Graph Deep Learning: Methods and Applications

Muhan Zhang
Washington University in St. Louis

Follow this and additional works at: https://openscholarship.wustl.edu/eng_etds

Part of the Artificial Intelligence and Robotics Commons

Recommended Citation
https://openscholarship.wustl.edu/eng_etds/504

This Dissertation is brought to you for free and open access by the McKelvey School of Engineering at Washington University Open Scholarship. It has been accepted for inclusion in Engineering and Applied Science Theses & Dissertations by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.
Graph Deep Learning: Methods and Applications

by

Muhan Zhang

A dissertation presented to
The Graduate School
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

December 2019
St. Louis, Missouri
# Table of Contents

List of Figures ........................................................................................................... v  
List of Tables ........................................................................................................... viii  
Acknowledgments ..................................................................................................... x  
Abstract .................................................................................................................. xiii  

**Chapter 1: Introduction** ......................................................................................... 1  
  1.1 Graph Deep Learning .................................................................................... 1  
  1.2 A Brief History of Graph Neural Networks .................................................. 4  
  1.3 Graph Neural Networks Basics .................................................................. 6  
  1.4 A Categorization of Graph Neural Networks ............................................... 8  
    1.4.1 GNNs for node-level tasks ................................................................. 9  
    1.4.2 GNNs for graph-level tasks .............................................................. 18  
    1.4.3 GNNs for edge-level tasks .............................................................. 19  

**Chapter 2: Graph Neural Networks for Graph Representation Learning** .... 22  
  2.1 Graph Neural Networks for Graph Classification ......................................... 23  
    2.1.1 Traditional graph classification methods: graph kernels ...................... 23  
    2.1.2 Limitations of existing GNNs for graph classification ......................... 25  
    2.1.3 Deep Graph Convolutional Neural Network (DGCNN) ...................... 26  
    2.1.4 Training through backpropagation .................................................... 30  
    2.1.5 Discussion ...................................................................................... 31  
    2.1.6 Experimental results ........................................................................ 34  
    2.1.7 Conclusion ..................................................................................... 41  
  2.2 Graph Neural Networks for Medical Ontology Embedding ....................... 42  
    2.2.1 Introduction .................................................................................... 42
4.4 Experiments

4.4.1 Reconstruction accuracy, prior validity, uniqueness and novelty ........ 121
4.4.2 Predictive performance of latent representation.............................. 122
4.4.3 Bayesian optimization ................................................................... 123
4.4.4 Latent space visualization ............................................................ 126

4.5 Conclusion ....................................................................................... 128

Chapter 5: Conclusions ........................................................................ 129

References ............................................................................................. 133

Appendix A: ........................................................................................... [147]

A.1 Additional details about link prediction baselines ............................ [147]
A.2 More related work on neural architecture search and Bayesian network structure learning .................................................................... [149]
A.3 More details about neural architecture search ................................. [151]
A.4 More details about Bayesian network structure learning .............. [153]
A.5 Baselines for D-VAE .................................................................... [154]
A.6 VAE training details .................................................................... [157]
A.7 More details of the prior validity experiment .................................. [158]
A.8 SGP training details .................................................................... [159]
A.9 The generated neural architectures ............................................... [160]
A.10 The generated Bayesian networks ............................................... [161]
List of Figures

Figure 1.1: 2D convolution (left) vs. graph convolution (right). Graph convolution can be seen as generalizing 2D convolution on grids to arbitrary structures, where a node’s local receptive field is no longer a fixed-size subgrid, but is defined to be its one-hop neighboring nodes. Figure is from [166]. ................................................................. 6

Figure 2.1: A consistent input ordering is crucial for CNNs’ successes on graph classification. If we randomly shuffle the pixels of the left image, then state-of-the-art convolutional neural networks (CNNs) will fail to recognize it as an eagle.............................................................. 27

Figure 2.2: The overall structure of DGCNN. An input graph is first passed through multiple message passing layers where node information is propagated between neighbors. Then the vertex features are sorted and pooled with a SortPooling layer, and passed to 1-D convolutional layers to learn a predictive model............................................................ 29

Figure 2.3: Training curves of SortPooling compared with 1) random order, 2) precomputed WL order, 3) summed node features, and 4) fixed weights. 40

Figure 2.4: Comparison between Gram and HAP. Gram only considers a node’s unordered ancestor set to compute its embedding. HAP hierarchically propagate information across the graph. In the bottom-up round, each parent aggregates information from its children. In the top-down round, each child aggregates information from its parents. The final embedding of each node effectively absorbs information from not only its ancestors, but the entire graph (ancestors, descendants, siblings and others). ................................................................. 43

Figure 2.5: t-SNE scatterplots of diagnosis code embeddings learned by HAP...... 59

Figure 2.6: t-SNE scatterplots of diagnosis code embeddings learned by Gram (left) and RNN (right). ................................................................. 60
Figure 3.1: The SEAL framework. For each target link, SEAL extracts a local enclosing subgraph around it, and uses a GNN to learn general graph structure features for link prediction. Note that the heuristics listed inside the box are just for illustration – the learned features may be completely different from existing heuristics.

Figure 3.2: An illustration of our IGMC framework. We extract a local enclosing subgraph around each training rating, and use a GNN to learn graph patterns that are useful for rating prediction. Note that the features listed inside the box are only for illustration – the learned features can be much more complex. We use the trained GNN to complete other missing entries of the matrix.

Figure 4.1: Computations can be represented by DAGs. Note that the left and right DAGs represent the same computation.

Figure 4.2: An illustration of the encoding procedure for a neural architecture. Following a topological ordering, we iteratively compute the hidden state for each node (red) by feeding in its predecessors’ hidden states (blue). This simulates how an input signal goes through the DAG, with \( \mathbf{h}_v \) simulating the output signal at node \( v \).

Figure 4.3: An illustration of the steps for generating a new node.

Figure 4.4: An example Bayesian network and its encoding.

Figure 4.5: Top 5 neural architectures found by each model and their true test accuracies.

Figure 4.6: Top 5 Bayesian networks found by each model and their BIC scores (higher the better).

Figure 4.7: Comparing BO with random search on neural architectures. Left: average weight-sharing accuracy of the selected points in each iteration. Right: highest weight-sharing accuracy of the selected points over time.

Figure 4.8: Comparing BO with random search on Bayesian networks. Left: average BIC score of the selected points in each iteration. Right: highest BIC score of the selected points over time.

Figure 4.9: Great circle interpolation starting from a point and returning to itself. Upper: D-VAE. Lower: S-VAE.

Figure 4.10: Visualizing a principal 2-D subspace of the latent space.

Figure A.1: Two bits of change in the string representations can completely change the computational purpose.
Figure A.2: 2-D visualization of decoded neural architectures. Left: D-VAE. Right: S-VAE.

Figure A.3: 2-D visualization of decoded Bayesian networks. Left: D-VAE. Right: S-VAE.
List of Tables

Table 2.1: DGCNN’s comparison with graph kernels. .................................................. 36
Table 2.2: Comparison with other deep learning approaches. ............................ 38
Table 2.3: Accuracy results on MUTAG for the supplementary experiments. ...... 41
Table 2.4: Statistics of ACTFAST and MIMIC-III datasets. .............................. 55
Table 2.5: Grouped and overall Accuracy@5 of sequential procedure prediction on ACTFAST data. ................................................................. 57
Table 2.6: Grouped and overall Accuracy@20 of sequential diagnosis prediction on MIMIC-III data. ............................................................. 58

Table 3.1: Popular heuristics for link prediction, see [96] for details. ............... 65
Table 3.2: Comparing SEAL with heuristic methods (AUC). .......................... 80
Table 3.3: Comparing SEAL with latent feature methods (AUC). ................... 80
Table 3.4: Inference time of SEAL ................................................................. 81
Table 3.5: Comparing SEAL with network embedding methods (AUC) on larger networks, OOM: out of memory). ........................................... 82
Table 3.6: Statistics of each dataset .................................................................... 94
Table 3.7: RMSE test results on Flixster, Douban and YahooMusic. .............. 95
Table 3.8: RMSE test results on MovieLens-100K. ........................................ 96
Table 3.9: RMSE test results on MovieLens-1M. ............................................ 97
Table 3.10: RMSE of transferring models trained on ML-100K to Flixster, Douban and YahooMusic. .......................................................... 97

Table 4.1: Reconstruction accuracy, prior validity, uniqueness and novelty (%)... 122
Table 4.2: Predictive performance of encoded means...................................... 123
Table A.1: Comparison of different link prediction methods [149]
Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Yixin Chen, for his guidance, advice and support throughout my PhD study. Thanks to his broad research interest, I got in touch with various interesting problems including link prediction, recommender systems, graph classification, graph neural networks, metabolic networks, time series prediction and so on. From these topics I gradually built up my research interest on graphs, and finally decided to give this dissertation on graph deep learning. I am grateful for all the collaboration opportunities with other departments that he gave to me too. Through interdisciplinary research, I got to know many fascinating real-world applications of machine learning and data mining, which greatly broadened my interest and horizon. I also enjoy talking with him a lot for his wisdom and sense of humor, and always get valuable inspirations and innovative ideas from him. The research skills I learned from him will be invaluable to me in my future career.

I would also like to thank the rest of my committee, Professors Michael Avidan, Sanmay Das, Roman Garnett, Brendan Juba, and Yinjie Tang, for their helpful suggestions to the dissertation. Furthermore, I would like to thank Prof. Avidan for hosting our ACTFAST meetings and pushing the project forward with his great enthusiasm for AI and healthcare. I would like to thank Prof. Das, Prof. Garnett and Prof. Juba for teaching those wonderful courses on artificial intelligence, machine learning and multi-agent systems which I really

x
enjoyed. I am also grateful for the discussions, help and encouragement they provided for my research. I would like to thank Prof. Yinjie Tang for his continuous trust on me, including inviting me to give talks to his class and collaborating on metabolic networks etc.

I would like to thank Anand Bhaskar, Wenlin Chen, Aude Hofleitner, Cheng Ju, Joy Zhang and Zhe Feng for their help during my internship at Facebook. I really enjoyed the time being together with them. Working with them lets me know what excellent researchers do in industry. I would like to thank my collaborators: Zhicheng Cui, Shali Jiang, Christopher R. King, Marion Neumann, Tolutola Oyetunde, for the inspiring discussions, fantastic collaborations, and great teamwork. I would like to thank my friends at WashU: Haipeng Dai, Zehao Dong, Yujie He, Zhiyang Huang, Zhuoshu Li, Haoran Li, Xiaoxin Liu, Chen Liu, Junjie Liu, Yehan Ma, Gustavo Malkomes, Wei Tang, Zhihao Xia, Hao Yan, Hang Yan, Yajie Yan, Jinghan Yang, Mingquan Yuan, Zhen Zhang, Kai Zhou, Liang Zhou, Maede Zolanvari and all others who are not listed here for all the great time and fun we had together.

I would like to thank my father Hongliang Zhang and mother Yanling Sun for their eternal love. They cultivated my interest in computer since I was a child. I still remember how I was attracted by the first computer bought by my father when I was three.

Finally, I would like to thank my beloved wife, Liran Wang, who, regardless of successes or failures, has always accompanied and trusted me. With you, I am not afraid of even the deepest dark, because you are my light.

Muhan Zhang

Washington University in Saint Louis

December 2019
Dedicated to my wife, Liran, and my parents.
ABSTRACT OF THE DISSERTATION

Graph Deep Learning: Methods and Applications

by

Muhan Zhang

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2019

Professor Yixin Chen, Chair

The past few years have seen the growing prevalence of deep neural networks on various application domains including image processing, computer vision, speech recognition, machine translation, self-driving cars, game playing, social networks, bioinformatics, and healthcare etc. Due to the broad applications and strong performance, deep learning, a subfield of machine learning and artificial intelligence, is changing everyone’s life.

Graph learning has been another hot field among the machine learning and data mining communities, which learns knowledge from graph-structured data. Examples of graph learning range from social network analysis such as community detection and link prediction, to relational machine learning such as knowledge graph completion and recommender systems, to mutli-graph tasks such as graph classification and graph generation etc.

An emerging new field, graph deep learning, aims at applying deep learning to graphs. To deal with graph-structured data, graph neural networks (GNNs) are invented in recent years which directly take graphs as input and output graph/node representations. Although GNNs have shown superior performance than traditional methods in tasks such as semi-supervised node classification, there still exist a wide range of other important graph learning problems
where either GNNs’ applicabilities have not been explored or GNNs only have less satisfying performance.

In this dissertation, we dive deeper into the field of graph deep learning. By developing new algorithms, architectures and theories, we push graph neural networks’ boundaries to a much wider range of graph learning problems. The problems we have explored include: 1) graph classification; 2) medical ontology embedding; 3) link prediction; 4) recommender systems; 5) graph generation; and 6) graph structure optimization.

We first focus on two graph representation learning problems: graph classification and medical ontology embedding. For graph classification, we develop a novel deep GNN architecture which aggregates node features through a novel SortPooling layer that replaces the simple summing used in previous works. We demonstrate its state-of-the-art graph classification performance on benchmark datasets. For medical ontology embedding, we propose a novel hierarchical attention propagation model, which uses attention mechanism to learn embeddings of medical concepts from hierarchically-structured medical ontologies such as ICD-9 and CCS. We validate the learned embeddings on sequential procedure/diagnosis prediction tasks with real patient data.

Then we investigate GNNs’ potential for predicting relations, specifically link prediction and recommender systems. For link prediction, we first develop a theory unifying various traditional link prediction heuristics, and then design a framework to automatically learn suitable heuristics from a given network based on GNNs. Our model shows unprecedented strong link prediction performance, significantly outperforming all traditional methods. For recommender systems, we propose a novel graph-based matrix completion model, which uses a GNN to learn graph structure features from the bipartite graph formed by user and item interactions. Our model not only outperforms various matrix completion baselines, but
also demonstrates excellent transfer learning ability – a model trained on MovieLens can be
directly used to predict Douban movie ratings with high performance.

Finally, we explore GNNs’ applicability to graph generation and graph structure optimization.
We focus on a specific type of graphs which usually carry computations on them, namely
directed acyclic graphs (DAGs). We develop a variational autoencoder (VAE) for DAGs and
prove that it can injectively map computations into a latent space. This injectivity allows
us to perform optimization in the continuous latent space instead of the original discrete
structure space. We then apply our VAE to two types of DAGs, neural network architectures
and Bayesian networks. Experiments show that our model not only generates novel and
valid DAGs, but also finds high-quality neural architectures and Bayesian networks through
performing Bayesian optimization in its latent space.
Chapter 1

Introduction

1.1 Graph Deep Learning

Deep learning is changing everyone’s life. One important reason for the superior performance of deep learning over traditional algorithms is that deep learning integrates feature extraction into the model learning itself, i.e., raw input signals (such as image pixels or audio waveforms) are directly fed into the model without performing feature engineering beforehand. Such an end-to-end procedure greatly improves the quality of the extracted features.

Conventional neural network architectures, such as feed-forward neural networks (FFNN), convolutional neural networks (CNNs) and recurrent neural networks (RNNs), require input signals to be represented in fixed-size tensor forms, where each element of a tensor corresponds to a fixed raw input dimension. This way, neural network layers are able to hierarchically extract features and learn patterns from the data. Although achieving great successes on various data types, these conventional neural networks cannot directly be applied to graphs.
Unlike images, graph-structured data do not have a tensor representation that can be readily read by conventional neural networks, which has limited deep learning’s use cases for graphs.

Graph-structured data are abundant in the real world, e.g., social networks, citation networks, biological networks, molecular structures, power grids, knowledge graphs, etc. Furthermore, graph is also an important subject in machine learning, since many machine learning models, such as neural networks and Bayesian networks, are realized as computations on graphs. There exist a wide range of learning problems related to graphs, such as semi-supervised node classification, graph classification, link prediction, community detection, graph clustering, graph generation, network embedding, etc. Due to the abundance of graph data and graph learning problems, it is very important to study how to learn from graphs.

Graph learning is a challenging problem. Firstly, the number of nodes in a graph can be variable, which poses a great challenge for traditional machine learning models that can only take fixed-size input. Secondly, graphs have the isomorphism problem, meaning that the same graph can have factorially many different expressions by simply permuting the nodes, which brings additional challenges to distinguishing graphs. Thirdly, the graph topology contains rich information important for the learning tasks, yet is extremely hard to extract and learn. All these difficulties make graph learning special and different from traditional learning tasks in regular domains.

Because of the special characteristics of graphs, traditional graph learning methods typically rely on predefined structural features such as node degrees, paths, walks, subtrees, frequent subgraphs, etc., and then apply standard machine learning algorithms on the extracted features. Such a two-step procedure separates feature extraction from model learning, which is against deep learning’s end-to-end training principle, thus often having less expressive power. Another way is to use graph kernels [15, 59, 85, 116, 142, 159], which compute
some positive semidefinite graph similarity measures so that kernel machines such as SVM become feasible for some graph learning tasks. However, graph kernels introduce several new problems. Firstly, computing and storing the kernel matrices require at least quadratic time and space complexity w.r.t. the number of graphs, which is often infeasible for large-scale problems in practice. Secondly, the design of graph kernels is often by heuristics. There is no principled way to measure graph similarities, introducing the need to carefully design different graph kernels for different datasets. Thirdly, graph kernels usually lack the ability to learn representations of graphs, which limits their use cases to only a small range of problems such as graph classification.

To better learn from graphs, graph deep learning aims at leveraging the superior feature learning ability of deep learning for graphs. Since conventional neural networks such as CNNs and RNNs do not work, graph neural networks (GNNs), a new type of neural networks designed particularly for graphs, have recently been proposed [7, 18, 39, 43, 77, 94, 121, 136]. GNNs iteratively pass messages between each node and its neighbors in order to extract local substructure features around nodes. Then, an aggregation operation such as summing is applied to all nodes to get a graph-level feature vector. GNNs are parametric models, thus avoiding the need to compute and store kernel matrices. The learnable parameters in the message passing and aggregation layers equip GNNs with excellent graph representation learning abilities and great flexibility for different graphs. GNNs also enable end-to-end training. Because of these advantages, GNNs gain great popularity in a short time, achieving state-of-the-art performance on semi-supervised node classification [77], network embedding [57], etc.

