.

In summary, each round improves either the user or item embeddings using the updated embeddings from the previous round. This mutually recursive relationship ensures that the embeddings continually improve over time. Finally, the process terminates when a stopping condition for training is reached; in our experiments, we train for a fixed number of training rounds, then evaluate the model with lowest validation loss.

Graph convolution without learnable parameters. Since there is no gradient flow between the user and item round, we opt to simplify the GNN convolution layer in the item round to a layer without learnable parameters, while the user round uses a learnable RNN to characterize user preferences through aggregation of items over time. Fixed graph convolutions have been shown to be effective in recent simplified graph neural network methods (Wu et al., 2019; Klicpera et al., 2019b). In the spirit of these methods, we divide the separate roles of traditional graph convolution operations into separate model components, using a fixed convolution operation to widen the receptive field of the model, and successive applications of the RNN component to

	GNN	BiDyn	APPNP-I
Memory usage per batch	$O(D^L)$	$O(D)$	$O(1)$
Receptive field radius	L	L	L
Number of nonlinearities	$O(L)$	$O(L)$	0

Table 9.2: The scalable training scheme of BiDyn is memory-efficient, with memory cost for each training batch comparable to that of a scalable static GNN, APPNP-I. Simultaneously, it is capable of a wide receptive field and deep network, enabling high performance. Here D is the maximum number of neighbors subsampled at each layer during minibatching, and L is the number of layers in the model.

increase the neural network depth. Hence, we define

$$\text{Convolve}(u, N(u), X) = \alpha x_u + (1-\alpha) \sum_{v \in N(u)} x_v, \tag{9.7}$$

a sum aggregation with running average update. The use of sum aggregation can be viewed as a multi-dimensional version of the update step of the HITS algorithm for identifying trustworthy nodes in a network (Kleinberg, 1999). We demonstrate in the experiments that sum aggregation performs better than more complex alternatives, such as autoencoders, while also being much more efficient. We also compare against mean aggregation, which performs better in some cases, in the experiments. The hyperparameter $\alpha \in [0, 1]$ controls the rate of diffusion of information throughout the network; thus, a high α will more heavily prioritize nodes closer to u, better preserving the local neighborhood. We also design a convolution operation in the setting with coarse-grained timestamps:

$$\text{Convolve}(u, N(u), X) = \alpha x_u + (1-\alpha) \frac{1}{|N(u)|} \sum_{t=0}^{T_{\max}} \frac{1}{|S_t|} \sum_{v \in S_t} x_v \tag{9.8}$$

with the α and S_t as defined from before. We choose this convolution empirically, with ablations in the experiments. Overall, only the RNN component of BiDyn contains learnable parameters, hence backpropagation only occurs during the user round.

Memory usage. As shown in Table 9.2, the stacked ensemble training scheme enables the model to take advantage of many of the performance benefits of a deep model without the memory expense of end-to-end training. The memory usage of BiDyn approaches that of the inference-time variant of APPNP (denoted by APPNP-I) (Klicpera et al., 2019b), a scalable GNN model that trains a classifier on individual nodes and uses random walk propagation at inference time only. Each round of backpropagation for BiDyn only requires querying a node's direct neighbors, only uses linear memory $O(D)$, and minibatches can easily fit on a GPU, even with very large number of

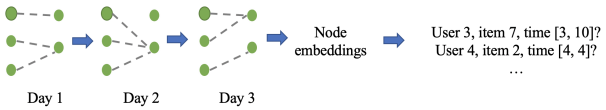


Figure 9.3: The "random access query" pretraining task learns static node representations that can be used to make dynamic predictions about whether an event occurs between a user and item in a given time range.

neighbors D^3. At the same time, the receptive field of the model is not bounded by memory constraints, since each user or item round increases the receptive field radius by one. Furthermore, each user round adds at least one additional nonlinearity for all embeddings that pass through it, so this stacking approach naturally receives some of the performance benefits of a deeper model.

9.4.3 Pretraining Framework

We further propose a scheme for pretraining in this alternating training framework, in order to benefit from unsupervised objectives. To pretrain on an auxiliary objective, simply train on this auxiliary objective during the user round for a number of epochs, then switch to the main objective. This method effectively creates a prior on both the embeddings and the network weights.

Due to the label sparsity common to abuse tasks, we propose an unsupervised pretraining task that can serve as a prior. We develop a pretraining task that draws from the anomaly detection approach to abuse detection, which leverages the phenomenon that abusive nodes will have lower probability than normal nodes under normal nodes (Observation 1) (Chalapathy and Chawla, 2019). Hence, we propose a probabilistic model of user behavior in the dynamic graph setting under which abusive users will be distinct from normal users, thereby providing an initial separation of nodes for classification. Since user activity on online platforms often includes both time series and graph information, we propose a simple pretraining task that generalizes probabilistic time series and graph models. We assume a probabilistic model of user behavior in which each node u has an associated latent vector z_u, and the distribution of events (dynamic edges) between two nodes $u \in A$ and $v \in B$ is a general point process that depends only on z_u and z_v:

[3]Note that storage requirements for the graph G and the embedding matrix X are not included in the memory requirements for any of the methods in Table 9.2, since they they do not need to be stored on GPU during backpropagation.

$$\log P(G|Z) = \sum_{u \in A} \sum_{v \in A} \Big(\sum_{t:(u,v,t) \in N(u)} \log \lambda(z_u, z_v, t) + \int_0^{T_{\max}} (1 - \lambda(z_u, z_v, t)) dt \Big) \tag{9.9}$$

where Z denotes the matrix of all latent vectors z_u, and

$$\lambda(z_u, z_v, t) := \lim_{\Delta t \to 0} \frac{1}{\Delta t} P(\exists t' \in [t, t + \Delta t] : (u, v, t') \in E | z_u, z_v, t) \tag{9.10}$$

gives the intensity of an event occurring between nodes u and v at time t, given their latent vectors and the observed history of events between the two nodes until time t.

We propose the following "random access query" pretraining task which models dynamics on several levels of resolution. Our task is to learn an embedding z_u for each node in the graph. The task is to predict, for a user-item pair $(u \in A, v \in B)$ and time interval $[t_1, t_2]$, and given u's history of events until time t_1, whether an edge appears between (u, v) in the desired time interval. We mask out all edges associated with u occurring at any time $t \geq t_1$. We use a multi-layer perceptron (MLP) to estimate the probability of the edge appearing within this time interval:

$$P(\exists (u, v, t) \in E, t \in [t_1, t_2] | z_u, z_v, t_1) = \sigma(\text{MLP}(z_u, z_v, \phi(t_1), \phi(t_2))) \tag{9.11}$$

In principle, modeling the behavior within every time interval allows us to estimate λ (by taking $t_2 \to t_1$), hence the task fully models the time series between two nodes, while being extremely lightweight in terms of memory consumption (requiring no backpropagation through time). Furthermore, the random access objective allows for simultaneous modeling of different time scales: a model trained on this objective can perform both time series and graph autoencoder tasks, predicting events at a specific time when $t_1 \approx t_2$ and predicting static links when $t_1 = T_{\text{min}}$ and $t_2 = T_{\text{max}}$.

The objective, given an edge $(u, v, t) \in E$ and arbitrary time interval $[t_1, t_2]$, is as follows:

$$\begin{aligned} L(u, v, t_1, t_2) = {} & M(z_u, z_v, f(u), f(v), \phi(t_1), \phi(t_2)) \\ & + M(z_u, z_{v'}, f(u), f(v'), \phi(t_1), \phi(t_2)) \end{aligned} \tag{9.12}$$

Here M is the cross entropy loss of the MLP, where the label is whether an edge truly appears

between the given user and item in the given time interval. v' is a randomly chosen (perturbed) item node (hence the second term represents negative examples). We encode t_1 and t_2 using the sinusoidal encoding ϕ used previously. We optimize this objective for uniformly randomly selected time intervals and perturbed item nodes.