Despite the success of GNNs in certain problems, as a new tool, GNNs still either do not have satisfying performance or do not find applicabilities in many other important graph learning problems, mainly due to the immature architectures and the shallow understandings people
have for GNNs. In this dissertation, with a series of innovations in algorithms, architectures and theories, we explore GNNs’ potential and limits in three general fields, namely graph representation learning, relation prediction, and graph structure optimization. We first focus on graph representation learning (Chapter 2), the goal of which is to learn representations for graphs or nodes within a graph. Leveraging innovative GNN architectures and designs, we achieve state-of-the-art results for two graph representation learning tasks, namely graph classification and medical ontology embedding. The second field we explore is using GNNs to predict relations (Chapter 3), e.g., predicting links in social networks and recommending items to users. We show that by extracting local enclosing subgraphs around relations, we are able to automatically learn general graph structure features useful for relation prediction based on GNNs instead of using predefined heuristics. The last field we explore is using GNNs to generate and optimize graph structures (Chapter 4). For this problem, we train a GNN-based variational autoencoder (VAE) for directed acyclic graphs (DAGs), and optimize their structures in the VAE’s latent space based on Bayesian optimization. Our model not only generates valid and novel DAG structures, but also provides promising directions to two important DAG structure optimization problems: neural architecture search (NAS) and Bayesian network structure learning (BNSL).

In the remaining part of this chapter, we review the development history of GNNs and introduce the basics of GNNs, which serve as preliminaries for the following chapters of the dissertation.

1.2 A Brief History of Graph Neural Networks

The earliest graph neural networks can date back to Gori et al. (2005) [55] and Scarselli et al. (2009) [136]. These early attempts use recurrent architectures to learn a target node’s
representation by iteratively propagating neighbor information until reaching a stable fixed point, which is computationally very expensive. Recently, encouraged by the success of CNNs in computer vision, a great number of approaches have been developed in parallel that generalize the notion of convolution from images to graph data, namely graph convolutions.

Based on which domain the convolution operation is performed in, these graph convolution approaches can be categorized into spectral-based approaches and spatial-based approaches. The first remarkable spectral-based method was developed by Bruna et al. (2013) [18], which developed a graph convolution operation based on the spectral graph theory, where learnable filters are applied to a graph’s frequency modes computed by graph Fourier transform [135]. From then on, many approximations of spectral-based graph convolution are proposed [39, 60, 77], which either greatly reduce the computation complexity or make the convolution filters localized. For example, Defferrard et al. [39] parameterize the spectral filters as Chebyshev polynomials of eigenvalues, which achieves both efficient and localized filters. One limitation of the above spectral formulations is that they rely on the fixed spectrum of the graph Laplacian, thus are suitable only for graphs with a single structure (but varying signals on vertices).

Spatial-based graph convolutions, on the contrary, are not restricted to a fixed graph structure. To extract local features, several works independently propose to propagate messages between neighboring vertices, inheriting the ideas from early GNNs [136]. Duvenaud et al. [43] propose differentiable Neural Graph Fingerprints, which propagate features between 1-hop neighbors to simulate the traditional circular fingerprint for molecules. Atwood et al. [7] propose Diffusion-CNN, which propagates neighbors with different hops to the center using different weights. Later, Kipf and Welling (2016) [77] develop a first-order approximation for the spectral convolution of [39] which also simplifies to propagation between neighboring vertices. Niepert et al. [121] propose another way of spatial graph convolution by extracting
fixed-sized local patches from nodes’ neighborhoods and linearizing these patches with graph labeling methods and graph canonization tools. Figure 1.1 illustrates the similarity and difference between 2D image convolution and spatial graph convolution. Compared to spectral approaches, spatial graph convolutions are more flexible, easier to implement, and largely reduce the computation complexity. Therefore, they have become the mainstream graph convolution approaches used in graph neural networks. In the remaining part of the dissertation, without special notations, graph neural networks all refer to those using spatial approaches.

1.3 Graph Neural Networks Basics

Graph neural networks (GNNs) have many independently developed formulations [18, 35, 39, 43, 94, 136], most of which can be unified into a message passing framework [53]. Given an undirected graph $G$ with node features $x_v$ (a row vector), the forward pass of a GNN typically contains two phases: a message passing phase (a.k.a. graph convolution) used to extract
local substructure features around nodes, and an aggregation phase (a.k.a. readout, graph pooling, etc.) used to summarize individual node features into a graph-level feature vector. We will consistently use this message passing form to describe our graph neural networks, but will discuss graph neural networks’ other formulations in the next section. We give a brief introduction to the basics of message passing graph neural networks here for readers who want to skip the next section.

The message passing (graph convolution) phase runs for $T$ iterations and involves message functions $M_t$ and vertex update functions $U_t$. At each message passing step, vertex hidden states $z^t_v$ are updated based on messages $m^t_v$ according to:

$$m^{t+1}_v = \sum_{u \in \Gamma(v)} M_t(z^t_v, z^t_u),$$  \hspace{1cm} (1.1)$$

$$z^{t+1}_v = U_t(z^t_v, m^{t+1}_v),$$  \hspace{1cm} (1.2)$$

where $\Gamma(v)$ denotes the set of neighbors of $v$ in graph $G$, $M_t$ and $U_t$ are both differentiable functions with learnable parameters. We omit edge features for simplicity, as edge features are often unavailable. For $z^0_v$, we can let them be the initial node features $x_v$.

The message passing form above has many explanations such as first-order approximation of graph Fourier Transform [39, 77], differentiable approximation of neural graph fingerprints [43], CNN’s generalization from regular grids to graphs [121], etc. However, it also has a straightforward explanation as follows: each step of message passing propagates each node’s 1-hop neighbors’ information to itself, thus summarizing the local substructure patterns around individual nodes; multiple steps of such propagations summarize multi-hop neighborhood information around nodes.
The message passing can be seen as a convolutional operator applied on nodes to extract local features, similar to what a convolutional layer in traditional CNNs does for each pixel. Most existing graph neural networks can be incorporated into this message passing framework. The differences lie in the unique designs of $M_t$ and $U_t$ in different works. For example, $M_t$ can be as simple as a sum/mean [43] or concatenation [121], or use advanced neural architectures such as the attention mechanism [158] and RNNs [57]. The update function $U_t$ can also range from a single linear layer to multi-layer perceptrons (MLPs) [179] and GRUs [94, 185], etc.

With the extracted hidden states of nodes, we can use them for node-level tasks such as semi-supervised node classification and node embedding. However, we still need to get a graph feature vector for doing graph-level tasks such as graph classification and graph generation. The aggregation phase does this job by:

$$z_G = R(\{z_v^T \mid v \in G\}),$$

(1.3)

where $R$ is a readout (pooling) function that is invariant to permutations of nodes in order for GNNs to be invariant to graph isomorphism. Most previous GNN formulations use a simple summing/averaging over the final node states [7, 35, 43, 94]. We will discuss the disadvantages of using such an averaging operation in the second chapter.

### 1.4 A Categorization of Graph Neural Networks

In this section, we give a categorization of graph neural networks with respect to the problems they are addressing, and put our own work into the literature. In each categorization, we introduce the most representative works, including their motivations, formulations, and advantages/disadvantages, etc.
There are mainly three types of problems that all graph neural networks are addressing: node-level tasks, graph-level tasks, and edge-level tasks. Node-level tasks include all node-related graph learning problems, such as semi-supervised node classification [77], network embedding [57], node clustering [162], etc. In graph-level tasks, representations of the entire graphs are learned to allow graph classification [173, 184], graph regression, graph generation [185], etc. In edge-level tasks, representations of edges are learned to enable link prediction [78, 181] or learning recommender systems [berg2017graph, 113, 180], etc., where the task is to predict existence of edges or values of edges.

1.4.1 GNNs for node-level tasks

In node-level tasks, there is typically only one large graph (network) given, and the task is to learn representations for individual nodes of the graph so that downstream node-level tasks can be performed. The node representations can be either learned in a semi-supervised way (i.e., train on some given node labels in an end-to-end way), or in an unsupervised way (i.e., node labels are unknown, and the training is performed by minimizing some auxiliary loss such as reconstructing the graph). Currently, node-level tasks are still the most popular tasks for graph neural networks due to their wide applicabilities in network analysis.

The first graph neural network for learning node states of general graphs is the Graph Neural Network\(^1\) model, which uses a contractive mapping to recurrently find steady states of nodes [136]. However, to ensure convergence, the parameters of the recurrent function has to be constrained via a penalty term on the Jacobian matrix. The repeated use of contractive mapping to find the steady states also poses efficiency problems to practical applications.

\(^1\)It is also where the name GNN comes from. Note that, however, people now use GNN to refer to any neural network that deals with graphs.
Gated Graph Neural Network (GGNN) [94] uses a Gated Recurrent Unit (GRU) [26] as the recurrent function and also no longer requires finding steady states of nodes. Instead, the recurrence is reduced to a fixed number of steps. A node hidden state is updated with

\[ z^{t+1}_v = \text{GRU}(z^t_v, \sum_{u \in \Gamma(v)} Wz^t_u). \]

(1.4)

GGNN uses the same GRU parameters across different steps (layers), which is different from most GNNs that use different weight parameters across different layers. The advantage is that the recurrent unit, GRU, has the ability to learn to forget and keep information from previous steps, which could benefit long-range node propagations in graphs with large diameters. GGNN is designed for undirected graphs, where the message passing is performed simulateneously for all nodes for multiple rounds. In Chapter 4, we will introduce our work on training a variational autoencoder for directed acyclic graphs (DAGs), where we also leverage a GRU as the node update function yet only need to propagate once for each node following a topological order of nodes.

Instead of using the same recurrent unit for different layers, the majority of graph neural networks use convolutional operators with different parameters for different layers, which is inspired by convolutional neural networks (CNNs) for images. The earliest convolutional neural networks for graphs define convolutions in the **spectral domain**. Based on the theories on graph signal processing [135, 144], the spectrum of a graph is given by the eigenvalues of the normalized Laplacian matrix of the graph:

\[ L = U \Lambda U^\top, \]

(1.5)
where $\Lambda$ is the diagonal matrix of the eigenvalues, $U$ is the matrix of the corresponding eigenvectors, and $L = I - \frac{1}{2}AD\frac{1}{2}$ is the normalized Laplacian matrix with $A \in \{0, 1\}^{n \times n}$ denoting the adjacency matrix and $D$ denoting the diagonal degree matrix of the graph.

The graph Fourier transform to a signal on nodes $x \in \mathbb{R}^n$ is defined as:

$$F(x) = U^T x,$$

and the inverse graph Fourier transform is defined as:

$$F^{-1}(\hat{x}) = U\hat{x},$$

where $\hat{x}$ is the spectral-domain signal resulted from the graph Fourier transform. The graph Fourier transform projects the graph signal $x$ into an orthogonal space formed by the eigenvectors of the normalized graph Laplacian matrix. In this spectral domain, we can define graph convolution as the element-wise (Hadamard) product of a filter $F(g)$ and the transformed signal $F(x)$, after which we can inversely transform the result signal back to the spatial domain:

$$x *_G g := F^{-1}(F(x) \odot F(g))$$

$$= U(U^T x \odot U^T g).$$

(1.8)

If we define $g_\theta = \text{diag}(U^T g)$, then the spectral graph convolution is equivalent to:

$$x *_G g_\theta = U g_\theta U^T x.$$  

(1.9)
All spectral-based graph convolutional neural networks follow this definition and adopt different filters $g_\theta$. The earliest work Spectral Convolutional Neural Network (Spectral-CNN) [18] lets $g_\theta$ be a diagonal matrix of learnable parameters, and considers multiple channels of graph signals. Let $\Theta_{i,j}^t$ be the filter between the $i$th channel of layer $t+1$ and the $j$th channel of layer $t$, the graph convolution layer $t$ is defined as:

$$Z_{i,j}^{t+1} = f(\sum_{i=1}^{c_t} U \Theta_{i,j}^t U^T Z_{i,i}^t),$$

(1.10)

where $Z_{i,i}^t \in \mathbb{R}^{n \times 1}$ denotes the $i$th channel of the graph signal $Z^t \in \mathbb{R}^{n \times c_t}$ in the $t$th layer, and $c_t$ is the number of channels in the $t$th layer. One great limitation of Spectral-CNN is its $O(n^3)$ computation complexity with respect to the number of nodes $n$, due to the expensive eigen-decomposition.

Later, two follow-up works, ChebNet [39] and GCN [77] reduce the complexity from $O(n^3)$ to $O(m)$ ($m$ denotes the number of edges) by making simplifications and approximations to (1.10).

Chebyshev Spectral CNN (ChebNet) [39] makes an approximation to $g_\theta$ by using Chebyshev polynomials of the diagonal matrix of eigenvalues:

$$g_\theta = \sum_{k=0}^{K-1} \theta_k T_k(\hat{\Lambda}), \text{ where } \hat{\Lambda} = 2\Lambda/\lambda_{\text{max}} - I.$$

(1.11)

The Chebyshev polynomials can be recursively computed by:

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x),$$

(1.12)
with \(T_0(x) = 1\) and \(T_1(x) = x\). Then, the original spectral graph convolution (1.9) can be written as:

\[
x \ast_G g_\theta = U \left(\sum_{k=0}^{K-1} \theta_k T_k(\hat{\Lambda})\right) U^\top x
\]

\[
= \sum_{k=0}^{K-1} \theta_k T_k(\hat{L})x,
\]

where \(\hat{L} = 2L/\lambda_{\text{max}} - I\). It is proved that \((L^k)_{i,j} = 0\) for nodes \(i\) and \(j\) with \(d_G(i,j) > K\), where \(d_G(i,j)\) denotes the shortest path length between \(i,j\). Consequently, one advantage of ChebNet compared to Spectral-CNN is its localized convolution filters by restricting the order of the Chebyshev polynomials \(K\). In contrast, the filters in (1.10) are global, meaning that even signals from distant nodes will contribute to the convolution result of a center node. This contradicts with the principle of traditional CNNs for images which leverage localized filters to learn translation invariant features. Thus, ChebNet is more like CNNs than Spectral-CNN.

However, both ChebNet and Spectral-CNN have another limitation – they only work on a single graph structure. This is because the graph Laplacian matrix they rely on is dependent on the global graph structure – any perturbation to the graph structure can result in a change of the eigenbasis and eigenvalues. This has no influence on node-level and edge-level tasks with a single large graph structure given. However, it will pose problems for graph-level tasks, since the learned structure-dependent filters cannot be applied to graphs with different structures.

Instead of defining graph convolutions in the spectral domain, **spatial methods** define graph convolutions in the spatial domain (nodes’ spatial relations with other nodes). Spatial-based graph convolutions are motivated directly by traditional CNNs on images where a center pixel updates its value by a weighted average of the pixel values in a (e.g., \(3 \times 3\)) patch.
around it. Spatial-based GNNs similarly convolve a center node state with its neighbors’ states to get an the updated representation of the center node.

Spatial methods are not restricted to a single graph structure, since all convolutions are done locally within a node’s neighborhood without using the global graph structure. Thus, spatial methods can not only be applied to node-level and edge-level tasks, but also graph-level tasks.

The most popular spatial-based graph neural network is the Graph Convolutional Network (GCN) model [77]. Its convolution form is derived by making a first-order approximation of the ChebNet (1.13). In particular, by using order $K = 1$ and assuming $\lambda_{\text{max}} = 2$, GCN simplifies (1.13) to

$$x *_{G} g_{\theta} = \theta_{0} x - \theta_{1} D^{-\frac{1}{2}} A D^{-\frac{1}{2}} x.$$  \hspace{1cm} (1.15)

Then, GCN makes a further simplification by assuming $\theta_{0} = -\theta_{1} := \theta$, which effectively reduces the number of parameters to only 1 between every input and output channel. The single-channel graph convolution then becomes:

$$x *_{G} g_{\theta} = \theta (I + D^{-\frac{1}{2}} A D^{-\frac{1}{2}}) x.$$  \hspace{1cm} (1.16)

And the multi-channel form of GCN can be written into a matrix multiplication form as follows:

$$Z^{t+1} = f(\hat{A} Z^{t} \Theta),$$  \hspace{1cm} (1.17)

where $\hat{A} := I + D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$ and $f$ is an activation function. However, using $\hat{A} = I + D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$ means that the magnitude of node states in $Z^{t}$ cumulatively gets bigger layer by layer, which
might cause numerical instabilities. In this regard, GCN uses a renormalization trick which replaces \( \hat{\mathbf{A}} \) with \( \tilde{\mathbf{D}}^{-\frac{1}{2}} \hat{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}} \) where \( \hat{\mathbf{A}} := \mathbf{A} + \mathbf{I} \) and \( \mathbf{D} \) is a diagonal degree matrix of \( \hat{\mathbf{A}} \) (\( \tilde{D}_{ii} = \sum_j \hat{A}_{ij} \)).

If we look at individual rows \( \mathbf{z}_i \) of \( \mathbf{Z} \), we can rewrite (1.17) into:

\[
\mathbf{z}'_i = f(\frac{1}{\tilde{D}_{ii}} \mathbf{y}_i + \sum_{j \in \Gamma(i)} \frac{1}{\sqrt{\tilde{D}_{ii} \tilde{D}_{jj}}} \mathbf{y}_j),
\]

where \( \Gamma(i) \) denotes the neighbor set of node \( i \), and \( \mathbf{y}_j := \Theta^\top \mathbf{z}_j \). As we can see, the GCN graph convolution reduces to a weighted sum of the transformed center node state \( \mathbf{y}_i \) and neighbor node states \( \mathbf{y}_j, j \in \Gamma(i) \), which is in a spatial-based graph convolution form. GCN establishes a relationship between spectral methods and spatial methods.

Diffusion Convolutional Neural Network (DCNN) [7] is another early spatial-based GNN. It treats graph convolutions as a diffusion process. It uses a probability transition matrix \( \mathbf{P} = \mathbf{D}^{-1} \mathbf{A} \) to propagate neighbor states from different hops:

\[
\mathbf{Z}^{(k)} = f(\mathbf{W}^{(k)} \odot \mathbf{P}^k \mathbf{X}),
\]

where the final node representations are given by the concatenation of \( \mathbf{Z}^{(1)}, \mathbf{Z}^{(2)}, \ldots, \mathbf{Z}^{(K)} \).

Using a diffusion matrix \( \mathbf{P} \) automatically decreases the contribution of faraway nodes to the center node. In contrast to GCN [77], nodes more than one hop away from the center node directly propagate their states to the center instead of propagating to the center through multiple layers of graph convolution.

In the previous section, we have briefly introduced message passing neural networks (MPNN) [53], which is a uniform framework for spatial-based GNNs. The complete form of a message
passing function is given by

\[ m^{t+1}_v = \sum_{u \in \Gamma(v)} M_t(z^t_v, z^t_u, x^e_{vu}), \]  

\[ z^{t+1}_v = U_t(z^t_v, m^{t+1}_v), \]  

where \( x^e_{vu} \) is the edge feature between nodes \( v, u \), \( M_t \) is the message function used to aggregate neighboring node states into a message \( m_v \), and \( U_t \) is the update function used to update center node \( v \)'s state \( z_v \) based on the message \( m_v \). Edge features are not always available in graph datasets, thus are often ignored in other works, and are more usually handled in different formulations when edge features are discrete and countable which will be discussed in more details in Section 3.2.

Graph Isomorphism Network (GIN) [168] studies the representative power of message passing networks and finds that message passing networks can be at most as powerful as the Weisfeiler-Lehman algorithm [165] by adding an irrational weight \( \epsilon^{t+1} \) in the update function. It uses a graph convolution form as follows:

\[ z^{t+1}_v = f((1 + \epsilon^{t+1})z^t_v + \sum_{u \in \Gamma(v)} W^e z^t_u), \]  

where the irrational number \( \epsilon^{t+1} \) distinguishes the center node from its neighbors when they are assembled together, thus is able to injectively encode the rooted depth-1 subtree of the center node, the same as how the Weisfeiler-Lehman algorithm color nodes (it also uses a perfect hashing function to give unique colors to unique rooted subtree patterns). The Weisfeiler-Lehman algorithm is a powerful tool for distinguishing different graphs, and a promising direction towards efficiently solving the hard graph isomorphism (GI) problem. GIN theoretically shows the graph neural networks can be as powerful as Weisfeiler-Lehman...
algorithm which demonstrates the representative power of GNNs. In practice, GIN lets $\epsilon^{(t+1)}$ be a learnable parameter that is trained together with other parameters.

Graph Attention Network (GAT) [158] does not assume identical contributions of neighboring nodes like previous works such as GCN [77] and GIN [168]. Instead, it leverages an attention mechanism to learn the contribution of each neighboring node to the center node. GAT’s graph convolution form is defined as:

$$z^{t+1}_v = f\left( \sum_{u \in \Gamma(v) \cup v} \alpha_{vu} W^t z^t_u \right),$$

where the attention weight $\alpha_{vu}$ is given by:

$$\alpha_{vu} = \text{softmax}[g(a^\top \text{concat}(W^t z^t_v, W^t z^t_u))],$$

where $g$ is a LeakyReLU activation function and $a$ is a vector of parameters that transforms concatenated node states into a scalar raw weight. And the softmax operation calculates normalized weights so that $\sum_{u \in \Gamma(v) \cup v} \alpha_{vu} = 1$. Multi-head attention can be further used to increase the model capacity. GAT shows improvements over GCN in node classification tasks.

Right now, we have discussed many representative graph neural networks for node-level tasks, mainly in the categories of spectral-based approaches and spatial-based approaches. In Section 2.2, we will discuss our contribution on medical ontology embedding which falls into node-level tasks. We propose a Hierarchical Attention Propagation (HAP) method to learn embeddings of node concepts in a medical ontology such as ICD-9. Our HAP adopts an attention mechanism like GAT, but uses a different propagation order to deal with the special structure of hierarchical medical ontologies. In Section 2.1, we will discuss our contribution to a graph-level task, graph classification, where the node representation learning parts use a message passing form similar to GCN.
1.4.2 GNNs for graph-level tasks

GNNs for graph-level tasks are generally less studied than GNNs for node-level tasks. However, node-level GNNs are essential preliminary steps for a successful graph-level GNN. Concretely speaking, a graph-level GNN is typically composed of a node-level GNN used to extract individual node feature states and a pooling layer used to summarize node states into a graph representation. Our main contribution for graph classification is an advanced pooling layer that takes into account the global topology of a graph.

Neural Graph Fingerprint [43] is an early attempt of graph neural networks on graph-level tasks. It aims at learning differentiable fingerprints (sparse feature vectors) for molecules using neural networks through end-to-end training instead of the previous approaches that use hash functions to map certain local structures to bits. Neural Graph Fingerprint uses a spatial graph convolution to sum node (atom) features around each center atom followed by a linear transformation and nonlinear activation. Then, a softmax function is applied to the further linearly transformed atom feature vector to learn a sparsified feature vector. All the final sparse atom feature vectors are directly summed to construct the final fingerprint for the molecule. Neural Graph Fingerprint uses a sum-based pooling module to aggregate individual node states into a graph representation.

Using a symmetric sum/mean/max is perhaps the simplest way to pool node states. Given final node states $h_1, h_2, \ldots, h_n$, a graph representation $z_G$ is given by:

$$z_G = \text{sum/mean/}\max(h_1, h_2, \ldots, h_n).$$  \hspace{1cm} (1.25)
Many early graph neural networks adopt such symmetric pooling layers for graph-level tasks \cite{7, 35, 43}. Some other works use attention mechanisms to improve sum/mean pooling \cite{53, 94}.

However, these existing methods do not consider the ordering of nodes and treat all nodes symmetrically. This can be problematic for nodes with a natural order, such as directed acyclic graphs (DAGs) with a natural topological ordering of nodes. In Section 4, we will discuss our work on graph neural networks for DAGs, which performs message passing following a topological ordering of nodes in a DAG and uses the final node state as the graph state. We prove that our D-VAE model \cite{185} can injectively encode computations represented by DAGs and thus superior than existing symmetric GNNs when modeling DAGs.

Existing sum/mean/max based pooling layers also directly pool node states into a graph feature vector in one step, which is too large and too rough, and can lose a lot of individual node information as well as the global graph topology. In Section 2.1, we will introduce our work on a novel SortPooling layer for graph classification \cite{184}. SortPooling first sorts nodes according to their structural roles and then keep the top $K$ node states as the graph representation. After SortPooling, traditional 1D convolutions are applied on the node sequences to learn from both individual node states and global graph topology within the node ordering. The proposed SortPooling also inspired many follow-up works studying advanced graph pooling layers, such as DiffPool \cite{173} and SAGPooling \cite{91}.

### 1.4.3 GNNs for edge-level tasks

Edge-level tasks are least studied using GNNs. However, recent GNN-based link prediction algorithms \cite{181} and recommender systems \cite{berg2017graph, 172} have shown GNNs’ great potentials for this type of problems.
Edge-level tasks have been studied using both node-level GNNs and graph-level GNNs. Node-based methods combine the two end nodes’ states learned by a node-level GNN as an edge’s feature vector, which is then used for edge-level tasks such as link prediction. For example, Varitional Graph Autoencoder (VGAE) [78] first applies GCN [77] to extract node states, and use the inner product of two node states to reconstruct the existing edge between them. After training, a feature vector is learned for each node, which can be used to predict unseen links. It is similar to matrix factorization techniques [82] with the difference that the latent factors are learned through GNNs.

Later, VGAE-typed GNNs are generalized to recommender systems. For example, [113] uses multi-graph CNN model to extract user and item latent features from their respective networks and use the latent features to predict the ratings. [berg2017graph] proposes graph convolutional matrix completion (GC-MC) to directly operate on user-item bipartite graphs to extract user and item latent features using a GNN equipped with relational graph convolution operators [138] that assign different weight matrices to different edge types. Pinsage [172] also uses GNNs to learn node features from the rich content features provided by each pin, and is successfully used in recommending related pins in Pinterest.

Our work is different from existing node-based approaches that use two node states to represent an edge. Instead, we propose to use graph-level GNNs to learn edge features from local enclosing subgraphs around edges. The advantages are: 1) We are able to learn from the rich topological features within the neighborhood of each edge, rather than learning two nodes’ local substructure features independently and combining them later. In link prediction, such pure topological features are very important link predictors, and are extensively studied in previous works known as link prediction heuristics [96]. However, previous works mainly use manually defined heuristics. In Section 3.1, we will discuss our work SEAL [181] that automatically learns link prediction heuristics from networks themselves. 2) Another
advantage is that the learned topological features are inductive, in contrast to node-based approaches that often learn transductive latent features of nodes. The inductive features are not only generalizable to nodes unseen during training, but also transferrable to new tasks. In Section 3.2, we will discuss our work on inductive graph-based matrix completion (IGMC) [180], which uses a graph-level GNN to perform inductive matrix completion without resorting to any node content (side) features.
In this section, we introduce our work on GNN-based graph representation learning. Learning a good graph representation is the first-step for many graph learning problems. There are two levels of representations to learn, one is node representations (or node embeddings) within a given graph or network; the other is representation for the entire graph.

Learning representations for entire graphs enables graph-level learning tasks, such as graph classification. Our first contribution in the dissertation is to propose a novel graph neural network architecture specifically designed for graph classification. The significance is that we for the first time study advanced pooling layers rather than the simple summing/averaging used in previous GNNs. The pooling layer we propose is called SortPooling, which sorts vertex features in a meaningful order before feeding to the later layers. This enables learning from the global graph topology and alleviates the great information loss occurs in summing/averaging.
Our proposed Deep Graph Convolutional Neural Network (DGCNN) [184] achieves state-of-the-art graph classification results.

Learning representations for nodes are equally important as learning graph representations. Like word embedding’s importance to natural language processing, learning node representations can facilitate many network analysis problems. In this dissertation, we focus on a particular kind of networks, medical code networks, or medical ontologies. There are many well developed medical ontologies such as the ICD-9 and ICD-10, which hierarchically organize medical concepts into categories/subcategories, providing a valuable source of domain knowledge that can potentially improve healthcare systems’ performance. To learn medical ontology embeddings, we propose a Hierarchical Attention Propagation (HAP) model, which hierarchically propagate attention across the medical ontology. We prove that HAP learns most expressive medical concept embeddings – from any medical concept embedding we are able to fully recover the entire ontology structure. Experimental results on sequential procedure/diagnosis prediction tasks using real patient data demonstrate HAP’s superior predictive performance.

2.1 Graph Neural Networks for Graph Classification

2.1.1 Traditional graph classification methods: graph kernels

Given a dataset containing graphs in the form of \((G, y)\) where \(G\) is a graph and \(y\) is its class, graph classification is to learn a function mapping \(G\) to its class \(y\). For example, in bioinformatics, we may need to classify molecules into enzymes or not. In material science, we may need to classify whether a material has a given property. Traditional machine learning
algorithms such as SVMs and neural networks cannot directly classify graphs, since graphs often do not have fixed-size tensor representations as input to the algorithms.

Graph kernels make kernel machines such as kernel SVMs feasible for graph classification by computing some positive semidefinite graph similarity measures, which have achieved state-of-the-art classification results on many graph datasets [142, 159]. A pioneering work was introduced as the convolution kernel in [59], which decomposes graphs into small substructures and computes kernel functions by adding up the pair-wise similarities between these components. Common types of substructures include walks [159], subgraphs [85], paths [15], and subtrees [116, 142]. [123] reformulated many well-known substructure-based kernels in a general way called graph invariant kernels. [171] proposed deep graph kernels which learn latent representations of substructures to leverage their dependency information. Convolution kernels compare two graphs based on all pairs of their substructures. Assignment kernels, on the other hand, tend to find a correspondence between parts of two graphs. [8] proposed aligned subtree kernels incorporating explicit subtree correspondences. [84] proposed the optimal assignment kernels for a type of hierarchy-induced kernels. Most existing graph kernels focus on comparing small local patterns. Recent studies show comparing graphs more globally can improve the performance [81, 114]. [35] represented each graph using a latent variable model and then explicitly embedded them into feature spaces in a way similar to graphical model inference. The results compared favorably with standard graph kernels in both accuracy and efficiency.

However, graph kernels require at least quadratic time and space complexity to compute and store the kernel matrices, which is unsuitable for modern large-scale practical problems. In addition, existing graph kernels are often designed heuristically – no principled way exists to measure graph similarities. This motivates people to study GNNs for graph classification.
GNNs are actually closely related to a type of graph kernels based on structure propagation, especially the Weisfeiler-Lehman (WL) subtree kernel [142] and the propagation kernel (PK) [116]. To encode the structural information of graphs, WL and PK iteratively update a node’s feature based on its neighbors’ features. WL operates on hard vertex labels, while PK operates on soft label distributions. As this operation can be efficiently implemented as a random walk, these kernels are efficient on large graphs. Compared to WL and PK, GNNs has additional parameters $W$ between propagations which are trained through end-to-end optimization. This allows supervised end-to-end feature learning from the label information, making it different from the two-stage framework of graph kernels.

### 2.1.2 Limitations of existing GNNs for graph classification

To use GNNs for graph classification, a pooling (readout) operation needs to be performed to aggregate node features extracted by message passing layers into a graph representation. Existing GNNs simply use summing or averaging. Two great limitations of using summing/averaging to aggregate node states are that 1) it loses much information of individual nodes, and 2) it does not allow learning from the global graph topology. A graph may have over hundreds or thousands of nodes, yet after summing/averaging, all the node states are reduced to one single vector, which is a too large and too rough step for learning the graph-level feature vector. In addition, the summing-based aggregation loses the graph topology entirely. Specifically, although the final node states summarize the local topology patterns around nodes, the global topology such as how nodes are positioned relatively to each other within the graph and which nodes share symmetric structural roles within the graph, etc., are all lost. Therefore, the summing-based aggregation can only classify graphs based on local patterns, but loses the ability to learn from the more global graph topology.
To address the problems of summing-based aggregation, we propose Deep Graph Convolutional Neural Network (DGCNN). DGCNN uses a simplified message passing form, and a novel sorting-based aggregation named SortPooling, which sorts vertex states according to vertices’ structural roles such that individual node information and the global topology are preserved. Then, it applies 1-D convolutions to the node sequences to learn from the global graph topology.

**Message passing layers.** We first introduce the message passing (graph convolution) layers of DGCNN. For node \(v\), the message passing takes the following form:

\[
\begin{align*}
\mathbf{m}^{t+1}_v &= \frac{1}{|\Gamma(v)| + 1} \left( z^t_v + \sum_{u \in \Gamma(v)} z^t_u \right), \\
\mathbf{z}^{t+1}_v &= f(\mathbf{W}^t \mathbf{m}^{t+1}_v),
\end{align*}
\]

where \(f\) is an element-wise nonlinear transformation such as tanh, \(\mathbf{W}^t\) is a learnable parameter matrix. The above formulation first calculates the message \(\mathbf{m}^{t+1}_v\) by averaging the vertex states of \(v\) and \(v\)’s neighbors. Then, a one-layer feedforward neural network is applied to \(\mathbf{m}^{t+1}_v\) to output \(v\)’s state at next time step. It is a particular realization of (1.1) and (1.2), working pretty well in practice.

If we vertically (row-wise) stack the node states \(z^t_v\) into a matrix \(\mathbf{Z}^t\), where the node order is the same as in the adjacency matrix \(A\) of the graph, then we can have a matrix formulation of the above message passing:

\[
\mathbf{Z}^{t+1} = f(\tilde{\mathbf{D}}^{-1} \tilde{\mathbf{A}} \mathbf{Z}^t \mathbf{W}^t),
\]
where $\tilde{A} = A + I$, $\tilde{D}$ is a diagonal degree matrix with $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$. It reduces to the vector forms (2.1) and (2.2) if we split the above calculations into rows.

After multiple message passing layers, we concatenate the outputs $Z^t, t = 1, \ldots, T$ horizontally, written as $Z^{1:T} := [Z^1, \ldots, Z^T]$. In the concatenated output $Z^{1:T} \in \mathbb{R}^{n \times c}$ where $n$ is the number of nodes and $c$ is the total number of feature channels, each row can be regarded as a “feature descriptor” of a vertex, encoding its multi-hop local substructure information.

**The SortPooling layer.** Next, we introduce the SortPooling layer, which is used to replace the plain summing layer in previous work. We notice that images and many other types of data are naturally presented with some order. For example, image pixels are arranged in a spatial order, and document words are presented in a sequential order. Figure 2.1 gives an example. Graphs, on the other hand, usually lack a tensor representation with fixed ordering. Thus, can we sort graph nodes ourselves to attach an order to graphs?

The main function of the SortPooling layer is to sort the feature descriptors, each of which represents a vertex, in a consistent order before feeding them into 1-D convolutional layers. The question is by what order should we sort the vertices? In image classification, pixels are naturally arranged with some spatial order. In text classification, we can use dictionary order...
to sort words. In graphs, we can sort vertices according to their *structural roles* within the graph. The structural roles of nodes can be given by the Weisfeiler-Lehman (WL) algorithm [165], which iteratively encodes nodes’ neighborhoods into integer colors, so that the same neighborhoods are encoded into the same color and different neighborhoods are encoded into different colors. After convergence, the WL colors can mark the relative structural positions of the nodes within the graph.

We notice that our message passing scheme shares the same idea as WL – it also iteratively encodes neighborhoods into vertex states, except for using continuous hidden states instead of integer colors and using a learnable encoding function. We thus can regard the hidden states $Z^t$, $t = 1, \ldots, T$ as the *continuous WL colors*, and use these continuous WL colors to sort the vertices.

Given the $n \times c$ input $Z^{1:T}$, where each row is a vertex’s feature descriptor and each column is a feature channel, the output of SortPooling is a $k \times c$ tensor, where $k$ is a user-defined integer. In the SortPooling layer, the input $Z^{1:T}$ is first sorted row-wise according to $Z^T$. We can regard these final hidden states as the vertices’ most refined continuous WL colors, and sort all the vertices using these final colors. This way, a consistent ordering is imposed for graph vertices, making it possible to train traditional neural networks on the sorted graph representations. Ideally, we need the graph convolution layers to be deep enough (meaning $T$ is large), so that $Z^T$ is able to partition vertices into different colors/groups as finely as possible.