9.5 Abuse Detection Experiments

We perform many comparisons to demonstrate the efficacy of each component of our model. We demonstrate that BiDyn effectively predicts abuse on several domains, thus the approach is general.

1. BiDyn's dynamic graph architecture shows favorable performance in the transductive prediction setting compared to both ablations and alternative dynamic graph methods.
2. Alternating training improves performance across datasets compared to end-to-end training, while using much less GPU memory, enabling higher throughput for production-scale use cases.
3. Our pretraining framework and task improve performance compared to alternatives.

Experimental setup. We consider a transductive framework in which the entire dynamic graph is visible during training, but only a subset of the user labels are visible. The goal is to predict the labels of the remaining users in the graph. 5% of the labels are visible during training; the remainder are split equally between validation and test. We train all models on all datasets for 20 epochs and take the test performance on the model with lowest validation loss.

Datasets. We perform experiments on the following proprietary (**e-commerce**) and open dynamic graph datasets (**Wikipedia**,**Reddit**).

- **e-commerce**. We used anonymized and subsampled data from an e-commerce website. The data is subsampled in a manner so as to be non-reflective of actual production traffic. The data consists of 500K users (1K positive), 6M items and 113M interactions aggregated over an arbitrary 2 month window. Labels indicate users marked for abusive behavior.
- **Wikipedia**. The Wikipedia dataset consists of users linked to the pages they edit. Positive users indicate those who were banned in the data collection period. There are 8227 users (187 positive), 1000 pages and 157474 interactions.
- **Reddit**. This dataset consists of users linked to the communities they participate in. Positive users indicate those who were banned in the data collection period. There are 10000 users (333 positive), 1000 communities and 672447 interactions.

Baselines. We compare against alternative dynamic graph models, as well as models that only use graph or time information.

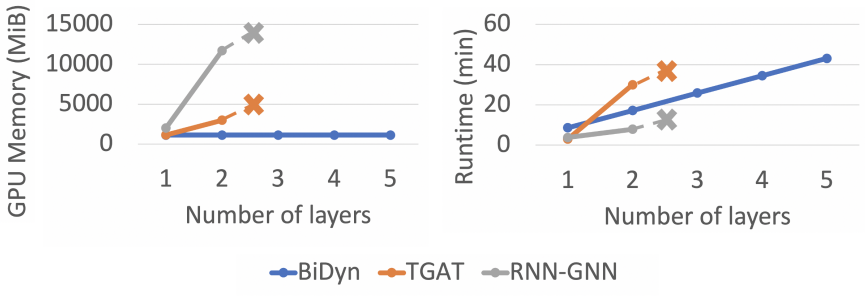


Figure 9.4: GPU memory usage (left) and runtime (right) for BiDyn, RNN-GNN and TGAT. TGAT and RNN-GNN run out of memory for 3 or more layers (denoted by X), while memory usage of BiDyn remains constant with increasing depth. Furthermore, BiDyn achieves comparable or lower runtime than the baselines.

- **Dynamic graph models**. We compare against TGAT (Xu et al., 2020), JODIE (Kumar et al., 2019), DyRep (Trivedi et al., 2019) and TGN (Rossi et al., 2020a), four recently-proposed dynamic graph models. TGAT uses a GNN with temporal attention layers to attend to a node's history. JODIE predicts trajectories of node embeddings over time as a form of self-supervision. DyRep models dynamics jointly on small and large time scales. TGN makes use of memory modules and graph-based operators. We adapt them to the transductive setting by training their dynamic node embeddings to predict the abuse labels (details in Appendix). We also compare against the core model architecture, RNN-GNN, with standard end-to-end training, using a 2-layer GNN (the maximum that can fit in GPU memory for the e-commerce dataset).
- **GNN models**. We compare against a static GNN to demonstrate that incorporating time information can improve performance in the transductive setting.
- **Recurrent models**. We compare against an RNN model with time encoding to demonstrate the value of a semisupervised approach (using information from neighboring nodes).

See the Appendix for implementation details and extended experimental results.

9.5.1 Model Comparison

Table 9.3 shows the performance of BiDyn in AUROC compared to the baselines. We compare different graph convolution variants: BiDyn (sum) represents the convolution in equation (6), BiDyn (mean) uses equation (7) but takes the mean $\frac{1}{|N(u)|}\sum_{v\in N(u)} x_v$ rather than the sum $\sum_{v\in N(u)} x_v$ over embeddings, and BiDyn (coarse) is the convolution in equation (8).

We evaluate the e-commerce data in both the regime where 5% of labels are seen in training ("e-com (5%)") and 50% of labels are seen in training ("e-com (50%)") (the 5% regime is used for the remainder of the experiments).

	Method	e-com (5%)	e-com (50%)	Wikipedia	Reddit
Ablation	GNN	+0.0	+0.0	69.6 ± 0.2	53.7 ± 1.8
	RNN	+0.0 ± 0.4	+1.8 ± 0.2	78.0 ± 2.3	51.3 ± 4.7
	RNN-GNN	-2.2 ± 0.5	+1.9 ± 0.5	70.0 ± 0.2	53.8 ± 2.2
Baselines	TGAT	-4.0 ± 0.1	-2.5 ± 0.6	73.6 ± 4.7	51.5 ± 2.9
	TGN	OOM	OOM	49.0 ± 0.6	**67.0 ± 0.6**
	DyRep	OOM	OOM	52.5 ± 0.2	61.4 ± 0.8
	JODIE	OOM	OOM	53.0 ± 0.5	61.2 ± 0.4
Ours	BiDyn (coarse)	+1.2 ± 0.5	**+3.9 ± 0.1**	—	—
	BiDyn (mean)	—	—	80.5 ± 2.3	56.0 ± 3.3
	BiDyn (sum)	—	—	86.5 ± 1.6	46.8 ± 3.9
	+ pretraining	**+4.5 ± 0.5**	+1.6 ± 0.2	**87.5 ± 0.6**	50.5 ± 2.6

Table 9.3: Model comparison (AUROC; absolute gain for e-commerce) classifying whether users in the network are abusive. For e-commerce, we report model performance relative to a GNN, while for public datasets we report absolute numbers. We observe that the scalable training scheme of BiDyn gives performance benefits over end-to-end training of an RNN-GNN architecture, as well as baseline dynamic graph models. Pretraining leads to further improvements in the regime with limited training labels. Many baselines do not scale, running out of memory (OOM) on e-commerce. '–' denotes entries that do not apply, e.g. BiDyn (coarse) only applies to datasets with coarse-grained timestamps.

Dynamic graph models. We see that BiDyn (even with standard end-to-end training), and indeed the pure RNN model, outperforms TGAT in the transductive setting. Hence, we confirm the importance of modeling the entire time series of events, which TGAT's attention layers do not sufficiently capture. By prepending an RNN making use of a compact event representation, we ensure that BiDyn models the entire time series while being light on memory usage.

Furthermore, alternating training leads to competitive or even improved performance compared to end-to-end training, despite using a GNN layer without learnable parameters. We attribute this improvement to the increased model depth made possible by a stacked model. While JODIE navigates the performance-scalability tradeoff by disconnecting gradient flow through time, BiDyn disconnects gradient flow across the layers of the stacked model, which is a more effective compromise in the transductive setting.

Finally, the performance of TGN, DyRep and JODIE are varied, with the models failing to learn the Wikipedia task (achieving near random performance) but achieving higher performance than BiDyn on the Reddit task. We hypothesize that BiDyn is more successful on Wikipedia due to its success in domains with greater network homophily: the Pearson correlation between the labels of pairs of users who share an item is 3% on Wikipedia, and only 0.5% on Reddit (different with $p < 0.001$). Despite their advantages on the Reddit domain, these models are unable to scale to the larger e-commerce dataset, running out of memory (OOM) when trained on GPU and taking over 300 hours per epoch when trained on CPU, hence cannot be used in a web-scale abuse detection

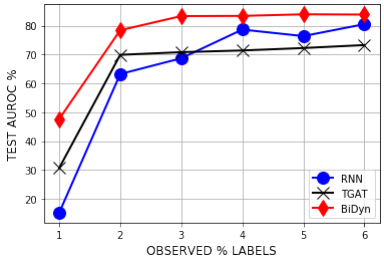


Figure 9.5: Comparison between BiDyn and baselines (RNN, TGAT) on AUROC by varying the amount of observed node labels on Wikipedia. Even when we observe less then 5% of node labels, BiDyn achieves high AUROC compared to baselines, hence it is robust to rare training labels.

pipeline. Appendix B also demonstrates that BiDyn performs comparably to these baselines on another dynamic graph dataset, MOOC (Kumar et al., 2019).