The vertex order based on $Z^T$ is calculated by first sorting vertices using the last channel of $Z^T$ in a descending order. If two vertices have the same value in the last channel, the tie is broken by comparing their values in the second to last channel, and so on. If ties still exist, we continue comparing their values in $Z^T_{i-1}$, $Z^T_{i-2}$, and so on until ties are broken. Such an
Figure 2.2: The overall structure of DGCNN. An input graph is first passed through multiple message passing layers where node information is propagated between neighbors. Then the vertex features are sorted and pooled with a SortPooling layer, and passed to 1-D convolutional layers to learn a predictive model.

order is similar to the lexicographical order, except for comparing sequences from right to left. We can prove that such a sorting scheme ensures permutation invariance which is important for graph isomorphism.

In addition to sorting vertex features in a consistent order, the next function of SortPooling is to unify the sizes of the output tensors. After sorting, we truncate/extend the output tensor in the first dimension from $n$ to $k$. The intention is to unify graph sizes, making graphs with different numbers of vertices unify their sizes to $k$. The unifying is done by deleting the last $n - k$ rows if $n > k$, or adding $k - n$ zero rows if $n < k$.

As a bridge between graph convolution layers and traditional layers, SortPooling has another great benefit in that it can pass loss gradients back to previous layers by remembering the sorted order of its input, making the training of previous layers’ parameters feasible.

After SortPooling, traditional 1-D convolutions are applied to the sorted node representations, similar to how convolutional filters move on image pixels. Figure 2.2 illustrates the overall architecture of DGCNN.
2.1.4 Training through backpropagation

The whole network can be trained efficiently through backpropagation. Let $\mathcal{L}$ denote the loss of a graph sample. It is standard to compute the gradients of $\mathcal{L}$ w.r.t. the traditional layers’ parameters and inputs. Here we show how to do it for graph convolution and SortPooling layers.

Let $P \in \{0, 1\}^{k \times n}$ and $Z^{sp} \in \mathbb{R}^{k \times \sum_{i}c_{i}}$ be the permutation matrix and output, respectively in the forward propagation of SortPooling, where $P_{ij} = 1$ if the $j^{th}$ row of $Z^{1:h}$ is ranked $i^{th}$ in $Z^{sp}$ and 0 otherwise. We have

$$Z^{sp} = PZ^{1:h}, \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial Z^{1:h}} = P^\top \frac{\partial \mathcal{L}}{\partial Z^{sp}}. \quad (2.4)$$

For the first graph convolution layer, we let $V := \tilde{D}^{-1} \tilde{A}XW$. Thus, $Z = f(\tilde{D}^{-1} \tilde{A}XW) = f(V)$. We have

$$\frac{\partial \mathcal{L}}{\partial X} = \tilde{D}^{-1} \tilde{A} \frac{\partial \mathcal{L}}{\partial V} W^\top, \quad \frac{\partial \mathcal{L}}{\partial W} = X^\top \tilde{D}^{-1} \tilde{A} \frac{\partial \mathcal{L}}{\partial V}, \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial V} = \frac{\partial \mathcal{L}}{\partial Z} \circ f'(V). \quad (2.5)$$

For the stacked graph convolution layers, we also let $V^t := \tilde{D}^{-1} \tilde{A}Z^t W^t$. Thus $Z^{t+1} = f(\tilde{D}^{-1} \tilde{A}Z^t W^t) = f(V^t)$. Here we need to further consider the gradients from the direct connection between $Z^t$ and $Z^{1:h}$. The complete loss gradient w.r.t. $Z^t$ is

$$\frac{\partial \mathcal{L}}{\partial Z^t} = \tilde{D}^{-1} \tilde{A} \frac{\partial \mathcal{L}}{\partial V^t} W^{t\top} + \left[ \frac{\partial \mathcal{L}}{\partial Z^{1:h}} \right]_{\{t\}}, \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial V^t} = \frac{\partial \mathcal{L}}{\partial Z^{t+1}} \circ f'(V^t), \quad (2.6)$$

where we use $\left[ \frac{\partial \mathcal{L}}{\partial Z^{t+1}} \right]_{\{t\}}$ to denote the $c_t$ columns in $\frac{\partial \mathcal{L}}{\partial Z^{t+1}}$ corresponding to $Z^t$. 

30
2.1.5 Discussion

Connection with Weisfeiler-Lehman subtree kernel

The Weisfeiler-Lehman subtree kernel [142] is a state-of-the-art graph kernel, which leverages the Weisfeiler-Lehman (WL) algorithm [165] as a subroutine to extract multi-scale subtree features for graph classification.

The basic idea of WL is to concatenate a vertex’s color with its 1-hop neighbors’ colors as the vertex’s WL signature, and then sort the signature strings lexicographically to assign new colors. Vertices with the same signature are assigned the same new color. A WL signature characterizes the height-1 subtree rooted at a vertex. The procedure is repeated until the colors converge or reaching some maximum iteration $h$. In the end, vertices with the same converged color share the same structural role within the graph and cannot be further distinguished. A vertex color at any iteration $t$ uniquely corresponds to a height-$t$ subtree rooted at the vertex.

WL is widely used in graph isomorphism checking: if two graphs are isomorphic, they will have the same multiset of WL colors at any iteration. The WL subtree kernel uses this idea to measure the similarity between two graphs $G$ and $G'$ as follows:

$$k(G, G') = \sum_{t=0}^{h} \sum_{v \in V} \sum_{v' \in V'} \delta(c^t(v), c^t(v')),$$

where $c^t(v)$ is the (integer) color of vertex $v$ in the $t^{th}$ iteration, and $\delta(x, y) = 1$ if $x = y$ and 0 otherwise. That is, it counts the common colors of two graphs in all iterations. The intuition is that two graphs are similar if they have many common subtrees rooted at their vertices, which are characterized by colors (same color $\iff$ same WL signature $\iff$ same rooted subtree).
The WL subtree kernel counts the common colors until iteration $h$ in order to compare two graphs at multiple scales.

To show the relation between the graph convolution in (2.3) and the WL subtree kernel, we rewrite $Y^t := X^t W^t$, and decompose (2.3) row-wise as follows:

$$Z^{t+1}_i = f([\hat{D}^{-1} \hat{A}]_i Y^t) = f(\hat{D}^{-1}_i (Y^t_i + \sum_{j \in \Gamma(i)} Y^t_j)).$$

In (2.8), we can view $Y^t_i$ as a continuous color of vertex $i$. In analogy to WL, (2.8) also aggregates $Y^t_i$ and its neighboring colors $Y^t_j$ into a WL signature vector $D^{-1}_i (Y^t_i + \sum_{j \in \Gamma(i)} Y^t_j)$. The nonlinear function $f$ maps unique WL signature vectors to unique continuous new colors if $f$ is injective. Therefore, the graph convolution (2.3) may be viewed as a “soft” version of the WL algorithm.

The soft version of WL has two benefits over the original WL. First, the convolution parameters $W^t$ allow hierarchical feature extraction of nodes’ original information matrix $X$, and are trainable through backpropagation, enabling better expressing power than the WL subtree kernel. Second, the soft WL is easy to compute using sparse matrix multiplication, avoiding the need to read and sort the possibly very long WL signature strings.

**Permutation invariance of the learned graph representation**

One important criterion of graph neural network design is that the network should map isomorphic graphs to the same representation and output the same prediction, otherwise any permutation in the adjacency matrix could result in a different prediction for a same graph. For summing-based methods, this is not an issue, as summing is invariant to vertex permutation. However, for sorting-based methods DGCNN and PATCHY-SAN [121], additional care is required. To ensure that isomorphic graphs are preprocessed to the same tensor,
PATCHY-SAN first uses the WL algorithm and then leverages Nauty, a graph canonization tool [109]. Although Nauty is efficient enough for small graphs, the problem of graph canonization is theoretically at least as computationally hard as graph isomorphism checking.

In comparison, we show that such a graph canonization step can be avoided in DGCNN. DGCNN sorts vertices using the last graph convolution layer’s outputs, which we show can be viewed as the continuous colors output by a “soft” WL. Thus, DGCNN is able to sort vertices as a by-product of graph convolution, which avoids explicitly running the WL algorithm like PATCHY-SAN. Moreover, due to the sorting scheme in SortPooling, graph canonization is no longer needed.

**Theorem 2.1.** In DGCNN, if two graphs $G_1$ and $G_2$ are isomorphic, their graph representations after SortPooling are the same.

**Proof.** Notice that the first phase’s graph convolutions are invariant to vertex indexing. Thus if $G_1$ and $G_2$ are isomorphic, they will have the same multiset of vertex feature descriptors after graph convolution. Since SortPooling sorts vertices in such a way that two vertices have a tie if and only if they have exactly the same feature descriptor, the sorted representation is invariant to which of the two vertices is ranked higher. Hence, $G_1$ and $G_2$ have the same sorted representation after SortPooling.

Therefore, DGCNN needs to explicitly run neither WL nor Nauty, which frees us from data preprocessing and external software, and provides a pure neural network architecture for end-to-end graph classification. We also comment that although DGCNN sorts vertices dynamically during training, the vertex order will gradually become stable with increasing training epochs. This is because the parameters $W$ are shared among all vertices. The updating of $W$ will increase or decrease the continuous WL colors of all vertices simultaneously.
Moreover, the learning rate of $W$ is iteratively decayed during training, making the overall vertex order stable over the course.

2.1.6 Experimental results

We conduct experiments on benchmark datasets to evaluate the performance of DGCNN against state-of-the-art graph kernels and other deep learning approaches. The code and data are available at https://github.com/muhanzhang/DGCNN.

Comparison with graph kernels

Datasets. We use five benchmark bioinformatics datasets to compare the graph classification accuracy of DGCNN with graph kernels. The datasets are: MUTAG, PTC, NCI1, PROTEINS, D&D. MUTAG [38] is a dataset of 188 mutagenic aromatic and heteroaromatic nitro compounds classified according to their mutagenic effects on a bacterium. PTC [156] is a dataset of 344 chemical compounds classified according to their carcinogenicity for male and female rats. NCI1 [161] contains anti-cancer screens for cell lung cancer and ovarian cancer cell lines. PROTEINS and D&D are graph collections of chemical compounds classified into two classes: enzyme and non-enzyme [41]. Graphs in these five bioinformatics datasets all have vertex labels. Only MUTAG and PTC contain edge labels, which are not used in this paper.

Baselines and experimental setting. We compare DGCNN with four graph kernels: the graphlet kernel (GK) [143], the random walk kernel (RW) [159], the propagation kernel (PK) [117], and the Weisfeiler-Lehman subtree kernel (WL) [142]. Due to the large literature, we could not compare to every graph kernel, but to some classical ones and those closely related to our approach. Following the conventional settings, we performed 10-fold cross validation
with LIBSVM [21] (9 folds for training and 1 fold for testing) using one training fold for hyperparameter searching, and repeated the experiments for 10 times (thus **100 runs per dataset**). The average accuracies and their standard deviations are reported. We searched the height parameter of WL and PK in \{0, 1, 2, 3, 4, 5\}, and set the bin width \(w\) of PK to 0.001. We set the size of the graphlets for GK to 3. We set the decay parameter \(\lambda\) of RW to the largest power of 10 that is smaller than the reciprocal of the squared maximum node degree as suggested in [142].

For the proposed DGCNN, to make a fair comparison with graph kernels, we used a **single network structure** on all datasets, and ran DGCNN using exactly the same folds as used in graph kernels in all the **100 runs** of each dataset. The network has four graph convolution layers with 32, 32, 32, 1 output channels, respectively. For convenience, we set the last graph convolution to have one channel and only used this single channel for sorting. We set the \(k\) of SortPooling such that 60% graphs have nodes more than \(k\). The remaining layers consist of two 1-D convolutional layers and one dense layer. The first 1-D convolutional layer has 16 output channels followed by a MaxPooling layer with filter size 2 and step size 2. The second 1-D convolutional layer has 32 output channels, filter size 5 and step size 1. The dense layer has 128 hidden units followed by a softmax layer as the output layer. A dropout layer with dropout rate 0.5 is used after the dense layer. We used the hyperbolic tangent function (\(\tanh\)) as the nonlinear function in graph convolution layers, and rectified linear units (ReLU) in other layers. Stochastic gradient descent (SGD) with the ADAM updating rule [76] was used for optimization. The only hyperparameters we optimized are the learning rate and the number of training epochs (details in the supplementary material). We implemented SortPooling and graph convolution layers using Torch [33] as standard \(nn\) modules, which can be seamlessly added to existing Torch architectures.
Table 2.1: DGCNN’s comparison with graph kernels.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MUTAG</th>
<th>PTC</th>
<th>NCI1</th>
<th>PROTEINS</th>
<th>D&amp;D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes (max)</td>
<td>28</td>
<td>109</td>
<td>111</td>
<td>620</td>
<td>5748</td>
</tr>
<tr>
<td>Nodes (avg.)</td>
<td>17.93</td>
<td>25.56</td>
<td>29.87</td>
<td>39.06</td>
<td>284.32</td>
</tr>
<tr>
<td>Graphs</td>
<td>188</td>
<td>344</td>
<td>4110</td>
<td>1113</td>
<td>1178</td>
</tr>
<tr>
<td>DGCNN</td>
<td>85.83±1.66</td>
<td>58.59±2.47</td>
<td>74.44±0.47</td>
<td>75.54±0.94</td>
<td>79.37±0.94</td>
</tr>
<tr>
<td>GK</td>
<td>81.39±1.74</td>
<td>55.65±0.46</td>
<td>62.49±0.27</td>
<td>71.39±0.31</td>
<td>74.38±0.69</td>
</tr>
<tr>
<td>RW</td>
<td>79.17±2.07</td>
<td>55.91±0.32</td>
<td>&gt;3 days</td>
<td>59.57±0.90</td>
<td>&gt;3 days</td>
</tr>
<tr>
<td>PK</td>
<td>76.00±2.69</td>
<td><strong>59.50±2.44</strong></td>
<td>82.54±0.47</td>
<td>73.68±0.68</td>
<td>78.25±0.51</td>
</tr>
<tr>
<td>WL</td>
<td>84.11±1.91</td>
<td>57.97±2.49</td>
<td><strong>84.46±0.45</strong></td>
<td>74.68±0.49</td>
<td>78.34±0.62</td>
</tr>
</tbody>
</table>

Results. Table 2.1 lists the results. As we can see, although a single structure was used for all datasets, DGCNN achieved highly competitive results with the compared graph kernels, including achieving the highest accuracies on MUTAG, PROTEINS, and D&D. Compared to WL, DGCNN has higher accuracies on all datasets except for NCI1, indicating that DGCNN is able to utilize node and structure information more effectively. Note that the height parameters in PK and WL were tuned individually for each dataset by searching from \( \{0,1,2,3,4,5\} \), while DGCNN used a **single height** \( h=4 \) for **all datasets**. Thus, we expect better performance if we use different structures for different datasets.

We compare the efficiency of DGCNN with one of the most efficient graph kernels, the WL kernel, on D&D, the benchmark dataset with the largest graph size. We omit the SVM training time of WL, since the computing time is dominated by the kernel computation. WL takes 252 seconds to construct the kernel matrix. For DGCNN, the training time varies with the iteration number. Here we limit the iteration number to 10, under which condition DGCNN already achieves comparable or better accuracy than WL. DGCNN takes 156 seconds, meaning that it is able to achieve competitive efficiency with the fastest graph kernels. Moreover, DGCNN is trained through SGD, avoiding the at least quadratic complexity w.r.t. the number of graphs required for graph kernels. Therefore, we expect to see a much greater advantage when applying to industrial-scale graph datasets.
DGCNN’s comparison with other deep approaches

Datasets. We compare DGCNN with other deep learning approaches for graph classification on six datasets, including three benchmark bioinformatics datasets: NCI1, PROTEINS, and D&D, as well as three social network datasets: COLLAB, IMDB-B, IMDB-M [171]. Graphs in these social network datasets do not have vertex labels, thus are pure structures. COLLAB is a scientific collaboration dataset where ego-networks are generated for researchers and are classified into three research fields. IMDB-B is a movie collaboration dataset where ego-networks for actors/actresses are classified into Action or Romance genres. IMDB-M is a multi-class version of IMDB-B containing genres Comedy, Romance, and Sci-Fi. The COLLAB, IMDB-B, and IMDB-M are from [171]. Graphs in these social network datasets contain neither vertex labels nor edge labels, thus are pure structures. We exclude the two smallest bioinformatics datasets: MUTAG and PTC, which only have a few hundred examples, since deep learning methods easily overfit them, reporting abnormally high variance in previous works [121].

Baselines and experimental setting. We compare DGCNN with four other deep learning approaches: including three recent neural network approaches for graph classification (PSCN, DCNN, and ECC), and a deep graph kernel approach (DGK). Among them, PATCHY-SAN (PSCN) [121] is the closest to ours. Diffusion-CNN (DCNN) [7] uses diffusion graph convolutions to extract multi-scale substructure features. ECC [145] can be viewed as a hierarchical version of the Neural Fingerprints [43]. Both DCNN and ECC use summed node features for graph classification. The Deep Graphlet Kernel (DGK) [171] learns substructure similarities via word embedding techniques. For PSCN, ECC, and DGK, we report the best results from the papers, as they were under the same setting as ours. For DCNN, since the original experiment split the training/validation/testing data equally, we redid the experiment
Table 2.2: Comparison with other deep learning approaches.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>NCI1</th>
<th>PROTEINS</th>
<th>D&amp;D</th>
<th>COLLAB</th>
<th>IMDB-B</th>
<th>IMDB-M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes (max)</td>
<td>111</td>
<td>620</td>
<td>5748</td>
<td>492</td>
<td>136</td>
<td>89</td>
</tr>
<tr>
<td>Nodes (avg.)</td>
<td>29.87</td>
<td>39.06</td>
<td>284.32</td>
<td>74.49</td>
<td>19.77</td>
<td>13.00</td>
</tr>
<tr>
<td>Graphs</td>
<td>4110</td>
<td>1113</td>
<td>1178</td>
<td>5000</td>
<td>1000</td>
<td>1500</td>
</tr>
<tr>
<td>DGCNN</td>
<td>74.44±0.47</td>
<td>75.54±0.94</td>
<td>79.37±0.94</td>
<td>73.76±0.49</td>
<td>70.03±0.86</td>
<td>47.83±0.85</td>
</tr>
<tr>
<td>PSCN</td>
<td>76.34±1.68</td>
<td>75.00±2.51</td>
<td>76.27±2.64</td>
<td>72.60±2.15</td>
<td>71.00±2.29</td>
<td>45.23±2.84</td>
</tr>
<tr>
<td>DCNN</td>
<td>56.61±1.04</td>
<td>61.29±1.60</td>
<td>58.09±0.53</td>
<td>52.11±0.71</td>
<td>49.06±1.37</td>
<td>33.49±1.42</td>
</tr>
<tr>
<td>ECC</td>
<td>76.82</td>
<td>–</td>
<td>72.54</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>DGK</td>
<td>62.48±0.25</td>
<td>71.68±0.50</td>
<td>–</td>
<td>73.09±0.25</td>
<td>66.96±0.56</td>
<td>44.55±0.52</td>
</tr>
<tr>
<td>DGCNN (sum)</td>
<td>69.00±0.48</td>
<td>76.26±0.24</td>
<td>78.72±0.40</td>
<td>69.45±0.11</td>
<td>51.69±1.27</td>
<td>42.76±0.97</td>
</tr>
</tbody>
</table>

using the standard setting. Among these methods, PSCN and ECC can leverage additional edge features. These augmented results are not reported here since edge features are missing from most graph datasets and all the other compared methods do not leverage edge features.