Graph models. BiDyn outperforms a static GNN (the GNN phase of the Core Model Architecture), hence demonstrating the value of incorporating time information for abuse detection.

Recurrent models. BiDyn outperforms the RNN model with time encoding (the RNN phase of the Core Model Architecture), hence leveraging information from nearby nodes is advantageous for dealing with label sparsity.

Pretraining. Pretraining leads to further performance improvement, indicating that incorporating domain knowledge via the random access query objective helps to deal with label sparsity. We select the best-performing variant of BiDyn on each dataset (coarse, mean or sum) for pretraining.

9.5.2 Robustness to Sparse Training Data

Figure 9.5 compares the effectiveness of BiDyn compared to TGAT and RNN at different amounts of supervision, varying the percentage of labeled nodes seen in training. We chose these baselines as they were the two best competitors to BiDyn on the Wikipedia dataset (Table 9.3). BiDyn outperforms the baselines when the number of training samples seen is very low ($< 5\%$). Moreover, when the number of samples is low, the methods that rely on graph information (BiDyn, TGAT) outperform RNN. For abuse detection, performing well with limited supervised data is crucial.

9.5.3 Memory and Time Comparison

Figure 9.4 shows BiDyn uses 10x less memory than the base RNN-GNN model (which already uses memory-saving feature representations) on the e-commerce dataset. These memory savings allow it to fit comfortably on a GPU instance and hence be efficiently trained in a production setting. Note in particular that it does not use extra memory per "layer" (training round), while

	e-com
Autoencoder	-13.4
Predict item labels	-3.8
BiDyn (sum)	-7.1
BiDyn (mean)	-3.3
BiDyn (coarse)	**0.0**

	Wikipedia
Autoencoder	79.8 ± 2.8
BiDyn (mean)	80.5 ± 2.3
BiDyn (sum)	$\mathbf{86.5} \pm \mathbf{1.6}$

Table 9.4: Comparison of different objectives and aggregation schemes in the item step (relative change in AUROC). Coarse-grained convolution performs best on e-commerce, while sum aggregation performs best on Wikipedia.

TGAT and RNN-GNN increase rapidly per layer. Its low memory usage allows for larger receptive fields and less subsampling during minibatching, accounting for its higher performance. Note that TGAT uses less memory than RNN-GNN, but does not perform as well in our transductive problem setting, which prioritizes modeling the whole time series associated with each node. Furthermore, BiDyn achieves comparable or lower runtime than the baselines. When comparing memory and runtime, all models are trained with the same batch size of 64. A 128-dimensional feature is used for user nodes, and no features are used for item nodes. Though RNN-GNN (barely) fits on GPU in 2 layers, the small batch size of 64 (which was used to provide an common ground for comparison) is too small to enable high throughput in a production setting, and the memory footprint is too large to accommodate higher-dimensional node features.

9.5.4 Architecture Ablation

Table 9.4 compares different choices for the convolution operation. In "predict item labels", the model must predict the binary label of the item for a separate but related item classification task. In "autoencoder", the layer takes in an RNN sequence of its neighbors' embeddings as input, outputs an embedding and uses that embedding as the first hidden state of a decoder RNN whose task is to output the same sequence of neighbor embeddings, with minimum mean-squared error. The coarse-grained convolution operation of equation (7) performs best on the e-commerce dataset. The autoencoder objective performs worst, which could be explained by the high MSE of the learned model, which suggests the task may be too difficult to learn usable representations from.

9.5.5 Pretraining Validation

Table 9.5 compares the random access query pretraining task with a standard (static) graph autoencoder objective (Kipf and Welling, 2016b) and finds that the random access query task leads to superior performance on the abuse detection task. Thus, it is important to jointly model the

	e-commerce	Wikipedia
No pretraining	0.0	86.5 ± 1.6
Graph autoencoder	-0.5 ± 2.2	86.7 ± 1.9
Random access	**+3.3 ± 0.5**	**87.5 ± 0.6**

Table 9.5: Comparison of performance on the abuse task after pretraining on different self-supervised tasks. The random access query pretraining task boosts performance on the abuse task compared to no pretraining, while the graph autoencoder task does not improve performance, hence self-supervision on both the temporal and graph structure provides an important advantage on the abuse detection task.

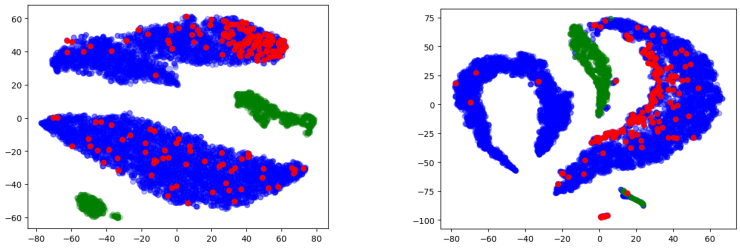


Figure 9.6: Pretraining provides initial separation between abusive and normal node embeddings on Wikipedia (left), which is further enhanced through fine-tuning on the abuse detection task (right). Here, blue nodes are normal users, red nodes are abusive users and green nodes are items. Visualized with TSNE.

temporal and graph structure for the abuse detection task. Figure 9.6 confirms that the pretraining task provides a useful prior by creating an initial separation of nodes: we see a separation between abusive and normal nodes after pretraining, which is further accentuated after regular training as the model fine-tunes the embeddings for the abuse detection task.

9.6 Deployment Strategies

BiDyn can be deployed to detect abuse on e-commerce websites, social media, etc. The training method is scalable, and existing GPU infrastructure can be easily used to train and deploy these models into production. Since periodic retraining of graph models as the graph data evolves is essential in practice, efficient GPU training and light memory usage can make it easier to retrain more often for best performance.

One can also envision a human-in-the-loop scenario, where BiDyn determines a set of users to be banned. These users can be sent to a human auditor who makes the final decision. This process will help improve the yield of human auditors, who can rely on the ML model to send them relevant users to audit, rather than (potentially) random users. Augmenting machine learning models with a human to take the final labeling decision has been explored before (Rao et al., 2010).

Chapter 10

Conclusions

This dissertation covers the topics of graph representation learning and graph neural networks (GNNs) through the general framework of GraphSAGE (PART I), the various GNN model improvements to achieve higher expressive power (PART II), as well as the real-world applications in diverse domains (PART III).

In Chapter 2, we explained limitations of traditional graph representation learning methods (e.g. Node2Vec, DeepWalk), and how the GraphSAGE framework brings remedy to these limitations. More importantly, we illustrate the key advantages of GraphSAGE: its ability to incorporate node, edge, graph features, its inductive power, and its framework flexibility. GraphSAGE defines the neighborhood deinition (sampling) and feature aggregation steps for each layer. This allows later GNN architectures to make improvements to the design and operations of each of the steps under the GraphSAGE framework.

In Chapter 3, we show that the GraphSAGE framework is further empowered by GNNExplainer, an explanation framework for explaining any trained GNN model under the GraphSAGE framework. Model explanation provides a powerful way for domain experts to gain understanding of the underlying reasons for GNN model predictions.

PART II focuses on extensions of the GraphSAGE framework to create highly expressive GNN models for a variety of tasks. In Chapter 4, we leveraged the idea of learnable hierarchical pooling to obtain graph representations from GNN node classifications. In Chapter 5, we tackled the graph learning problem by leveraging the inductive bias of tree-like structure in many graph datasets, and design a special GNN (HGCN) that maps nodes in graph into hyperbolic embedding spaces to better preserve the hierarchical property of the graph. Finally, in chapter 6, we demonstrate the combination of multi-hop neighborhood and attention aggregation that achieves state-of-the-art performance on most of the standard graph benchmarks. Note that despite its complexity, the

algorithm still fits well into the GraphSAGE framework.

Finally, PART III demonstrates the wide applications of GNNs in real-world scenarios. Chapter 7 introduces PinSAGE, the first deployed web-scale GNN on hundreds of millions of nodes and billions of edges, used in the recommender system at Pinterest, incorporating many innovations based on the GraphSAGE framework. Chapter 8 demonstrates a new exciting direction of using GNNs to perform simulations for graphics and physics. The learned model is highly efficient at inference time and highly generalizable to vastly different test scenarios. Chapter 9 tackles the challenging problem of abuse detection on e-commerce platforms. We employ a new dynamic GNN architecture, an efficient training strategy and pretraining to achieve signficantly better performance for abuse detection, and the system is currently deployed at Amazon.

Overall, graph neural network has achieved unprecedented success in so many different areas and is still a rapidly growing research area in both academia and industry. I am very fortunate to have worked in this area, and I am excited to continue explore related future directions. Thanks to its general representation, many objects and concepts can be represented as graphs, and I believe that learning on graphs will become even more popular as people discover innovative ways to model the world with different kinds of graphs. Hereby I would also like to encourage researchers and prospective students to explore the potential of graph learning, and advance this field together.

10.1 Future Directions

In the future, I hope to explore many exciting aspects of graph representation learning. In theoretical aspects, I am interested in exploring the expressive power of graph neural networks, potentially combining principled approaches that capture graph topology, such as persistent homology. In algorithm aspects, I aim to develop scalable novel architectures that can learn representations at different levels of abstractions including nodes, relation, network motifs, subgraphs and entire graphs. Finally, in application perspective, I am excited to continue researches that apply graph neural network techniques in industrial use cases and scientific advances. My main future directions include the following.

Knowledge Graph Reasoning. Knowledge graph is an essential component in human reasoning. I am not only interested in conventional reasoning tasks such as link prediction, but also more complex reasoning that further combines high-order graph structure and natural language information. My preliminary exploration in this direction includes hierarchical reasoning tasks on knowledge graphs, and multi-hop question answering using graphs of documents and sentences. I believe that progress in this field is the next step towards human-level reasoning via machine learning.

Biomedical Discovery. The recent pandemic motivates me to look into biomedical domain applications. Many real-world biological networks are not only heterogeneous but also hierarchical in structure (e.g. Gene Ontology, Reactome). I hope to further advance techniques in hierarchical graph neural networks, especially in the context of heterogeneous networks with diverse node and edge types, and achieve the goal of discovery in biological and medical domains, such as novel drug discovery and drug combinations.

Embedding Space for Graphs. My work on hyperbolic graph convolutional networks (Chapter 5) have demonstrated the effective use of embedding geometry to capture a variety of complex graph topology. These techniques include shape geoemtry as embeddings, such as boxes and cones, as well as non-Euclidean geometries such as hyperbolic and spherical geometry. I believe that research in representation geometry has potential to produce breakthrough in current graph representation learning, due to its increased expressive power as a result of the inductive biases of various geometries.

Bibliography

2018. *Graph Nets Library*. DeepMind.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

A. Adadi and M. Berrada. 2018. Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI). *IEEE Access*, 6:52138–52160.

Aaron B Adcock, Blair D Sullivan, and Michael W Mahoney. 2013. Tree-like structure in large social and information networks. In *ICDM*. IEEE.

J. Adebayo, J. Gilmer, M. Muelly, I. Goodfellow, M. Hardt, and B. Kim. 2018. Sanity checks for saliency maps. In *NeurIPS*.

Bijaya Adhikari, Liangyue Li, Nikhil Rao, and Karthik Subbian. 2021. Finding needles in heterogeneous haystacks. In *IAAI*.

Leman Akoglu, Mary McGlohon, and Christos Faloutsos. 2010. Oddball: Spotting anomalies in weighted graphs. In *PAKDD*.

Roy M Anderson and Robert M May. 1992. *Infectious diseases of humans: dynamics and control*. Oxford university press.

A. Andoni and P. Indyk. 2006. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS*.

Sanjeev Arora, Yingyu Liang, and Tengyu Ma. 2017. A simple but tough-to-beat baseline for sentence embeddings. In *ICLR*.

M. Gethsiyal Augasta and T. Kathirvalavakumar. 2012. Reverse Engineering the Neural Networks for Rule Extraction in Classification Problems. *Neural Processing Letters*, 35(2):131–150.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *ICLR*.

Ivana Balazevic, Carl Allen, and Timothy Hospedales. 2019. Tucker: Tensor factorization for knowledge graph completion. In *EMNLP*.

T. Bansal, D. Belanger, and A. McCallum. 2016. Ask the GRU: Multi-task learning for deep text recommendations. In *RecSys*. ACM.

Trapit Bansal, Da-Cheng Juan, Sujith Ravi, and Andrew McCallum. 2019. A2n: Attending to neighbors for knowledge graph inference. In *ACL*.

Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. 2016. Interaction networks for learning about objects, relations and physics. In *NeurIPS*.

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. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv:1806.01261*.

Mikhail Belkin and Partha Niyogi. 2002. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *NeurIPS*.

Jan Bender and Dan Koschier. 2015. Divergence-free smoothed particle hydrodynamics. In *SIGGRAPH*. ACM.

Y. Bengio, J. Louradour, R. Collobert, and J. Weston. 2009. Curriculum learning. In *ICML*.

Austin R Benson, David F Gleich, and Jure Leskovec. 2016. Higher-order organization of complex networks. *Science*, 353(6295):163–166.

R. van den Berg, T. N. Kipf, and M. Welling. 2017. Graph convolutional matrix completion. *arXiv preprint arXiv:1706.02263*.

Siddharth Bhatia, Bryan Hooi, Minji Yoon, Kijung Shin, and Christos Faloutsos. 2020. Midas: Microcluster-based detector of anomalies in edge streams. In *AAAI*.

Monica Bianchini, Marco Gori, and Franco Scarselli. 2001. Processing directed acyclic graphs with recursive neural networks. *IEEE Transactions on Neural Networks*, 12(6):1464–1470.

Silvere Bonnabel. 2013. Stochastic gradient descent on riemannian manifolds. *IEEE Transactions on Automatic Control*.

Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. In *NeurIPS*.

K. M. Borgwardt and H.-P. Kriegel. 2005. Shortest-path kernels on graphs. In *ICDM*.

Karsten M. Borgwardt, Cheng Soon Ong, Stefan Schönauer, S. V. N. Vishwanathan, Alex J. Smola, and Hans-Peter Kriegel. 2005. Protein function prediction via graph kernels. *Bioinformatics (Oxford, England)*.

A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *CIKM*.

M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. 2017. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*.

J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. 2014a. Spectral networks and deep locally connected networks on graphs. In *ICLR*.

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. 2014b. Spectral networks and locally connected networks on graphs. In *ICLR*.

Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2015. Grarep: Learning graph representations with global structural information. In *KDD*.

Raghavendra Chalapathy and Sanjay Chawla. 2019. Deep learning for anomaly detection: A survey. *arXiv preprint arXiv:1901.03407*.

Benjamin Paul Chamberlain, James Clough, and Marc Peter Deisenroth. 2017. Neural embeddings of graphs in hyperbolic space. *arXiv preprint arXiv:1705.10359*.

Ines Chami, Adva Wolf, Da-Cheng Juan, Frederic Sala, Sujith Ravi, and Christopher Ré. 2020. Low-dimensional hyperbolic knowledge graph embeddings. In *ACL*.

C.-C. Chang and C.-J. Lin. 2011. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*.