For DGCNN, we still use the same structure as when comparing with graph kernels, in order to show its robust performance across different datasets under a single structure. Since the new added social network datasets do not contain node labels, we set the $k$ of SortPooling such that 90% graphs have nodes more than $k$ in order to compensate the loss of node features.

**Results.** Table 2.2 lists the results. DGCNN shows the highest accuracies on PROTEINS, D&D, COLLAB, and IMDB-M. Compared to PATCHY-SAN, the improvement of DGCNN can be explained as follows. 1) By letting gradient information backpropagate through SortPooling, DGCNN enables parameter training even before the sorting begins, thus achieving better expressibility. 2) By sorting nodes on the fly, DGCNN is less likely to overfit a particular node ordering. In comparison, PATCHY-SAN sticks to a predefined node ordering. Another huge advantage of DGCNN is that it provides a unified way to integrate preprocessing into a neural network structure. This frees us from using any external software.
DGCNN shows significant accuracy improvement over DCNN which uses summed node features for classification. Another summing-based method, ECC, is slightly better on NCI1, but much worse on D&D. These results meet our expectation since summing will lose much individual node and global topology information. Compared to DGK, DGCNN shows better performance on all the reported datasets.

To demonstrate the advantage of SortPooling over summing, we further list the results of DGCNN (sum), which replaces the SortPooling and later 1-D convolution layers in DGCNN with a summing layer. As we can see, the performance worsens a lot in most cases.

Ablation studies of SortPooling

In order to better understand the effect of SortPooling, we conducted four supplementary experiments on MUTAG. In the first experiment, we turned off SortPooling and fed graphs into DGCNN with fixed random vertex orders. In the second experiment, we turned off SortPooling, and fed into DGCNN graphs with vertex orders calculated by the Weisfeiler-Lehman algorithm in a preprocessing step. In the third experiment, we replaced SortPooling with the summed node features as the input to later layers. In the fourth experiment, we turned on SortPooling, but did not update the weights of the previous graph convolution layers (the weights are fixed all-one matrices).

We show the training curves of the four experiments compared to using the original SortPooling in Figure 2.3. As we can see, the validation loss curve of using the original SortPooling is always lower than those without SortPooling. In the first experiment, we can see that using random vertex orders resulted in easily overfitting the training data. This is because training and validation graphs are fed into neural networks without a consistent vertex order. This experiment verifies the importance of the “sort” function in SortPooling, which provides a
Figure 2.3: Training curves of SortPooling compared with 1) random order, 2) precomputed WL order, 3) summed node features, and 4) fixed weights.

consistent vertex ordering based on soft WL colors. In the second experiment, the training curve of SortPooling coincides well with that of using precomputed WL orders. This meets our expectation since SortPooling is designed to use the continuous WL colors for sorting. Interestingly, we find that the validation curve of SortPooling is lower than that of using the precomputed WL orders. This is because SortPooling has an additional regularization effect by dynamically sorting vertices. By dynamically sorting vertices during training, vertices are less sensitive to the precomputed fixed orders, which leads to a lower risk of overfitting. This experiment verifies the regularization effect of SortPooling. In the third experiment, we can see that using summed node features have consistently higher training and validation loss than SortPooling, meaning that it is much more difficult for summing to fit the data than
Table 2.3: Accuracy results on MUTAG for the supplementary experiments.

<table>
<thead>
<tr>
<th></th>
<th>Random order</th>
<th>Precomputed WL order</th>
<th>Summed node features</th>
<th>Fixed weights</th>
<th>SortPooling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>78.72±3.04</td>
<td>82.56±2.71</td>
<td>78.50±2.43</td>
<td>82.50±2.74</td>
<td><strong>85.83±1.66</strong></td>
</tr>
</tbody>
</table>

for sorting. By summing, individual vertex features are averaged out instead of being kept, leading to a great loss of information. This experiment verifies SortPooling’s advantages of keep more vertex information thus achieving higher model capacity. We used the fourth experiment to verify the usefulness of training the graph convolution parameters. As we can see, the inclusion of trainable graph convolution parameters leads to better expressing ability. This experiment verifies the fourth advantage of SortPooling, i.e., allowing backpropagation training of previous graph convolution layers’ parameters.

With the same experimental settings, we further conducted 10 times 10-fold cross validations to test each variant’s performance. The accuracies and standard deviations are shown in Table 2.3. As we can see, using random vertex orders or summed node features have much lower accuracies than other variants. Compared to precomputed WL orders, the network with SortPooling increases the accuracy by 3%. Compared to using fixed weights in graph convolutions, SortPooling which backpropagates loss gradients and updates weights also shows about 3% accuracy improvement.

### 2.1.7 Conclusion

In this section, we have introduced a novel graph neural network architecture, DGCNN, for graph classification. DGCNN has a number of advantages over existing graph neural networks. First, it directly accepts graph data as input without the need of first transforming graphs into tensors, making end-to-end gradient-based training possible. Second, it enables learning from global graph topology by sorting vertex features instead of summing them up,
which is supported by a novel SortPooling layer. Finally, it achieves better performance than existing methods on many benchmark datasets.

2.2 Graph Neural Networks for Medical Ontology Embedding

2.2.1 Introduction

In recent years, tremendous effort has been put into developing intelligent healthcare prediction systems leveraging the power of big data and machine learning [68, 79]. Existing systems often face one challenge: how to effectively combine power of data-driven algorithms with **domain knowledge** from human experts for a better predictive performance. A pure data-driven model often requires a huge data volume to achieve a satisfying performance, and typically has a poor performance when predicting cases rarely present in the training data [27]. A proper incorporation of structured domain knowledge, such as the categorization or grouping relationships among different medical concepts might alleviate such problems. For example, disease $a$ and $b$ are both subclasses of disease class $c$. Then, we may expect $a$ and $b$ to have similar properties, so that even $b$ rarely appears in the training data, we could still learn from $a$ to predict $b$.

Luckily, there exist many well-established ontologies of medical concepts encoding such structured domain knowledge. Examples include International Classification of Diseases (ICD) [122], Clinical Classifications Software (CCS) [151], Unified Medical Language System (UMLS) [13], Systematized Nomenclature of Medicine-Clinical Terms (SNOMED-CT) [151], and National Drug Code (NDC) etc. These ontologies often have a hierarchical top-down structure, which systematically organize medical concepts into categories and subcategories.
Figure 2.4: Comparison between Gram and HAP. Gram only considers a node’s unordered ancestor set to compute its embedding. HAP hierarchically propagate information across the graph. In the bottom-up round, each parent aggregates information from its children. In the top-down round, each child aggregates information from its parents. The final embedding of each node effectively absorbs information from not only its ancestors, but the entire graph (ancestors, descendants, siblings and others).

of different levels from general to specific. Such hierarchical structures make identifying particular concepts and searching related concepts much easier. The contained structured domain knowledge can also potentially advance the power of healthcare prediction models. For example, nodes (medical concepts) close to each other in an ontology tend to be assigned to similar patients.

To incorporate the domain knowledge within an ontology into a machine learning algorithm, Gram, a graph-based attention model has been proposed recently [28]. Gram learns an embedding for each medical concept by adaptively summing its ancestors’ embeddings via an attention mechanism, so that the parent-child relations along a node’s paths to the root are encoded into the node’s embedding. Further, two leaf concepts sharing many common ancestors tend to have similar embeddings, which implicitly transfers knowledge between concepts and augments those medical concepts with few occurrences in the training data.

As a first attempt to learn from medical ontologies, Gram has several shortcomings. Firstly, Gram does not consider the order of a node’s ancestors – a node’s lower ancestors and higher ancestors are symmetrically treated in the attention mechanism, which loses the hierarchical information. Secondly, Gram only considers the ancestor information of a node.
It oversimplifies the ontology structure by ignoring the descendants and siblings of a node completely. Both of the shortcomings can be addressed by a more advanced graph attention model considering the full hierarchical structure.

In this paper, we propose the Hierarchical Attention Propagation (HAP) model. HAP does two rounds of knowledge propagation to learn embeddings of medical concepts from the entire ontology: first a bottom-up propagation from leaves to root, and second a top-down propagation from root to all leaves. In the bottom-up propagation, each node updates its embedding by adaptively combining its children’s embeddings using an attention mechanism. In the top-down propagation, nodes adaptively combine their parents’ embeddings using the same attention mechanism. Such a two-round propagation is inspired by the Belief Propagation algorithm [125] in graphical models, which is widely used to perform exact inference on tree/polytree models. The two-round knowledge propagation process in HAP effectively distributes a node’s attention across the graph, making a node’s final embedding no longer only a combination of its ancestors but aggregate information over the entire ontology. We prove that HAP is strictly more expressive than Gram, allowing any node embedding to reconstruct the complete ontology. Experimental results on two sequential procedure/diagnosis prediction tasks reveal HAP’s improved performance over Gram and other baselines.

2.2.2 Preliminaries

Notations

We use $c_1, c_2, \ldots, c_{|\mathcal{C}|} \in \mathcal{C}$ to denote the set of all leaf medical codes of a medical ontology $\mathcal{G}$, and use $c_{|\mathcal{C}|+1}, c_{|\mathcal{C}|+2}, \ldots, c_{|\mathcal{C}|+|\mathcal{C}'|} \in \mathcal{C}'$ to denote the non-leaf nodes (which are ancestors of the leaf nodes). The ontology $\mathcal{G}$ is expressed as a directed acyclic graph (DAG), where nodes
are hierarchically arranged in different levels, with the top level consisting of the single root node and the bottom level consisting of all the leaf nodes $\mathcal{C}$. Examples include the ICD-9, ICD-10 and CCS. We use knowledge DAG to refer to the ontology $\mathcal{G}$. In the knowledge DAG, a parent represents a related but more general concept over its children, such as the class of a disease or the category of a procedure. We use $A(i)$ to denote the set of $c_i$’s ancestors and $c_i$ itself. We use $P(i)$ and $C(i)$ to denote the parent set and children set of $c_i$ (both including $c_i$ itself), respectively.

We will consider sequential visit data from patients’ electronic health records (EHR) over time. The sequential visit data of a patient is denoted by $V_1, V_2, \ldots, V_T$, where each visit contains a subset of medical codes $V_t \subseteq \mathcal{C}$, indicating the procedures/diagnoses that the patient receives at the $t^{th}$ visit. $V_t$ can be represented as a binary vector $x_t \in \{0, 1\}^{|\mathcal{C}|}$, where the $i^{th}$ element is 1 if $c_i \in V_t$. For ease of presentation, we will propose our algorithms for a single patient in the rest of the paper. The sequential procedure/diagnosis prediction task is to predict the procedure/diagnosis codes $V_{t+1}$ given the past visits $V_1, V_2, \ldots, V_t$.

**Gram for medical ontology embedding**

To leverage the parent-child relationships of the ontology, Gram uses an attention mechanism to adaptively combine a node’s ancestors’ embeddings as its new embedding. More specifically, in the knowledge DAG, every node $c_i$ is first assigned an initial basic embedding $e_i$ (can be random embeddings or pretrained embeddings from other sources of information). Then, the final embedding $g_i$ for $c_i$ is given by a weighted sum of $\{e_j | j \in A(i)\}$:

$$g_i = \sum_{j \in A(i)} \alpha_{ij} e_j,$$  \hspace{1cm} (2.9)
where the weights are computed by the attention mechanism:

\[
\alpha_{ij} = \frac{\exp(f(e_i, e_j))}{\sum_{k \in A(i)} \exp(f(e_i, e_k))}.
\]  (2.10)

Here, \( f(e_i, e_j) \) is a multi-layer perceptron (MLP) which outputs a scalar value representing the raw attention between \( e_i \) and \( e_j \). The Softmax normalizes the attention weights so that they sum to 1.

The new embeddings are then used to represent the medical codes in the sequential visit data, which are fed to a RNN to train a sequential diagnosis prediction model in an end-to-end fashion. By leveraging the ontology, Gram has improved predictive performance, especially for predicting medical codes less observed in the training data. However, since only the ancestors of a node are considered, the domain knowledge within the ontology is not fully leveraged. Further, the ancestors from lower levels and higher levels are treated symmetrically in (2.9). Thus, the order of a node’s ancestors is completely ignored, which might lose important information about the hierarchy.

### 2.2.3 Methodology

In this paper, we propose Hierarchical Attention Propagation (HAP), a novel medical ontology embedding method which 1) fully leverages the knowledge DAG, and 2) respects the node ordering within the hierarchy. HAP does two rounds of knowledge propagation to iteratively update each level’s nodes’ embeddings: first a bottom-up propagation and second a top-down propagation. A comparison between HAP and Gram is illustrated in Figure A.1.

Suppose the ontology has \( L \) levels of nodes, where level 1 consists of the single root node and level \( L \) consists of only leaf medical codes. Level 2, 3, \ldots, \( L - 1 \) can contain either intermediate category nodes or leaf medical codes (because some medical codes do not have
a full $L$ levels of hierarchy). In the beginning, every node embedding $g_i$ is initialized using a basic embedding $e_i$. In the bottom-up propagation round, we sequentially update the embeddings of nodes from level $L - 1$, level $L - 2$, ..., until level 1. For node $c_i$ from level $l - 1$, we update its embedding by adaptively combining its current embedding with its children’s embeddings from level $l$ using an attention mechanism, given by:

$$g_i^{(l-1)} = \sum_{j \in C(i)} \alpha_{ij} g_j^{(l)}$$

(2.11)

where $g_j^{(l)} \in \mathbb{R}^{d_g}$ denotes the embedding of node $j$ before we start updating nodes from level $l - 1$. We use $d_g$ to denote the embedding size. The attention weight $\alpha_{ij}$ is given by:

$$\alpha_{ij} = \frac{\exp(f(g_i^{(l)}, g_j^{(l)}))}{\sum_{k \in C(i)} \exp(f(g_i^{(l)}, g_k^{(l)}))},$$

(2.12)

where $f(g_i^{(l)}, g_j^{(l)})$ is an MLP to compute the scalar raw attention between $g_i^{(l)}$ and $g_j^{(l)}$. In this work, we use a two layer neural network following [28]:

$$f(g_i^{(l)}, g_j^{(l)}) = u_a^\top \tanh(W_a \cdot \text{concat}(g_i^{(l)}, g_j^{(l)}) + b_a),$$

(2.13)

where $W_a \in \mathbb{R}^{d_a \times 2d_g}$ is the weight matrix for the column concatenation of $g_i^{(l)}$ and $g_j^{(l)}$, $b_a \in \mathbb{R}^{d_a}$ is the bias, and $u_a \in \mathbb{R}^{d_a}$ is the weight vector for generating the scalar raw attention.

Here, we use $d_a$ to denote the hidden size of $f$.

The bottom-up propagation starts from the second-to-last level, and goes all the way up to the root. The updating of nodes from the same level can be performed in parallel, while the updating of an upper level of nodes must wait until all its lower levels have been updated.
After the bottom-up propagation, HAP performs the second round of propagation in a top-down manner. Given the embeddings computed by the bottom-up propagation, we sequentially update the embeddings of nodes from level 2, level 3, ..., until level $L$. For node $c_i$ from level $l + 1$, we update its embedding using a similar attention mechanism by adaptively combining its own embedding with its parents’ embeddings from level $l$:

$$
\mathbf{g}_i^{(l+1)} = \sum_{j \in P(i)} \alpha_{ij} \mathbf{g}_j^{(l)},
$$

(2.14)

where $\mathbf{g}_j^{(l)}$ denotes the embedding of node $j$ before we start updating nodes from level $l + 1$. The attention weight $\alpha_{ij}$ is:

$$
\alpha_{ij} = \frac{\exp(f(\mathbf{g}_i^{(l)}, \mathbf{g}_j^{(l)}))}{\sum_{k \in P(i)} \exp(f(\mathbf{g}_i^{(l)}, \mathbf{g}_k^{(l)}))},
$$

(2.15)

where $f$ has the same form as in Equation (2.13), with a different set of parameters.

Finally, after the two rounds of propagation, each node has propagated its “attention” across the entire knowledge DAG. Thus, the final embedding of each node effectively absorbs knowledge from not only its ancestors, but also its descendants, siblings, and even some distant nodes. Furthermore, as the propagation order is strictly aligned with the hierarchy, the node ordering information is kept. For instance, in the top-down propagation phase, the ancestors of a node sequentially pass their information down level by level, rather than passing in one shot as in (2.9). This enables HAP to discriminate ancestors/descendants from different levels and encode the ordering information.