Michael B Chang, Tomer Ullman, Antonio Torralba, and Joshua B Tenenbaum. 2016. A compositional object-based approach to learning physical dynamics. *arXiv preprint arXiv:1612.00341*.

J. Chen, T. Ma, and C. Xiao. 2018a. Fastgcn: Fast learning with graph convolutional networks via importance sampling. *ICLR*.

J. Chen, J. Zhu, and L. Song. 2018b. Stochastic training of graph convolutional networks with variance reduction. In *ICML*.

Jianbo Chen, Le Song, Martin J Wainwright, and Michael I Jordan. 2018c. Learning to explain: An information-theoretic perspective on model interpretation. *arXiv preprint arXiv:1802.07814*.

Jie Chen, Haw-ren Fang, and Yousef Saad. 2009. Fast approximate kNN graph construction for high dimensional data via recursive lanczos bisection. *JMLR*.

Jie Chen, Tengfei Ma, and Cao Xiao. 2018d. Fastgcn: fast learning with graph convolutional networks via importance sampling. In *ICLR*.

Wei Chen, Wenjie Fang, Guangda Hu, and Michael W Mahoney. 2013. On the hyperbolicity of small-world and treelike random graphs. *Internet Mathematics*, 9(4):434–491.

Z. Chen, L. Li, and J. Bruna. 2019. Supervised community detection with line graph neural networks. In *ICLR*.

Justin Cheng, Cristian Danescu-Niculescu-Mizil, and Jure Leskovec. 2015. Antisocial behavior in online discussion communities. *CoRR*.

E. Cho, S. Myers, and J. Leskovec. 2011. Friendship and mobility: user movement in location-based social networks. In *KDD*.

Aaron Clauset, Cristopher Moore, and Mark EJ Newman. 2008. Hierarchical structure and the prediction of missing links in networks. *Nature*, 453(7191):98.

Gabriele Corso, Luca Cavalleri, Dominique Beaini, Pietro Lio, and Petar Velickovic. 2020. Principal neighbourhood aggregation for graph nets. *arXiv preprint arXiv:2004.05718*.

P. Covington, J. Adams, and E. Sargin. 2016. Deep neural networks for youtube recommendations. In *RecSys*. ACM.

Marco Cuturi. 2013. Sinkhorn distances: Lightspeed computation of optimal transportation distances.

Hanjun Dai, Bo Dai, and Le Song. 2016a. Discriminative embeddings of latent variable models for structured data. In *ICML*.

Hanjun Dai, Yichen Wang, Rakshit Trivedi, and Le Song. 2016b. Deep coevolutionary network: Embedding user and item features for recommendation. *arXiv preprint arXiv:1609.03675*.

A. Debnath et al. 1991. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry*, 34(2):786–797.

M. Defferrard, X. Bresson, and P. Vandergheynst. 2016a. Convolutional neural networks on graphs with fast localized spectral filtering. In *NeurIPS*.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016b. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*.

Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. 2018. Convolutional 2d knowledge graph embeddings. In *AAAI*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*.

I. S. Dhillon, Y. Guan, and B. Kulis. 2007. Weighted graph cuts without eigenvectors a multilevel approach. *PAMI*.

Bhuwan Dhingra, Christopher J Shallue, Mohammad Norouzi, Andrew M Dai, and George E Dahl. 2018. Embedding text in hyperbolic spaces. *NAACL HLT*.

P. D Dobson and A. J. Doig. 2003. Distinguishing enzyme structures from non-enzymes without alignments. *Journal of Molecular Biology*, 330(4):771 – 783.

Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*.

F. Doshi-Velez and B. Kim. 2017. Towards A Rigorous Science of Interpretable Machine Learning. ArXiv: 1702.08608.

D. Duvenaud et al. 2015a. Convolutional networks on graphs for learning molecular fingerprints. In *NeurIPS*.

D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams. 2015b. Convolutional networks on graphs for learning molecular fingerprints. In *NeurIPS*.

David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. 2015c. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*.

C. Eksombatchai, P. Jindal, J. Z. Liu, Y. Liu, R. Sharma, C. Sugnet, M. Ulrich, and J. Leskovec. 2018. Pixie: A system for recommending 3+ billion items to 200+ million users in real-time. *WWW*.

D. Erhan, Y. Bengio, A. Courville, and P. Vincent. 2009. Visualizing higher-layer features of a deep network. *University of Montreal*, 1341(3):1.

A. Feragen, N. Kasenburg, J. Petersen, M. D. Bruijne, and K. M. Borgwardt. 2013. Scalable kernels for graphs with continuous attributes. In *NeurIPS*.

M. Fey, J. E. Lenssen, F. Weichert, and H. Müller. 2018. SplineCNN: Fast geometric deep learning with continuous B-spline kernels. In *CVPR*.

A. Fisher, C. Rudin, and F. Dominici. 2018. All Models are Wrong but many are Useful: Variable Importance for Black-Box, Proprietary, or Misspecified Prediction Models, using Model Class Reliance. ArXiv: 1801.01489.

A. Fout, J. Byrd, B. Shariat, and A. Ben-Hur. 2017. Protein interface prediction using graph convolutional networks. In *NeurIPS*, pages 6533–6542.

Maurice Fréchet. 1948. Les éléments aléatoires de nature quelconque dans un espace distancié. In *Annales de l'institut Henri Poincaré*.

Octavian Ganea, Gary Bécigneul, and Thomas Hofmann. 2018. Hyperbolic neural networks. In *NeurIPS*.

Hongyang Gao and Shuiwang Ji. 2019. Graph u-nets. In *ICML*.

Hongyang Gao, Zhengyang Wang, and Shuiwang Ji. 2018. Large-scale learnable graph convolutional networks. In *KDD*.

J. Gilmer, S. Schoenholz, P. Riley, O. Vinyals, and G. Dahl. 2017. Neural message passing for quantum chemistry. In *ICML*.

Marco Gori, Gabriele Monfardini, and Franco Scarselli. 2005. A new model for learning in graph domains. In *IEEE International Joint Conference on Neural Networks*, volume 2, pages 729–734.

P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*.

Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Bernhard Schölkopf, and Alexander Smola. 2012. A kernel two-sample test. *Journal of Machine Learning Research*, 13(Mar):723–773.

Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *KDD*.

Radek Grzeszczuk, Demetri Terzopoulos, and Geoffrey Hinton. 1998. Neuroanimator: Fast neural network emulation and control of physics-based models. In *Proceedings of the 25th Annual Conference on Computer graphics and Interactive Techniques*, pages 9–20.

Albert Gu, Frederic Sala, Beliz Gunel, and Christopher Ré. 2019. Learning mixed-curvature representations in product spaces. In *ICLR*.

R. Guidotti et al. 2018. A Survey of Methods for Explaining Black Box Models. *ACM Comput. Surv.*, 51(5):93:1–93:42.

Caglar Gulcehre, Misha Denil, Mateusz Malinowski, Ali Razavi, Razvan Pascanu, Karl Moritz Hermann, Peter Battaglia, Victor Bapst, David Raposo, Adam Santoro, et al. 2019. Hyperbolic attention networks. In *ICLR*.

W. Hamilton, Z. Ying, and J. Leskovec. 2017a. Inductive representation learning on large graphs. In *NIPS*.

W. L. Hamilton, R. Ying, and J. Leskovec. 2017b. Representation learning on graphs: Methods and applications. *IEEE Data Engineering Bulletin*, 40(3):52–74.

William L Hamilton, Zhitao Ying, and Jure Leskovec. 2017c. Inductive representation learning on large graphs. In *NeurIPS*.

Jessica B Hamrick, Kelsey R Allen, Victor Bapst, Tina Zhu, Kevin R McKee, Joshua B Tenenbaum, and Peter W Battaglia. 2018. Relational inductive bias for physical construction in humans and machines. *arXiv preprint arXiv:1806.01203*.

Siyu He, Yin Li, Yu Feng, Shirley Ho, Siamak Ravanbakhsh, Wei Chen, and Barnabás Póczos. 2019. Learning to predict the cosmological structure formation. *Proceedings of the National Academy of Sciences*, 116(28):13825–13832.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Philipp Holl, Vladlen Koltun, and Nils Thuerey. 2020. Learning to control PDEs with differentiable physics. *arXiv preprint arXiv:2001.07457*.

G. Hooker. 2004. Discovering additive structure in black box functions. In *KDD*.

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Difftaichi: Differentiable programming for physical simulation. *arXiv preprint arXiv:1910.00935*.

Yuanming Hu, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. 2018. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Trans. Graph.*, 37(4).

W.B. Huang, T. Zhang, Y. Rong, and J. Huang. 2018. Adaptive sampling towards fast graph representation learning. In *NeurIPS*.

S. Ioffe and C. Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*.

W. Jin, C. W. Coley, R. Barzilay, and T. S. Jaakkola. 2017. Predicting organic reaction outcomes with Weisfeiler-Lehman network. In *NeurIPS*.

Edmond Jonckheere, Poonsuk Lohsoonthorn, and Francis Bonahon. 2008. Scaled gromov hyperbolic graphs. *Journal of Graph Theory*.

Bo Kang, Jefrey Lijffijt, and Tijl De Bie. 2019. Explaine: An approach for explaining network embedding-based link predictions. *arXiv:1904.12694*.

S. Kearnes, K. McCloskey, M. Berndl, V. Pande, and P. Riley. 2016. Molecular graph convolutions: moving beyond fingerprints. volume 30.

K. Kersting, N. M. Kriege, C. Morris, P. Mutzel, and M. Neumann. 2016. Benchmark data sets for graph kernels.

Valentin Khrulkov, Leyla Mirvakhabova, Evgeniya Ustinova, Ivan Oseledets, and Victor Lempitsky. 2019. Hyperbolic image embeddings. *arXiv preprint arXiv:1904.02239*.

Diederik Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *ICLR*.

Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. In *NeurIPS*.

T. N. Kipf and M. Welling. 2017. Semi-supervised classification with graph convolutional networks. In *ICML*.

Thomas Kipf, Ethan Fetaya, Kuan-Chieh Wang, Max Welling, and Richard Zemel. 2018. Neural relational inference for interacting systems. In *ICML*.

Thomas N Kipf and Max Welling. 2016a. Semi-supervised classification with graph convolutional networks. In *ICLR*.

Thomas N Kipf and Max Welling. 2016b. Variational graph auto-encoders. In *NeurIPS Workshop on Bayesian Deep Learning*.

Jon M Kleinberg. 1999. Authoritative sources in a hyperlinked environment. *JACM*.

Robert Kleinberg. 2007. Geographic routing using hyperbolic space. In *IEEE International Conference on Computer Communications*.

Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. 2019a. Predict then propagate: Graph neural networks meet personalized pagerank. In *ICLR*.

Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. 2019b. Combining neural networks with personalized pagerank for classification on graphs. In *ICLR*.

Johannes Klicpera, Stefan Weißenberger, and Stephan Günnemann. 2019c. Diffusion improves graph learning. In *NeurIPS*.

P. W. Koh and P. Liang. 2017. Understanding black-box predictions via influence functions. In *ICML*.

N. M. Kriege, P.-L. Giscard, and R. Wilson. 2016. On valid optimal assignment kernels and applications to graph classification. In *NeurIPS*.

Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguná. 2010. Hyperbolic geometry of complex networks. *Physical Review E*.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *NeurIPS*.

Joseph B Kruskal. 1964. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27.

Srijan Kumar, William L Hamilton, Jure Leskovec, and Dan Jurafsky. 2018. Community interaction and conflict on the web. In *WWW*.

Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting dynamic embedding trajectory in temporal interaction networks. In *KDD*. ACM.

L'ubor Ladickỳ, SoHyeon Jeong, Barbara Solenthaler, Marc Pollefeys, and Markus Gross. 2015. Data-driven fluid simulations using regression forests. *ACM Transactions on Graphics (TOG)*, 34(6):1–9.

H. Lakkaraju, E. Kamar, R. Caruana, and J. Leskovec. 2017. Interpretable & Explorable Approximations of Black Box Models. In *KDD (FAT ML Workshop)*.

Joyce van de Leemput, Nathan C Boles, Thomas R Kiehl, Barbara Corneo, Patty Lederman, Vilas Menon, Changkyu Lee, Refugio A Martinez, Boaz P Levi, Carol L Thompson, et al. 2014. Cortecon: a temporal transcriptome analysis of in vitro human cerebral cortex development from human embryonic stem cells. *Neuron*.

T. Lei, W. Jin, R. Barzilay, and T. S. Jaakkola. 2017. Deriving neural architectures from sequence and graph kernels. In *ICML*.

Kingsly Leung and Christopher Leckie. 2005. Unsupervised anomaly detection in network intrusion detection using clusters. In *Australasian conference on Computer Science*.

Qimai Li, Zhichao Han, and Xiao-Ming Wu. 2018a. Deeper insights into graph convolutional networks for semi-supervised learning. In *AAAI*.

Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. 2016. Gated graph sequence neural networks. In *ICLR*.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. In *ICLR*.

Yunzhu Li, Jiajun Wu, Russ Tedrake, Joshua B Tenenbaum, and Antonio Torralba. 2018b. Learning particle dynamics for manipulating rigid bodies, deformable objects, and fluids. *arXiv preprint arXiv:1810.01566*.

Yunzhu Li, Jiajun Wu, Jun-Yan Zhu, Joshua B Tenenbaum, Antonio Torralba, and Russ Tedrake. 2019. Propagation networks for model-based control under partial observation. In *ICRA*. IEEE.

R. Liao, M. Brockschmidt, D. Tarlow, A. L. Gaunt, R. Urtasun, and R. Zemel. 2018. Graph partition neural networks for semi-supervised classification. In *ICLR (Workshop Track)*.

Meng Liu, Hongyang Gao, and Shuiwang Ji. 2020. Towards deeper graph neural networks. In *KDD*.

Qi Liu, Maximilian Nickel, and Douwe Kiela. 2019a. Hyperbolic graph neural networks. In *NeurIPS*.

Ziqi Liu, Chaochao Chen, Longfei Li, Jun Zhou, Xiaolong Li, Le Song, and Yuan Qi. 2019b. Geniepath: Graph neural networks with adaptive receptive paths. In *AAAI*.

S. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *NeurIPS*.

A. Lusci, G. Pollastri, and P. Baldi. 2013. Deep architectures and deep learning in chemoinformatics: The prediction of aqueous solubility for drug-like molecules. *Journal of Chemical Information and Modeling*, 53(7):1563–1575.

Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. 2014. Unified particle physics for real-time applications. *ACM Transactions on Graphics (TOG)*, 33(4):1–12.

Franco Manessi, Alessandro Rozza, and Mario Manzo. 2020. Dynamic graph convolutional networks. *Pattern Recognition*, 97:107000.

C. Merkwirth and T. Lengauer. 2005. Automatic generation of complementary descriptors with molecular graph networks. *Journal of Chemical Information and Modeling*.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *NeurIPS*.

Joe J Monaghan. 1992. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics*, 30(1):543–574.

Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodola, Jan Svoboda, and Michael M Bronstein. 2017. Geometric deep learning on graphs and manifolds using mixture model cnns. In *CVPR*.

Damian Mrowca, Chengxu Zhuang, Elias Wang, Nick Haber, Li F Fei-Fei, Josh Tenenbaum, and Daniel L Yamins. 2018. Flexible neural representation for physics prediction. In *NeurIPS*.

Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. 2007. Position based dynamics. *Journal of Visual Communication and Image Representation*.

Galileo Namata, Ben London, Lise Getoor, Bert Huang, and UMD EDU. 2012. Query-driven active surveying for collective classification.

Onuttom Narayan and Iraj Saniee. 2011. Large-scale curvature of networks. *Physical Review E*.