The final medical code embeddings are used in the sequential procedure/diagnosis prediction tasks. Following [28], we adopt an end-to-end RNN framework. The final embeddings $\mathbf{g}_1, \mathbf{g}_2, \ldots, \mathbf{g}_{|C|}$ are concatenated column-wise to form an embedding matrix $\mathbf{G} \in \mathbb{R}^{d_g \times |C|}$. 
Remember each visit record $V_t$ can be represented as a multi-hot vector $x_t$. To get an embedding vector $v_t$ for all medical codes in $V_t$, we multiply $G$ with $x_t$ and apply a nonlinear transformation by:

$$v_t = \text{tanh}(Gx_t).$$  \hspace{1cm} (2.16)

Then, we sequentially feed $v_1, v_2, \ldots, v_T$ into a RNN, which outputs the hidden state for each visit. The hidden state $h_t$ for $v_t$ is given by feeding the visit embeddings from all timestamps up to $t$:

$$h_t = \text{RNN}(v_1, v_2, \ldots, v_t).$$  \hspace{1cm} (2.17)

Then the prediction for the next timestamp $t + 1$ is given by:

$$\hat{y}_t = \hat{x}_{t+1} = \text{Softmax}(Wh_t + b),$$  \hspace{1cm} (2.18)

where $W \in \mathbb{R}^{|C| \times d_h}$ and $b \in \mathbb{R}^{d_h}$ are the weight and bias of the final prediction network, and $d_h$ is the dimension of the RNN’s hidden states. The prediction $\hat{y}_t$ is a vector of dimension $|C|$, indicating the probability of each medical code in visit $t + 1$. Note that following Gram, we use Softmax instead of dimension-wise sigmoid to predict multiple medical codes in the next visit as it showed better performance.

We use batch gradient descent to minimize the prediction loss of all timestamps (except timestamp 1). The prediction loss for a single patient is given by:

$$L(x_1, x_2, \ldots, x_T) = -\frac{1}{T-1} \sum_{t=1}^{T-1} \left[ x_{t+1}^\top \log(\hat{y}_t) \\
+ (1 - x_{t+1})^\top \log(1 - \hat{y}_t) \right].$$  \hspace{1cm} (2.19)
Following Gram, we not only train the model weights, but also train the basic embeddings $e_i$. The initialization of the basic embeddings follow the same procedure as in Gram, i.e., using the GloVe [126] embeddings learned from the cooccurrence matrix of augmented codes within the visit records. See section 2.4 of [28] for more details.

### 2.2.4 Theoretical analysis

In this subsection, we theoretically analyze the properties of our proposed Hierarchical Attention Propagation model, and show it has strictly higher expressive ability than Gram in terms of encoding knowledge DAGs.

Firstly, we use a counter example to show that recovering the knowledge DAG from Gram embeddings is not always possible due to its not encoding the order of the ancestors.

**Proposition 2.1.** With the embeddings $g_1, g_2, \ldots, g_{|C|}$ computed by Gram (Equation (2.9)), we **cannot** always perfectly reconstruct the knowledge DAG.

**Proof.** Consider two DAGs, $A \rightarrow B \rightarrow C$ and $B \rightarrow A \rightarrow C$. In both DAGs, the leaf node $C$ gets the same embedding using Equation (2.9). Thus, from $C$’s embedding we cannot differentiate which is the original DAG.

This counter-example reveals the limited expressive ability of Gram due to its ignorance of the node ordering and improper use of the hierarchy. It indicates that, after running Gram, the ontology information is not fully kept in the medical code embeddings. Thus, the Gram embeddings are essentially a lossy encoding of the knowledge DAG. Next, we study the expressive ability of HAP. In contrast to Gram, we show that from the HAP embeddings we can perfectly reconstruct the knowledge DAG.
Theorem 2.2. Assume the basic embeddings $e_i$ are unique identifiers of the medical concepts they represent. Then, from any embedding $g_i$ computed by HAP we can always perfectly reconstruct the knowledge DAG, given that every update in (2.11) and (2.14) is injective.

Proof. In the bottom-up propagation, since every update in (2.11) is injective, from the result $g_i^{(l-1)}$ of (2.11) we can reversely infer $g_i^{(l)}$ and $\{g_j^{(l)} \mid j \in C(i)\}$. Consider any node $i$ from level $L-1$. From its updated embedding $g_i^{(L-1)}$ we can always reconstruct the rooted sub-DAG formed by itself and its children. Now suppose every embedding $g_j^{(l)}$ from level $l$ injectively encodes the sub-DAG formed by $j$ and all of its descendants. Since Equation (2.11) is injective, any embedding $g_i^{(l-1)}$ from level $l-1$ also injectively encodes the sub-DAG formed by $i$ and all of its descendants. Applying structural induction, we get the conclusion that the root embedding at the end of the bottom-up propagation injectively encodes the entire knowledge DAG.

In the top-down propagation, since every update in (2.14) is also injective, any node embedding will injectively encode the sub-DAG formed by itself and all of its ancestors (including the root) represented by their new embeddings from the bottom-up propagation. Thus, after the top-down propagation, we can reconstruct the entire knowledge DAG from any node embedding (by recovering the root embedding).

The above theorem demonstrates that the medical code embeddings learned by HAP are strictly more expressive than those learned by Gram. It indicates that HAP embeddings can keep all the knowledge from the ontology – they are an lossless encoding of the knowledge DAG. Note that Theorem 2.2 requires the update functions in (2.11) and (2.14) to be injective. We now prove that the attention mechanism can be an injective mapping. First, we need the following lemma.
**Lemma 2.1.** Suppose $a, b, a', b' \in \mathbb{Z}^+$, $a \neq b$, $a' \neq b'$. Then $2^a - 2^b = 2^{a'} - 2^{b'}$ if and only if $a = a'$, $b = b'$.

*Proof.* Firstly, it is straightforward that $a = a', b = b' \Rightarrow 2^a - 2^b = 2^{a'} - 2^{b'}$. We now prove that $2^a - 2^b = 2^{a'} - 2^{b'}$ only if $a = a'$, $b = b'$.

Since $a \neq b$ and $a' \neq b'$, if $a > b$ and $a' < b'$ or $a < b$ and $a' > b'$, the equation $2^a - 2^b = 2^{a'} - 2^{b'}$ will not hold. Thus, w.l.o.g., we assume $a > b$ and $a' > b'$. We have

$$2^b(2^{a-b} - 1) = 2^{b'}(2^{a'-b'} - 1). \tag{2.20}$$

We will prove $a = a'$ and $b = b'$ by contradiction. Assume $b \neq b'$. Without loss of generality, we let $b > b'$. Then,

$$2^{b-b'}(2^{a-b} - 1) = (2^{a'-b'} - 1). \tag{2.21}$$

In the above equation, LHS is a product of an even number and an odd number which is even, while RHS is an odd number. Thus we have reached a contradiction, which means that $b = b'$. Eliminating $b, b'$ from $2^a - 2^b = 2^{a'} - 2^{b'}$, we have $a = a'$ too. \hfill \Box

Now we prove that the update functions in (2.11) and (2.14) can indeed be injective.

**Theorem 2.3.** There exists a function $f$ such that the update function

$$U(g_i, \{g_j \mid j \in S\}) := \sum_{j \in S \cup i} \alpha_{ij} g_j, \tag{2.22}$$

where $\alpha_{ij} = \frac{\exp(f(g_i, g_j))}{\sum_{k \in S \cup i} \exp(f(g_i, g_k))}, \tag{2.23}$

52
is injective w.r.t. its inputs \((g_i, \{g_j | j \in S\})\), where \(g_i\) and \(g_j\) are unique rational embeddings of the knowledge DAG nodes and \(S \neq \emptyset, i \notin S\).

Proof. Since an ontology has a limited number of concepts, \(g_i\) and \(g_j\) are from a countable universe. Thus, we can construct a function \(\phi\) such that \(\phi(i, j)\) maps every ordered \((g_i, g_j)\) pair to a unique positive integer. Assume there are \(n\) different \(g_i\) choices in the countable universe. Then, \(\phi(i, j)\) has \(n^2\) possible outputs.

Having \(\phi(i, j)\), we will construct \(f\) such that \(\alpha_{ij}\) is unique for every combination of \((g_i, g_j)\) and \(\{g_k | k \in S \cup i\}\). We let

\[
f(g_i, g_j) = 2^{\phi(i, j)}. \tag{2.24}\]

Then, the attention weight \(\alpha_{ij}\) becomes

\[
\alpha_{ij} = \frac{\exp(2^{\phi(i, j)})}{\sum_{k \in S \cup i} \exp(2^{\phi(i, k)})} = \frac{1}{1 + \sum_{k \in S \cup i, k \neq j} \exp(2^{\phi(i, k)} - 2^{\phi(i, j)})}. \tag{2.25}
\]

According to Lemma 2.1, \(2^{\phi(i, k)} - 2^{\phi(i, j)}\) is unique for each ordered tuple of \((i, j, k)\) when \(j \neq k\). Considering the linear independence among integer powers of \(e\), the summation in the denominator of the above equation constitutes a unique irrational representation for \((i, j, \{k | k \in S\})\). This means \(\alpha_{ij}\) is a unique irrational number for each different \((i, j, S)\) (the reciprocal of an irrational number is also irrational). Under fixed \(i\) and \(S\), a unique irrational \(\alpha_{ij}\) is associated with each \(g_j\). Besides, \(\alpha_{ij}\) for different \(j\) are linearly independent using rational coefficients (only multiplying \(\alpha_{ij}\) by integer powers of \(e\) can we recover other \(\alpha_{ij}\)). Thus, the summation \(\sum_{j \in S \cup i} \alpha_{ij} g_j\) is a unique representation for \((i, S, \{j \in S \cup i\})\), which
means it is a unique representation for \((i, S)\). Therefore, \(U(g_i, \{g_j | j \in S\})\) is an injective function.

To model such an \(f\), we use an MLP (2.13) with trainable weights thanks to the universal approximation theory [61]. Note that Theorem 2.3 requires the node embeddings \(g_i\) to be rational, which can be easily satisfied for the initial basic embeddings. For intermediate embeddings, the update functions in (2.11) and (2.14) will output irrational vectors. We can apply another MLP to the updated vectors to map them back to rational vectors. In practice, however, we find that the current scheme without another MLP already works well.

**2.2.5 Experiments**

**Experimental setup**

**Prediction tasks.** We use two datasets to evaluate the performance of HAP: 1) We conduct a sequential procedure prediction task using the ACTFAST dataset, which contains procedure codes of 14.9K patients who received surgery with anesthesia at Barnes-Jewish Hospital between June 2012 and August 2016. Given the history visit records of a patient’s ICD9 procedure codes, we aim to predict all the procedure codes she/he will receive in the next visit. 2) We conduct a sequential diagnosis prediction task using the open-source MIMIC-III dataset [70], which contains the medical records of 7.5K intensive care unit (ICU) patients over 11 years. Given the history of a patient’s ICD9 diagnosis codes in each visit, we aim to predict all the diagnosis codes she/he will receive in the next visit. A summary of the two datasets are provided in Table 2.4. Note that our setting is different from the setting of the Gram paper [28], the tasks of which were to predict CCS single-level groups of medical codes instead of the exact codes. This means that the number of possible target codes was much
smaller than ours. In other words, our tasks are more difficult and test a method’s precise prediction abilities. Our setting is also more natural and practical.

Table 2.4: Statistics of ACTFAST and MIMIC-III datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>ACTFAST</th>
<th>MIMIC-III</th>
</tr>
</thead>
<tbody>
<tr>
<td># of patients</td>
<td>14,878</td>
<td>7,499</td>
</tr>
<tr>
<td># of visits</td>
<td>39,803</td>
<td>19,911</td>
</tr>
<tr>
<td>Avg. # of visits per patient</td>
<td>2.68</td>
<td>2.66</td>
</tr>
<tr>
<td># of unique ICD9 codes</td>
<td>2,034</td>
<td>4,893</td>
</tr>
<tr>
<td>Avg. # of codes per visit</td>
<td>2.32</td>
<td>13.1</td>
</tr>
<tr>
<td>Max # of codes per visit</td>
<td>3</td>
<td>39</td>
</tr>
</tbody>
</table>

For both datasets, we filter out patients with less than two visits.

We calculate $\text{Accuracy}@k$ for each medical code. For each visit $V_t$, we get a 1 if the target medical code appears in the top $k$ predictions and 0 otherwise. Then, we report the overall $\text{Accuracy}@k$ for all medical codes as well as $\text{Accuracy}@k$ for grouped medical codes. To calculate grouped $\text{Accuracy}@k$, we first sort all possible target codes by their counts in the training data. Then, we divide all target codes into four groups $[0, 25], [25, 50], [50, 75], [75, 100]$ with each group’s codes having the same summed counts. That is, group $[0, 25]$ contains the the rarest medical codes which constitute 25% of the code counts in the training data. Group $[75, 100]$, on the other hand, contains the most frequent codes which in total constitute 25% of the training codes. For the ACTFAST data, we calculate $\text{Accuracy}@5$. For the MIMIC-III data, we calculate $\text{Accuracy}@20$ considering the large average number of ICD9 codes per visit.

We use the CCS multi-level procedure hierarchy\(^2\) ($L = 5$) as our knowledge DAG for the ACTFAST dataset, and use the CCS multi-level diagnosis hierarchy\(^3\) ($L = 6$) as our knowledge DAG for the MIMIC-III dataset.

\(^2\)https://www.hcup-us.ahrq.gov/toolssoftware/ccs/AppendixDMultiPR.txt
\(^3\)https://www.hcup-us.ahrq.gov/toolssoftware/ccs/AppendixCMultiDX.txt
We randomly split each dataset into the training, validation and test sets using 0.7:0.1:0.2 ratio. We train a model with the training set for 50 epochs, and use the model parameters at the epoch with the smallest validation loss to evaluate on the test set. We repeat each experiment for five times with different random seeds (thus using five different data splits in total). The average test accuracies and standard deviations are reported in the paper.

Models for comparison. We include the following models for comparison.

• HAP: The proposed Hierarchical Attention Propagation model. All levels of the hierarchy are used.

• HAP (lv3): The proposed HAP model using only the lowest three levels of the hierarchy. That is, the bottom-up propagation stops in level L-2, and the top-down propagation begins in level L-2 too. Using only lower levels of the hierarchy can sometimes provide sufficient domain knowledge while reducing the computation complexity.

• HAP (lv2): The proposed HAP model using only the lowest two levels of the hierarchy.

• Gram: The Graph-based Attention Model described in Preliminaries. A leaf code’s embedding \( g_i \) is a weighted sum of the basic embeddings of itself and its ancestors.

• Gram (lv3): The Gram model using only the lowest three levels of the hierarchy. A leaf code’s embedding \( g_i \) is a weighted sum of the basic embeddings of itself and its ancestors within the lowest three levels.

• Gram (lv2): The Gram model using only the lowest two levels of the hierarchy.

• RNN: A leaf embedding \( g_i \) takes its own basic embedding without considering the hierarchy. The basic embeddings are initialized using the GloVe embeddings learned from the cooccurrence matrix of leaf concepts, and are trained together with the RNN.
Table 2.5: Grouped and overall Accuracy@5 of sequential procedure prediction on ACTFAST data.

<table>
<thead>
<tr>
<th>Model</th>
<th>0-25</th>
<th>25-50</th>
<th>50-75</th>
<th>75-100</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>HAP</td>
<td>0.2036±0.0178</td>
<td>0.5064±0.0143</td>
<td>0.6284±0.0030</td>
<td>0.7325±0.0194</td>
<td>0.5169±0.0126</td>
</tr>
<tr>
<td>HAP (lv3)</td>
<td>0.1824±0.0107</td>
<td>0.4837±0.0107</td>
<td>0.6253±0.0026</td>
<td><strong>0.7380±0.0179</strong></td>
<td>0.5066±0.0094</td>
</tr>
<tr>
<td>HAP (lv2)</td>
<td><strong>0.2098±0.0120</strong></td>
<td><strong>0.5143±0.0079</strong></td>
<td><strong>0.6353±0.0112</strong></td>
<td>0.7359±0.0221</td>
<td><strong>0.5229±0.0123</strong></td>
</tr>
<tr>
<td>Gram</td>
<td>0.1553±0.0187</td>
<td>0.4619±0.0181</td>
<td>0.6299±0.0021</td>
<td>0.7326±0.0207</td>
<td>0.4921±0.0126</td>
</tr>
<tr>
<td>Gram (lv3)</td>
<td>0.0926±0.0120</td>
<td>0.3516±0.0167</td>
<td>0.5577±0.0065</td>
<td>0.7230±0.0148</td>
<td>0.4308±0.0116</td>
</tr>
<tr>
<td>Gram (lv2)</td>
<td>0.2081±0.0035</td>
<td>0.5113±0.0143</td>
<td>0.6323±0.0051</td>
<td>0.7357±0.0245</td>
<td>0.5210±0.0081</td>
</tr>
<tr>
<td>RNN</td>
<td>0.1967±0.0055</td>
<td>0.5124±0.0162</td>
<td>0.6336±0.0011</td>
<td>0.7365±0.0179</td>
<td>0.5189±0.0084</td>
</tr>
<tr>
<td>Random</td>
<td>0.1984±0.0137</td>
<td>0.5001±0.0113</td>
<td>0.6215±0.0057</td>
<td>0.7298±0.0160</td>
<td>0.5114±0.0105</td>
</tr>
<tr>
<td>Rollup</td>
<td>0.1020±0.0068</td>
<td>0.3880±0.0172</td>
<td>0.5646±0.0093</td>
<td>0.7158±0.0155</td>
<td>0.4419±0.0078</td>
</tr>
</tbody>
</table>

- Random: Each leaf concept is assigned a fixed random embedding $g_i$. This is to compare HAP with a naive baseline with no embedding learning.
- Rollup: Each leaf concept is assigned a fixed random embedding which is the same as its parent's embedding. This is to compare HAP with a common grouping scheme.

All models are implemented with Theano and optimized using Adadelta with a mini-batch size of 100 patients. We ensure the common hyperparameters are the same for all models: All models use the same GRU-based RNN with a hidden size $d_h = 400$ and a dropout rate of 0.4. The embedding size $d_g$ is 400 for all models. For HAP and Gram, the attention weights $W_a, b_a, u_a$ have a dimension $d_a = 100$.