D. Neil et al. 2018. Interpretable Graph Convolutional Neural Networks for Inference on Noisy Knowledge Graphs. In *ML4H Workshop at NeurIPS*.

Andrew Y Ng, Michael I Jordan, Yair Weiss, et al. 2001. On spectral clustering: Analysis and an algorithm. In *NeurIPS*.

Maximillian Nickel and Douwe Kiela. 2017. Poincaré embeddings for learning hierarchical representations. In *NeurIPS*, pages 6338–6347.

Maximillian Nickel and Douwe Kiela. 2018. Learning continuous hierarchies in the lorentz model of hyperbolic geometry. In *ICML*.

Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. 2016. Learning convolutional neural networks for graphs. In *ICML*.

Caleb C Noble and Diane J Cook. 2003. Graph-based anomaly detection. In *KDD*.

Kenta Oono and Taiji Suzuki. 2020. Graph neural networks exponentially lose expressive power for node classification. In *ICLR*.

A. Van den Oord, S. Dieleman, and B. Schrauwen. 2013. Deep content-based music recommendation. In *NeurIPS*.

OpenMP Architecture Review Board. 2015. OpenMP application program interface version 4.5.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *EMNLP*.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *KDD*.

Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. 2017. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *CVPR*.

Nikhil Rao, Joseph Harrison, Tyler Karrels, Robert Nowak, and Timothy T Rogers. 2010. Using machines to improve human saliency detection. In *ASILOMAR*. IEEE.

Erzsébet Ravasz and Albert-László Barabási. 2003. Hierarchical organization in complex networks. *Physical review E*.

Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *LREC*.

M. Ribeiro, S. Singh, and C. Guestrin. 2016. Why should i trust you?: Explaining the predictions of any classifier. In *KDD*.

Joel W Robbin and Dietmar A Salamon. 2011. Introduction to differential geometry. *ETH, Lecture Notes, preliminary version*.

Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2020a. Temporal graph networks for deep learning on dynamic graphs. In *ICML Workshop on Graph Representation Learning*.

Emanuele Rossi, Fabrizio Frasca, Ben Chamberlain, Davide Eynard, Michael Bronstein, and Federico Monti. 2020b. Sign: Scalable inception graph neural networks. In *ICML*.

Frederic Sala, Chris De Sa, Albert Gu, and Christopher Ré. 2018. Representation tradeoffs for hyperbolic embeddings. In *ICML*.

Alvaro Sanchez-Gonzalez, Victor Bapst, Kyle Cranmer, and Peter Battaglia. 2019. Hamiltonian graph networks with ode integrators. *arXiv preprint arXiv:1909.12790*.

Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. 2018. Graph networks as learnable physics engines for inference and control. *arXiv preprint arXiv:1806.01242*.

Rik Sarkar. 2011. Low distortion delaunay embedding of trees in hyperbolic plane. In *International Symposium on Graph Drawing*.

F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. 2009a. The graph neural network model. *Transactions on Neural Networks*, 20(1):61–80.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009b. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80.

M. Schlichtkrull, T. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling. 2018a. Modeling relational data with graph convolutional networks. In *ESWC*.

M. Schlichtkrull, T. N Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling. 2018b. Modeling relational data with graph convolutional networks. In *Extended Semantic Web Conference*.

G. J. Schmitz, C. Aldrich, and F. S. Gouws. 1999. ANN-DT: an algorithm for extraction of decision trees from artificial neural networks. *IEEE Transactions on Neural Networks*.

Samuel S Schoenholz and Ekin D Cubuk. 2019. Jax, md: End-to-end differentiable, hardware accelerated, molecular dynamics in pure python. *arXiv preprint arXiv:1912.04232*.

K. Schütt, P. J. Kindermans, H. E. Sauceda, S. Chmiela, A. Tkatchenko, and K. R. Müller. 2017. SchNet: A continuous-filter convolutional neural network for modeling quantum interactions. In *NeurIPS*.

Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective classification in network data. *AI magazine*.

Neil Shah, Alex Beutel, Bryan Hooi, Leman Akoglu, Stephan Gunnemann, Disha Makhija, Mohit Kumar, and Christos Faloutsos. 2016. Edgecentric: Anomaly detection in edge-attributed networks. In *ICDMW*.

Chao Shang, Yun Tang, Jing Huang, Jinbo Bi, Xiaodong He, and Bowen Zhou. 2019. End-to-end structure-aware convolutional networks for knowledge base completion. In *AAAI*.

Uday Shankar Shanthamallu, Jayaraman J Thiagarajan, and Andreas Spanias. 2020. A regularized attention mechanism for graph attention networks. In *ICASSP*.

N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt. 2011a. Weisfeiler-Lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561.

N. Shervashidze, S. V. N. Vishwanathan, T. H. Petri, K. Mehlhorn, and K. M. Borgwardt. 2009. Efficient graphlet kernels for large graph comparison. In *International Conference on Artificial Intelligence and Statistics*.

Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. 2011b. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561.

A. Shrikumar, P. Greenside, and A. Kundaje. 2017. Learning Important Features Through Propagating Activation Differences. In *ICML*.

M. Simonovsky and N. Komodakis. 2017. Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *CVPR*.

K. Simonyan and A. Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Aravind Subramanian, Pablo Tamayo, Vamsi K Mootha, Sayan Mukherjee, Benjamin L Ebert, Michael A Gillette, Amanda Paulovich, Scott L Pomeroy, Todd R Golub, Eric S Lander, et al. 2005. Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles. *Proceedings of the National Academy of Sciences*, 102(43):15545–15550.

Deborah Sulsky, Shi-Jian Zhou, and Howard L Schreyer. 1995. Application of a particle-in-cell method to solid mechanics. *Computer physics communications*, 87(1-2):236–252.

Chen Sun, Per Karlsson, Jiajun Wu, Joshua B Tenenbaum, and Kevin Murphy. 2019a. Stochastic prediction of multi-agent interactions from partial observations. *arXiv preprint arXiv:1902.09641*.

Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. 2019b. Rotate: Knowledge graph embedding by relational rotation in complex space. In *ICLR*.

M. Sundararajan, A. Taly, and Q. Yan. 2017. Axiomatic Attribution for Deep Networks. In *ICML*.

Damian Szklarczyk, John H Morris, Helen Cook, Michael Kuhn, Stefan Wyder, Milan Simonovic, Alberto Santos, Nadezhda T Doncheva, Alexander Roth, Peer Bork, et al. 2016. The string database in 2017: quality-controlled protein–protein association networks, made broadly accessible. *Nucleic acids research*.

Andrea Tacchetti, H Francis Song, Pedro AM Mediano, Vinicius Zambaldi, Neil C Rabinowitz, Thore Graepel, Matthew Botvinick, and Peter W Battaglia. 2018. Relational forward models for multi-agent learning. *arXiv preprint arXiv:1809.11044*.

Jian Tang, Jingzhou Liu, Ming Zhang, and Qiaozhu Mei. 2016. Visualizing large-scale and high-dimensional data. In *WWW*.

Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. Line: Large-scale information network embedding. In *WWW*.

Yun Tang, Jing Huang, Guangtao Wang, Xiaodong He, and Bowen Zhou. 2020. Orthogonal relation transforms with graph context modeling for knowledge graph embedding. In *ACL*.

Yi Tay, Luu Anh Tuan, and Siu Cheung Hui. 2018. Hyperbolic representation learning for fast and efficient neural question answering. In *WSDM*.

Alexandru Tifrea, Gary Becigneul, and Octavian-Eugen Ganea. 2019. Poincaré glove: Hyperbolic word embeddings. In *ICLR*.

Kristina Toutanova and Danqi Chen. 2015. Observed versus latent features for knowledge base and text inference. In *CVSC-WS*.

Rakshit Trivedi, Hanjun Dai, Yichen Wang, and Le Song. 2017. Know-evolve: Deep temporal reasoning for dynamic knowledge graphs. In *ICML*.

Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. Dyrep: Learning representations over dynamic graphs. In *ICLR*.

Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. 2016. Complex embeddings for simple link prediction. In *ICML*.

Benjamin Ummenhofer, Lukas Prantl, Nils Thürey, and Vladlen Koltun. 2020. Lagrangian fluid simulation with continuous convolutions. In *ICLR*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NeurIPS*.

P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. 2018. Graph attention networks. In *ICLR*.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2018a. Graph attention networks. In *ICLR*.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2018b. Graph attention networks. In *ICLR*.

Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. 2020. Neural execution of graph algorithms. In *ICLR*.

Saurabh Verma and Zhi-Li Zhang. 2018. Graph capsule convolutional neural networks. *arXiv preprint arXiv:1805.08090*.

Cédric Villani. 2003. *Topics in optimal transportation*. American Mathematical Soc.

Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. 2015. Order matters: Sequence to sequence for sets. *arXiv:1511.06391*.

Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. 2013. Regularization of neural networks using dropconnect. In *ICML*.

Daixin Wang, Peng Cui, and Wenwu Zhu. 2016. Structural deep network embedding. In *KDD*.

Guangtao Wang, Rex Ying, Jing Huang, and Jure Leskovec. 2019a. Improving graph attention networks with large margin-based constraints. In *NeurIPS Workshop*.

Quan Wang, Pingping Huang, Haifeng Wang, Songtai Dai, Wenbin Jiang, et al. 2019b. Coke: Contextualized knowledge graph embedding. *arXiv preprint arXiv:1911.02168*.

Xiao Wang, Peng Cui, Jing Wang, Jian Pei, Wenwu Zhu, and Shiqiang Yang. 2017. Community preserving network embedding. In *AAAI*.

Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. 2021. Inductive representation learning in temporal networks via casual anonymous walk. In *ICLR*.

Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *Nature*, 393(6684):440–442.

etc. Weihua Hu, Matthias Fey. 2020. Open graph benchmark: Datasets for machine learning on graphs. In *NeurIPS*.

Steffen Wiewel, Moritz Becher, and Nils Thuerey. 2019. Latent space physics: Towards learning the temporal evolution of fluid flow. In *Computer Graphics Forum*, pages 71–82. Wiley Online Library.

David H Wolpert. 1992. Stacked generalization. *Neural Networks*.

Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. 2019. Simplifying graph convolutional networks. In *ICML*, pages 6861–6871.

Liwei Wu, Hsiang-Fu Yu, Nikhil Rao, James Sharpnack, and Cho-Jui Hsieh. 2020. Graph dna: Deep neighborhood aware graph encoding for collaborative filtering. In *AISTATS*.

T. Xie and J. Grossman. 2018. Crystal graph convolutional neural networks for an accurate and interpretable prediction of material properties. In *Phys. Rev. Lett.*

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. 2020. On layer normalization in the transformer architecture. In *ICML*.

Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive representation learning on temporal graphs. In *ICLR*.

K. Xu, W. Hu, J. Leskovec, and S. Jegelka. 2019. How powerful are graph neural networks? In *ICRL*.

K. Xu, C. Li, Y. Tian, T. Sonobe, K. Kawarabayashi, and S. Jegelka. 2018a. Representation learning on graphs with jumping knowledge networks. In *ICML*.

Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018b. Representation learning on graphs with jumping knowledge networks. In *ICML*.

Linchuan Xu, Xiaokai Wei, Jiannong Cao, and Philip S Yu. 2017. Embedding identity and interest for social networks. In *WWW*.

Sijie Yan, Yuanjun Xiong, and Dahua Lin. 2018. Spatial temporal graph convolutional networks for skeleton-based action recognition. In *AAAI*.

P. Yanardag and S. V. N. Vishwanathan. 2015a. A structural smoothing framework for robust graph comparison. In *NeurIPS*.

Pinar Yanardag and SVN Vishwanathan. 2015b. Deep graph kernels. In *KDD*.

Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. 2015. Embedding entities and relations for learning and inference in knowledge bases. In *ICLR*.

Zhilin Yang, William Cohen, and Ruslan Salakhutdinov. 2016. Revisiting semi-supervised learning with graph embeddings. In *ICML*.

C. Yeh, J. Kim, I. Yen, and P. Ravikumar. 2018. Representer point selection for explaining deep neural networks. In *NeurIPS*.

R. Ying, R. He, K. Chen, P. Eksombatchai, W. Hamilton, and J. Leskovec. 2018a. Graph convolutional neural networks for web-scale recommender systems. In *KDD*.

Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018b. Graph convolutional neural networks for web-scale recommender systems. In *KDD*.

Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. 2018c. Hierarchical graph representation learning with differentiable pooling. In *NeurIPS*.

Minji Yoon, Bryan Hooi, Kijung Shin, and Christos Faloutsos. 2019. Fast and accurate anomaly detection in dynamic graphs with a two-pronged approach. In *KDD*.

Jiaxuan You, Bowen Liu, Zhitao Ying, Vijay Pande, and Jure Leskovec. 2018a. Graph convolutional policy network for goal-directed molecular graph generation. In *NeurIPS*.

Jiaxuan You, Rex Ying, and Jure Leskovec. 2019. Position-aware graph neural networks. In *ICML*.

Jiaxuan You, Rex Ying, Xiang Ren, William L Hamilton, and Jure Leskovec. 2018b. Graphrnn: Generating realistic graphs with deep auto-regressive models. In *ICML*.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. 2017. Deep sets. In *NeurIPS*.

M. Zeiler and R. Fergus. 2014. Visualizing and Understanding Convolutional Networks. In *ECCV*.

Hongyi Zhang, Sashank J Reddi, and Suvrit Sra. 2016. Riemannian svrg: Fast stochastic optimization on riemannian manifolds. In *NeurIPS*.

Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. 2018a. Gaan: Gated attention networks for learning on large and spatiotemporal graphs. In *UAI*.

M. Zhang and Y. Chen. 2018. Link prediction based on graph neural networks. In *NeurIPS*.

M. Zhang, Z. Cui, M. Neumann, and Y. Chen. 2018b. An end-to-end deep learning architecture for graph classification. In *AAAI*.

Shuai Zhang, Yi Tay, Lina Yao, and Qi Liu. 2019. Quaternion knowledge graph embedding. In *NeurIPS*.

Yongqi Zhang, Quanming Yao, Wenyuan Dai, and Lei Chen. 2020. Autosf: Searching scoring functions for knowledge graph embedding. In *ICDE*.

Z. Zhang, Peng C., and W. Zhu. 2018c. Deep Learning on Graphs: A Survey. *arXiv:1812.04202*.

Li Zheng, Zhenpeng Li, Jian Li, Zhao Li, and Jun Gao. 2019. Addgraph: Anomaly detection in dynamic graph using attention-based temporal gcn. In *IJCAI*.

J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. 2018. Graph Neural Networks: A Review of Methods and Applications. *arXiv:1812.08434*.

Chenyi Zhuang and Qiang Ma. 2018. Dual graph convolutional networks for graph-based semi-supervised classification. In *WWW*.

J. Zilke, E. Loza Mencia, and F. Janssen. 2016. DeepRED - Rule Extraction from Deep Neural Networks. In *Discovery Science*. Springer International Publishing.

L. Zintgraf, T. Cohen, T. Adel, and M. Welling. 2017. Visualizing deep neural network decisions: Prediction difference analysis. In *ICLR*.

M. Zitnik, M. Agrawal, and J. Leskovec. 2018a. Modeling polypharmacy side effects with graph convolutional networks. *Bioinformatics*, 34.

M. Zitnik, M. Agrawal, and J. Leskovec. 2018b. Modeling polypharmacy side effects with graph convolutional networks. *Bioinformatics*.

Marinka Zitnik, Marcus W Feldman, Jure Leskovec, et al. 2019. Evolution of resilience in protein interactomes across the tree of life. *Proceedings of the National Academy of Sciences*, 116(10):4426–4433.

Marinka Zitnik and Jure Leskovec. 2017. Predicting multicellular function through multi-layer tissue networks. *Bioinformatics*.