**Prediction performance**

We present the sequential procedure prediction results on ACTFAST in Table 2.5, and present the sequential diagnosis prediction results on MIMIC-III in Table 2.6. In both tasks, HAP and its variants show advantages over other models. The gain is greater for less frequent codes. For example, in MIMIC-III, the HAP (lv3) is nearly twice more accurate than the Random baseline in the 0-25 percentile range, and is 3 percent more accurate than Random.
Table 2.6: Grouped and overall Accuracy@20 of sequential diagnosis prediction on MIMIC-III data.

<table>
<thead>
<tr>
<th>Model</th>
<th>0-25</th>
<th>25-50</th>
<th>50-75</th>
<th>75-100</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>HAP</td>
<td>0.0414±0.0062</td>
<td>0.2179±0.0103</td>
<td>0.3813±0.0119</td>
<td>0.7983±0.0171</td>
<td>0.3619±0.0027</td>
</tr>
<tr>
<td>HAP (lv3)</td>
<td>0.0434±0.0033</td>
<td>0.2119±0.0070</td>
<td>0.3884±0.0102</td>
<td>0.8006±0.0118</td>
<td>0.3633±0.0020</td>
</tr>
<tr>
<td>HAP (lv2)</td>
<td>0.0428±0.0062</td>
<td>0.2168±0.0091</td>
<td>0.3905±0.0128</td>
<td>0.7970±0.0128</td>
<td>0.3640±0.0027</td>
</tr>
<tr>
<td>Gram</td>
<td>0.0426±0.0055</td>
<td>0.2042±0.0081</td>
<td>0.3733±0.0118</td>
<td>0.8084±0.0045</td>
<td>0.3591±0.0044</td>
</tr>
<tr>
<td>Gram (lv3)</td>
<td>0.0417±0.0058</td>
<td>0.2127±0.0130</td>
<td>0.3858±0.0148</td>
<td>0.7965±0.0167</td>
<td>0.3614±0.0029</td>
</tr>
<tr>
<td>Gram (lv2)</td>
<td>0.0399±0.0058</td>
<td>0.2082±0.0124</td>
<td>0.3838±0.0188</td>
<td>0.8031±0.0116</td>
<td>0.3609±0.0031</td>
</tr>
<tr>
<td>RNN</td>
<td>0.0399±0.0064</td>
<td>0.2167±0.0147</td>
<td>0.3806±0.0112</td>
<td>0.8044±0.0160</td>
<td>0.3625±0.0028</td>
</tr>
<tr>
<td>Random</td>
<td>0.0244±0.0018</td>
<td>0.1828±0.0121</td>
<td>0.3745±0.0096</td>
<td>0.8097±0.0120</td>
<td>0.3500±0.0031</td>
</tr>
<tr>
<td>Rollup</td>
<td>0.0304±0.0040</td>
<td>0.1804±0.0067</td>
<td>0.3659±0.0163</td>
<td>0.8013±0.0014</td>
<td>0.3466±0.0028</td>
</tr>
</tbody>
</table>

in the 25-50 percentile range, indicating that learning embeddings from ontology structures benefits the prediction of rare codes.

Both HAP and Gram leverage the ontology structure, yet Gram seems to be much more sensitive to the number of hierarchy levels used. By varying the number of hierarchy levels, Gram shows much larger performance variances. Especially for ACTFAST, we observe that Gram (lv3) has a large drop of accuracies compared to Gram (lv2). This indicates that Gram is not robust to the number of hierarchy levels used and requires carefully selecting this number. In contrast, HAP shows robust performance consistently. One possible explanation is that the use of the level $L-2$ hierarchy makes a Gram’s leaf embedding confused about its parent and grandparent (since the attention scheme in Gram is order-unaware). Thus, adding this additional hierarchy level on the contrary loses some performance. On the other hand, HAP is order-aware. It always first does bottom-up propagation for level $L-1$ and then for level $L-2$, respecting the ordering within the hierarchy. With the further increase of the number of hierarchy levels used, Gram’s performance recovers a little as the adding of more hierarchical information compensates for the loss of ordering.

We also find that HAP and Gram do not perform the best using the full hierarchy. This indicates that lower levels of the hierarchy provide the most important domain knowledge for
predicting leaf codes, which is reasonable since the higher levels correspond to very broad categories of procedures or diseases that might not be as useful as those fine categories in lower levels. Compared to RNN, Gram and its variants show some advantages in predicting rare medical codes, often at the cost of slight drops in overall accuracies. In comparison, HAP and its variants have gains for rare medical codes too, but do not sacrifice the overall performance by showing the highest overall accuracies in both tasks.

Visualization

We qualitatively evaluate HAP by visualizing the embeddings learned by HAP from MIMIC-III. We choose all 1,120 level-6 leaf codes and show their t-SNE [107] embeddings in a 2-D

Figure 2.5: t-SNE scatterplots of diagnosis code embeddings learned by HAP.
space (Figure 2.5). We color all dots by their level-5 categories (if two dots belong to the same level-5 category, they will have the same color). Then, we manually label some clear clusters in the 2-D space with their level-5 category descriptions. For comparison, we also show the t-SNE visualizations of the embeddings learned by Gram (Figure 2.6 left) and RNN (Figure 2.6 right). The t-SNE hyperparameters are the same for all models. As we can see, both HAP and Gram give more interpretable embeddings than RNN by demonstrating more structured code distributions. This verifies that the learned embeddings well incorporate the domain knowledge from the ontology structure. In contrast, RNN’s code distribution seems completely random without showing any clear patterns.

2.2.6 Related work

Attention mechanism is a widely used framework in neural networks to adaptively learn the importance of each component w.r.t. the target component. It has been successfully used in
computer vision [167], machine translation [157], speech recognition [31], semi-supervised node classification [158] etc. There has been work that applies attention mechanism to healthcare problems [28, 29]. Our work is closely related to the Gram model [28] by generalizing the attention mechanism used to aggregate ancestor information to a hierarchical multi-level propagation framework that learns from the entire DAG.

Our method is related to recent graph representation learning works such as network embedding [56, 127, 155] and graph neural networks (GNNs) [18, 39, 77, 94, 136]. Our work is more related to GNNs as both of them are supervised. GNNs iteratively pass messages between each node and its neighbors to extract local substructure features around nodes. The learnable parameters in the message passing layers equip GNNs with excellent node/graph representation learning abilities and great flexibility. GNNs have gained great popularity in recent years, achieving state-of-the-art performance on semi-supervised node classification [77], graph classification [184], network embedding [57], link prediction [181] etc. Despite the success, little work has been done to apply GNNs to healthcare. Our model can be seen as combining an attention-based graph convolution layer [158] with a particular message passing order respecting the multi-level hierarchy of the medical ontology, where the design is inspired by the Belief Propagation algorithm and ensures incorporating structural information of the entire knowledge DAG. There is also recent work studying GNNs for trees/DAGs [69, 185] with a focus on generating DAGs instead of learning embeddings of DAG nodes.

2.2.7 Conclusion

In this section, we have proposed Hierarchical Attention Propagation (HAP), a graph attention-based method to learn medical concept embeddings based on medical ontologies. HAP learns highly expressive embeddings by learning from the full ontology hierarchy, addressing previous work Gram’s limited expressibility. We have theoretically proved that from any embedding
learned by HAP we can recover the entire knowledge DAG. Experiments on one sequential procedure prediction task on ACTFAST and one sequential diagnosis prediction task on MIMIC-III verified the superior medical concept embedding ability of HAP.
Chapter 3

Graph Neural Networks for Relation Prediction

In this chapter, we study GNNs for relation prediction. We focus on two problems, link prediction and recommender systems. Link prediction refers to predicting whether two nodes in a network are likely to have a link. Example applications include predicting friendship relations in social networks and predicting protein interactions in biological networks. Recommender systems aims to recommend items to users, which can also be cast as a link prediction problem if we treat user-item interactions as a bipartite graph. Because the rating a user gives to an item is often not simply 1 or 0, recommender systems is actually predicting real values on the links instead of predicting existence of links as in link prediction.

For link prediction, we first thoroughly study traditional link prediction heuristics, such as common neighbors [96] and Katz index [74], and develop a $\gamma$-decaying theory that unifies existing heuristics into a single framework. Our theory not only shows that a wide range of heuristics share the same intrinsic $\gamma$-decaying form, but also demonstrates
their approximability from local subgraphs around links. This motivates us to use graph neural networks to automatically learn suitable heuristics for given networks instead of using predefined ones, the framework of which we call SEAL [181]. Experiments show that our SEAL framework achieves unprecedented strong performance for link prediction, outperforming all existing representative methods.

For recommender systems, we formulate the matrix completion problem as a labeled link prediction problem in bipartite graphs, where graph edges can have multiple types (correspond to different ratings), and also apply GNNs to learn structural features from local subgraphs. Our model achieves state-of-the-art performance on benchmark matrix completion datasets. Compared to traditional matrix completion baselines, we further show that our method is inductive and transferrable. That is, our model can be applied to user-item pairs unseen during the training, and can even be transferred to completely new tasks. Experiments show that our model trained on MovieLens can be directly used to predict the movie ratings of Douban, and even achieve better performance than some baselines trained exclusively on Douban.

3.1 Link Prediction Based on Graph Neural Networks

3.1.1 A brief review of link prediction methods

Link prediction is to predict whether two nodes in a network are likely to have a link [96], which is a key problem for network structured data. It has many applications, such as friend recommendation in social networks [2], product recommendation in e-commerce [82], knowledge graph completion [120], finding interactions between proteins [5], and recovering missing reactions in metabolic networks [124, 183]. One class of simple yet effective approaches
for link prediction is called heuristic methods. Heuristic methods compute some heuristic node similarity scores as the likelihood of links [96, 102]. Existing heuristics can be categorized based on the maximum hop of neighbors needed to calculate the score. For example, common neighbors (CN) and preferential attachment (PA) [9] are first-order heuristics, since they only involve the one-hop neighbors of two target nodes. Adamic-Adar (AA) and resource allocation (RA) [188] are second-order heuristics, as they are calculated from up to two-hop neighborhood of the target nodes. We define $h$-order heuristics to be those heuristics which require knowing up to $h$-hop neighborhood of the target nodes. There are also some high-order heuristics which require knowing the entire network. Examples include Katz, rooted PageRank (PR) [17], and SimRank (SR) [66]. Table 3.1 summarizes eight popular heuristics.

Table 3.1: Popular heuristics for link prediction, see [96] for details.

<table>
<thead>
<tr>
<th>Name</th>
<th>Formula</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>common neighbors</td>
<td>$</td>
<td>\Gamma(x) \cap \Gamma(y)</td>
</tr>
<tr>
<td>Jaccard</td>
<td>$\frac{</td>
<td>\Gamma(x) \cap \Gamma(y)</td>
</tr>
<tr>
<td>preferential attachment</td>
<td>$</td>
<td>\Gamma(x)</td>
</tr>
<tr>
<td>Adamic-Adar</td>
<td>$\sum_{z \in \Gamma(x) \cap \Gamma(y)} \frac{1}{\log</td>
<td>\Gamma(z)</td>
</tr>
<tr>
<td>resource allocation</td>
<td>$\sum_{z \in \Gamma(x) \cap \Gamma(y)} \frac{1}{</td>
<td>\Gamma(z)</td>
</tr>
<tr>
<td>Katz</td>
<td>$\sum_{l=1}^{\infty} \beta^l</td>
<td>\text{walks}^{(l)}(x,y)</td>
</tr>
<tr>
<td>PageRank</td>
<td>$[\pi_x]_y + [\pi_y]_x$</td>
<td>high</td>
</tr>
<tr>
<td>SimRank</td>
<td>$\gamma \sum_{a \in \Gamma(x)} \sum_{b \in \Gamma(y)} \frac{\text{score}(a,b)}{</td>
<td>\Gamma(x)</td>
</tr>
</tbody>
</table>

Notes: $\Gamma(x)$ denotes the neighbor set of vertex $x$. $\beta < 1$ is a damping factor. $|\text{walks}^{(l)}(x,y)|$ counts the number of length-$l$ walks between $x$ and $y$. $[\pi_x]_y$ is the stationary distribution probability of $y$ under the random walk from $x$ with restart, see [17]. SimRank score is a recursive definition. We exclude those heuristics which are simple variants of the above or are proven to be meaningless for large graphs (e.g., commute time [104]).
In fact, the heuristics belong to a more generic class, namely graph structure features. Graph structure features are those features located inside the observed node and edge structures of the network, which can be calculated directly from the graph. Besides graph structure features, latent features and explicit features are also studied for link prediction. Latent feature methods [5, 56, 82, 127] factorize some matrix representations of the network to learn a low-dimensional latent representation/embedding for each node. Examples include matrix factorization [82] and stochastic block model [5] etc. Recently, a number of network embedding techniques have been proposed, such as DeepWalk [127], LINE [155] and node2vec [56], which are also latent feature methods since they implicitly factorize some matrices too [129]. Compared to graph structure features, latent features cannot be directly seen from the graph (requires training), and sometimes need an extremely large dimension to express some simple heuristics [119]. Explicit features are often available in the form of node attributes, describing all kinds of side information about individual nodes. It is shown that combining graph structure features with latent features and explicit features can improve the performance [119, 186].

3.1.2 Limitations of existing methods

Among the three types of link prediction features, heuristic methods have the best scalability and are widely used in practice. Although working well in practice, heuristic methods have strong assumptions on when links may exist. For example, the common neighbor heuristic assumes that two nodes are more likely to connect if they have many common neighbors. This assumption may be correct in social networks, but is shown to fail in protein-protein interaction (PPI) networks – two proteins sharing many common neighbors are actually less likely to interact [83]. On the other hand, rooted PageRank [17] has exceptional performance
on predicting biological networks, but shows worse performance on power grids and router-level Internets [102]. This suggests a significant limitation of heuristic methods – they lack universal applicability to different kinds of networks. A survey paper compared over 20 different heuristics and found that none of them performs consistently well across all networks [102]. This implies the need to manually choose different heuristics for different networks based on prior beliefs or expensive trial and error.

Given that heuristics can be viewed as predefined graph structure features, can we automatically learn such features from the network? The answer is yes, and we will introduce how we apply GNNs to learn general graph structure features for link prediction in the following. The learned features are based on existing node connection patterns of the network, thus are network-specific, which avoids the need to manually select predefined heuristics.

Another limitation of existing methods is that there are no principled way to unify all three types of features for link prediction. Most existing combination methods simply add several heuristic terms in the optimization objective of latent feature methods – although improving the performance, the graph structure features are still restricted to fixed forms [119, 186]. To address this limitation, we propose SEAL (learning from Subgraphs, Embeddings and Attributes for Link prediction), which is able to jointly learn from all three types of features (graph structure features, latent features and explicit features) for link prediction based on a graph neural network.

### 3.1.3 A theory for unifying link prediction heuristics

In this subsection, we aim to understand deeper the mechanisms behind various link prediction heuristics, and thus motivating the idea of learning heuristics from local subgraphs. Due to the large number of graph learning techniques, note that we are not concerned with the
Figure 3.1: The SEAL framework. For each target link, SEAL extracts a local enclosing subgraph around it, and uses a GNN to learn general graph structure features for link prediction. Note that the heuristics listed inside the box are just for illustration – the learned features may be completely different from existing heuristics.

generalization error of a particular method, but focus on the information reserved in the subgraphs for calculating existing heuristics.

**Definition 3.1. (Enclosing subgraph)** For a graph $G = (V, E)$, given two nodes $x, y \in V$, the $h$-hop enclosing subgraph for $(x, y)$ is the subgraph $G^h_{x,y}$ induced from $G$ by the set of nodes $\{ i \mid d(i, x) \leq h \text{ or } d(i, y) \leq h \}$.

The enclosing subgraph describes the “$h$-hop surrounding environment” of $(x, y)$. Since $G^h_{x,y}$ contains all $h$-hop neighbors of $x$ and $y$, we naturally have the following theorem.

**Theorem 3.1.** Any $h$-order heuristic for $(x, y)$ can be accurately calculated from $G^h_{x,y}$.

For example, a 2-hop enclosing subgraph will contain all the information needed to calculate any first and second-order heuristics. However, although first and second-order heuristics are well covered by local enclosing subgraphs, an extremely large $h$ seems to be still needed for learning high-order heuristics. Surprisingly, our following analysis shows that learning high-order heuristics is also feasible with a small $h$. We support this first by defining the $\gamma$-decaying heuristic. We will show that under certain conditions, a $\gamma$-decaying heuristic...
can be very well approximated from the $h$-hop enclosing subgraph. Moreover, we will show that almost all well-known high-order heuristics can be unified into this $\gamma$-decaying heuristic framework.

**Definition 3.2. ($\gamma$-decaying heuristic)** A $\gamma$-decaying heuristic for $(x, y)$ has the following form:

$$\mathcal{H}(x, y) = \eta \sum_{l=1}^{\infty} \gamma^l f(x, y, l),$$

(3.1)

where $\gamma$ is a decaying factor between 0 and 1, $\eta$ is a positive constant or a positive function of $\gamma$ that is upper bounded by a constant, $f$ is a nonnegative function of $x, y, l$ under the given network.

Next, we will show that under certain conditions, a $\gamma$-decaying heuristic can be approximated from an $h$-hop enclosing subgraph, and the approximation error decreases at least exponentially with $h$.

**Theorem 3.2.** Given a $\gamma$-decaying heuristic $\mathcal{H}(x, y) = \eta \sum_{l=1}^{\infty} \gamma^l f(x, y, l)$, if $f(x, y, l)$ satisfies:

- (property 1) $f(x, y, l) \leq \lambda^l$ where $\lambda < \frac{1}{\gamma}$; and
- (property 2) $f(x, y, l)$ is calculable from $G_{x,y}^l$ for $l = 1, 2, \cdots, g(h)$, where $g(h) = ah + b$ with $a, b \in \mathbb{N}$ and $a > 0$,

then $\mathcal{H}(x, y)$ can be approximated from $G_{x,y}^h$ and the approximation error decreases at least exponentially with $h$.

**Proof.** We can approximate such a $\gamma$-decaying heuristic by summing over its first $g(h)$ terms.

$$\tilde{\mathcal{H}}(x, y) := \eta \sum_{l=1}^{g(h)} \gamma^l f(x, y, l).$$

(3.2)
The approximation error can be bounded as follows.

\[ |\mathcal{H}(x, y) - \widetilde{\mathcal{H}}(x, y)| = \eta \sum_{l=g(h)+1}^{\infty} \gamma^l f(x, y, l) \leq \eta \sum_{l=a h+b+1}^{\infty} \gamma^l \lambda^l = \eta (\gamma \lambda)^{a h+b+1}(1 - \gamma \lambda)^{-1}. \]

In practice, a small \( \gamma \lambda \) and a large \( a \) lead to a faster decreasing speed. Next we will prove that three popular high-order heuristics: Katz, rooted PageRank and SimRank, are all \( \gamma \)-decaying heuristics which satisfy the properties in Theorem 3.2. First, we need the following lemma.

**Lemma 3.1.** Any walk between \( x \) and \( y \) with length \( l \leq 2 h + 1 \) is included in \( G_{x,y}^h \).

**Proof.** Given any walk \( w = (x, v_1, \cdots, v_{l-1}, y) \) with length \( l \), we will show that every node \( v_i \) is included in \( G_{x,y}^h \). Consider any \( v_i \). Assume \( d(v_i, x) \geq h + 1 \) and \( d(v_i, y) \geq h + 1 \). Then, \( 2 h + 1 \geq l = |\langle x, v_1, \cdots, v_i \rangle| + |\langle v_i, \cdots, v_{l-1}, y \rangle| \geq d(v_i, x) + d(v_i, y) \geq 2 h + 2 \), a contradiction. Thus, \( d(v_i, x) \leq h \) or \( d(v_i, y) \leq h \). By the definition of \( G_{x,y}^h \), \( v_i \) must be included in \( G_{x,y}^h \). \( \square \)

Next we will analyze Katz, rooted PageRank and SimRank one by one.

**Katz index**

The Katz index [74] for \((x, y)\) is defined as

\[ \text{Katz}_{x,y} = \sum_{l=1}^{\infty} \beta^l |\text{walks}^{(l)}(x, y)| = \sum_{l=1}^{\infty} \beta^l [A^l]_{x,y}, \]  

(3.3)

where \( \text{walks}^{(l)}(x, y) \) is the set of length-\( l \) walks between \( x \) and \( y \), and \( A^l \) is the \( l \)th power of the adjacency matrix of the network. Katz index sums over the collection of all walks between \( x \) and \( y \) where a walk of length \( l \) is damped by \( \beta^l \) (\( 0 < \beta < 1 \)), giving more weight to shorter walks.
Katz index is directly defined in the form of a $\gamma$-decaying heuristic with $\eta = 1, \gamma = \beta$, and $f(x, y, l) = |\text{walks}^{(l)}(x, y)|$. According to Lemma 3.1, $|\text{walks}^{(l)}(x, y)|$ is calculable from $G^h_{x,y}$ for $l \leq 2h + 1$, thus property 2 in Theorem 3.2 is satisfied. Now we show when property 1 is satisfied.

**Proposition 3.1.** For any nodes $i, j$, $[A^l]_{i,j}$ is bounded by $d^l$, where $d$ is the maximum node degree of the network.

**Proof.** We prove it by induction. When $l = 1$, $A_{i,j} \leq d$ for any $(i, j)$. Thus the base case is correct. Now, assume by induction that $[A^l]_{i,j} \leq d^l$ for any $(i, j)$, we have

$$[A^{l+1}]_{i,j} = \sum_{k=1}^{V} [A^l]_{i,k} A_{k,j} \leq d^l \sum_{k=1}^{V} A_{k,j} \leq d d^l = d^{l+1}.$$  

Taking $\lambda = d$, we can see that whenever $d < 1/\beta$, the Katz index will satisfy property 1 in Theorem 3.2. In practice, the damping factor $\beta$ is often set to very small values like $5E-4$ [96], which implies that Katz can be very well approximated from the $h$-hop enclosing subgraph.

**PageRank**

The rooted PageRank for node $x$ calculates the stationary distribution of a random walker starting at $x$, who iteratively moves to a random neighbor of its current position with probability $\alpha$ or returns to $x$ with probability $1 - \alpha$. Let $\pi_x$ denote the stationary distribution vector. Let $[\pi_x]_i$ denote the probability that the random walker is at node $i$ under the stationary distribution.

Let $P$ be the transition matrix with $P_{i,j} = \frac{1}{|\Gamma(v_j)|}$ if $(i, j) \in E$ and $P_{i,j} = 0$ otherwise. Let $e_x$ be a vector with the $x^{th}$ element being 1 and others being 0. The stationary distribution
\[ \pi_x = \alpha P \pi_x + (1 - \alpha)e_x. \] (3.4)

When used for link prediction, the score for \((x, y)\) is given by \([\pi_x]_y\) (or \([\pi_y]_x\) for symmetry). To show that rooted PageRank is a \(\gamma\)-decaying heuristic, we introduce the inverse \(P\)-distance theory [65], which states that \([\pi_x]_y\) can be equivalently written as follows:

\[ [\pi_x]_y = (1 - \alpha) \sum_{w: x \rightarrow y} P[w] \alpha^{|\text{len}(w)|}, \] (3.5)

where the summation is taken over all walks \(w\) starting at \(x\) and ending at \(y\) (possibly touching \(x\) and \(y\) multiple times). For a walk \(w = \langle v_0, v_1, \ldots, v_k \rangle\), \(\text{len}(w) := |\langle v_0, v_1, \ldots, v_k \rangle|\) is the length of the walk. The term \(P[w]\) is defined as \(\prod_{i=0}^{k-1} \frac{1}{|\Gamma(v_i)|}\), which can be interpreted as the probability of traveling \(w\). Now we have the following theorem.

**Theorem 3.3.** The rooted PageRank heuristic is a \(\gamma\)-decaying heuristic which satisfies the properties in Theorem 3.2.

**Proof.** We first write \([\pi_x]_y\) in the following form.

\[ [\pi_x]_y = (1 - \alpha) \sum_{l=1}^{\infty} \sum_{\substack{w: x \rightarrow y \\text{len}(w)=l}} P[w] \alpha^l. \] (3.6)

Defining \(f(x, y, l) := \sum_{\substack{w: x \rightarrow y \\text{len}(w)=l}} P[w]\) leads to the form of a \(\gamma\)-decaying heuristic. Note that \(f(x, y, l)\) is the probability that a random walker starting at \(x\) stops at \(y\) with exactly \(l\) steps, which satisfies \(\sum_{z \in V} f(x, z, l) = 1\). Thus, \(f(x, y, l) \leq 1 \leq \frac{1}{\alpha}\) (property 1). According to Lemma 3.1, \(f(x, y, l)\) is also calculable from \(G^h_{x, y}\) for \(l \leq 2h + 1\) (property 2). \(\square\)
SimRank

The SimRank score [66] is motivated by the intuition that two nodes are similar if their neighbors are also similar. It is defined in the following recursive way: if \( x = y \), then 
\[
s(x, y) := 1
\]
otherwise,
\[
s(x, y) := \gamma \sum_{a \in \Gamma(x)} \sum_{b \in \Gamma(y)} s(a, b) \frac{1}{|\Gamma(x)| \cdot |\Gamma(y)|}
\]
where \( \gamma \) is a constant between 0 and 1. According to [66], SimRank has an equivalent definition:
\[
s(x, y) = \sum_{w: (x, y) \rightarrow (z, z)} P[w] \gamma^{|\text{len}(w)|},
\]
where \( w : (x, y) \rightarrow (z, z) \) denotes all simultaneous walks such that one walk starts at \( x \), the other walk starts at \( y \), and they first meet at any vertex \( z \). For a simultaneous walk \( w = \langle (v_0, u_0), \ldots, (v_k, u_k) \rangle \), \( |\text{len}(w)| = k \) is the length of the walk. The term \( P[w] \) is similarly defined as \( \prod_{i=0}^{k-1} \frac{1}{|\Gamma(v_i)| \cdot |\Gamma(u_i)|} \), describing the probability of this walk. Now we have the following theorem.

**Theorem 3.4.** SimRank is a \( \gamma \)-decaying heuristic which satisfies the properties in Theorem 3.2.

**Proof.** We write \( s(x, y) \) as follows.
\[
s(x, y) = \sum_{l=1}^{\infty} \sum_{\substack{w: (x, y) \rightarrow (z, z) \\text{len}(w)=l}} P[w] \gamma^l,
\]
Defining \( f(x, y, l) := \sum_{\substack{w: (x, y) \rightarrow (z, z) \\text{len}(w)=l}} P[w] \) reveals that SimRank is a \( \gamma \)-decaying heuristic. Note that \( f(x, y, l) \leq 1 < \frac{1}{\gamma} \). It is easy to see that \( f(x, y, l) \) is also calculable from \( G^h_{x,y} \) for \( l \leq h \). \( \square \)
Discussion. There exist several other high-order heuristics based on path counting or random walk [102] which can be as well incorporated into the $\gamma$-decaying heuristic framework. We omit the analysis here. Our results reveal that most high-order heuristics inherently share the same $\gamma$-decaying heuristic form, and thus can be effectively approximated from an $h$-hop enclosing subgraph with exponentially smaller approximation error. We believe the ubiquity of $\gamma$-decaying heuristics is not by accident – it implies that a successful link prediction heuristic is better to put exponentially smaller weight on structures far away from the target, as remote parts of the network intuitively make little contribution to link existence. Our results build the foundation for learning heuristics from local subgraphs, as they imply that local enclosing subgraphs already contain enough information to learn good graph structure features for link prediction which is much desired considering learning from the entire network is often infeasible. To summarize, from the small enclosing subgraphs extracted around links, we are able to accurately calculate first and second-order heuristics, and approximate a wide range of high-order heuristics with small errors. Therefore, given adequate feature learning ability of the model used, learning from such enclosing subgraphs is expected to achieve performance at least as good as a wide range of heuristics. There is some related work which empirically verifies that local methods can often estimate PageRank and SimRank well [23, 67]. Another related theoretical work [10] establishes a condition of $h$ to achieve some fixed approximation error for ordinary PageRank.

3.1.4 SEAL: An implementation of the theory using GNN

In this subsection, we describe our SEAL framework for link prediction. SEAL does not restrict the learned features to be in some particular forms such as $\gamma$-decaying heuristics, but instead learns general graph structure features for link prediction. It contains three steps: 1) enclosing subgraph extraction, 2) node information matrix construction, and 3) GNN learning.
Given a network, we aim to learn automatically a “heuristic” that best explains the link formations. Motivated by the theoretical results, this function takes local enclosing subgraphs around links as input, and output how likely the links exist. To learn such a function, we train a graph neural network (GNN) over the enclosing subgraphs. Thus, the first step in SEAL is to extract enclosing subgraphs for a set of sampled positive links (observed) and a set of sampled negative links (unobserved) to construct the training data.

A GNN typically takes \((A, X)\) as input, where \(A\) (with slight abuse of notation) is the adjacency matrix of the input enclosing subgraph, \(X\) is the node information matrix each row of which corresponds to a node’s feature vector. The second step in SEAL is to construct the node information matrix \(X\) for each enclosing subgraph. This step is crucial for training a successful GNN link prediction model. In the following, we discuss this key step. The node information matrix \(X\) in SEAL has three components: structural node labels, node embeddings and node attributes.

Node labeling

The first component in \(X\) is each node’s structural label. A node labeling is function \(f_l : V \rightarrow \mathbb{N}\) which assigns an integer label \(f_l(i)\) to every node \(i\) in the enclosing subgraph. The purpose is to use different labels to mark nodes’ different roles in an enclosing subgraph: 1) The center nodes \(x\) and \(y\) are the target nodes between which the link is located. 2) Nodes with different relative positions to the center have different structural importance to the link. A proper node labeling should mark such differences. If we do not mark such differences, GNNs will not be able to tell where are the target nodes between which a link existence should be predicted, and lose structural information.
Our node labeling method is derived from the following criteria: 1) The two target nodes $x$ and $y$ always have the distinctive label “1”. 2) Nodes $i$ and $j$ have the same label if $d(i, x) = d(j, x)$ and $d(i, y) = d(j, y)$. The second criterion is because, intuitively, a node $i$’s topological position within an enclosing subgraph can be described by its radius with respect to the two center nodes, namely $(d(i, x), d(i, y))$. Thus, we let nodes on the same orbit have the same label, so that the node labels can reflect nodes’ relative positions and structural importance within subgraphs.

Based on the above criteria, we propose a Double-Radius Node Labeling (DRNL) as follows. First, assign label 1 to $x$ and $y$. Then, for any node $i$ with $(d(i, x), d(i, y)) = (1, 1)$, assign label $f_i(i) = 2$. Nodes with radius $(1, 2)$ or $(2, 1)$ get label 3. Nodes with radius $(1, 3)$ or $(3, 1)$ get 4. Nodes with $(2, 2)$ get 5. Nodes with $(1, 4)$ or $(4, 1)$ get 6. Nodes with $(2, 3)$ or $(3, 2)$ get 7. So on and so forth. In other words, we iteratively assign larger labels to nodes with a larger radius w.r.t. both center nodes, where the label $f_i(i)$ and the double-radius $(d(i, x), d(i, y))$ satisfy

1) if $d(i, x) + d(i, y) \neq d(j, x) + d(j, y)$, then $d(i, x) + d(i, y) < d(j, x) + d(j, y) \Leftrightarrow f_i(i) < f_i(j)$;

2) if $d(i, x) + d(i, y) = d(j, x) + d(j, y)$, then $d(i, x)d(i, y) < d(j, x)d(j, y) \Leftrightarrow f_i(i) < f_i(j)$.

One advantage of DRNL is that it has a perfect hashing function

$$f_i(i) = 1 + \min(d_x, d_y) + (d/2)[(d/2) + (d \% 2) - 1],$$

(3.10)

where $d_x := d(i, x)$, $d_y := d(i, y)$, $d := d_x + d_y$, $(d/2)$ and $(d \% 2)$ are the integer quotient and remainder of $d$ divided by 2, respectively. This perfect hashing allows fast closed-form computations.
For nodes with \( d(i, x) = \infty \) or \( d(i, y) = \infty \), we give them a null label 0. Note that DRNL is not the only possible way of node labeling, but we empirically verified its better performance than no labeling and other naive labelings. After getting the labels, we use their one-hot encoding vectors to construct \( X \).

**Incorporating latent and explicit features**

Other than the structural node labels, the node information matrix \( X \) also provides an opportunity to include latent and explicit features. By concatenating each node’s embedding/attribute vector to its corresponding row in \( X \), we can make SEAL simultaneously learn from all three types of features.

Generating the node embeddings for SEAL is nontrivial. Suppose we are given the observed network \( G = (V, E) \), a set of sampled positive training links \( E_p \subseteq E \), and a set of sampled negative training links \( E_n \) with \( E_n \cap E = \emptyset \). If we directly generate node embeddings on \( G \), the node embeddings will record the link existence information of the training links (since \( E_p \subseteq E \)). We observed that GNNs can quickly find out such link existence information and optimize by only fitting this part of information. This results in bad generalization performance in our experiments. Our trick is to temporally add \( E_n \) into \( E \), and generate the embeddings on \( G' = (V, E \cup E_n) \). This way, the positive and negative training links will have the same link existence information recorded in the embeddings, so that GNN cannot classify links by only fitting this part of information. We empirically verified the much improved performance of this trick to SEAL. We name this trick *negative injection*.

We name our proposed framework **SEAL** (learning from Subgraphs, Embeddings and Attributes for Link prediction), emphasizing its ability to jointly learn from three types of features.
3.1.5 Experimental results

We conduct extensive experiments to evaluate SEAL. Our results show that SEAL is a superb and robust framework for link prediction, achieving unprecedentedly strong performance on various networks. We use AUC as our evaluation metric. We run all experiments for 10 times and report the average results and standard deviations. SEAL is flexible with what GNN or node embeddings to use. Thus, we choose our own GNN architecture, DGCNN [184] (introduced in the last chapter), as the default GNN, and node2vec [56] as the default embeddings. The code and data are available at https://github.com/muhanzhang/SEAL.

Datasets. The eight datasets used are: USAir, NS, PB, Yeast, C.ele, Power, Router, and E.coli. USAir [11] is a network of US Air lines with 332 nodes and 2,126 edges. The average node degree is 12.81. NS [118] is a collaboration network of researchers in network science with 1,589 nodes and 2,742 edges. The average node degree is 3.45. PB [1] is a network of US political blogs with 1,222 nodes and 16,714 edges. The average node degree is 27.36. Yeast [160] is a protein-protein interaction network in yeast with 2,375 nodes and 11,693 edges. The average node degree is 9.85. C.ele [163] is a neural network of C. elegans with 297 nodes and 2,148 edges. The average node degree is 14.46. Power [163] is an electrical grid of western US with 4,941 nodes and 6,594 edges. The average node degree is 2.67. Router [149] is a router-level Internet with 5,022 nodes and 6,258 edges. The average node degree is 2.49. E.coli [183] is a pairwise reaction network of metabolites in E. coli with 1,805 nodes and 14,660 edges. The average node degree is 12.55.

We randomly remove 10% existing links from each dataset as positive testing data. Following a standard manner of learning-based link prediction, we randomly sample the same number of nonexistent links (unconnected node pairs) as negative testing data. We use the remaining 90% existing links as well as the same number of additionally sampled nonexistent links.
to construct the training data. We ensure all the methods compared use exactly the same training and testing data in every run to ensure a very fair empirical comparison.

**Comparison to heuristic methods.** We first compare SEAL with methods that only use graph structure features. We include eight popular heuristics (shown in Table 3.1): common neighbors (CN), Jaccard, preferential attachment (PA), Adamic-Adar (AA), resource allocation (RA), Katz, PageRank (PR), and SimRank (SR). We additionally include Ensemble (ENS) which trains a logistic regression classifier on the eight heuristic scores. We also include two heuristic learning methods: Weisfeiler-Lehman graph kernel (WLK) [142] and WLNM [182], which also learn from (truncated) enclosing subgraphs. We omit path ranking methods [89] as well as other recent methods which are specifically designed for knowledge graphs or recommender systems [113, 119]. As all the baselines only use graph structure features, we restrict SEAL to not include any latent or explicit features. In SEAL, the hop number $h$ is an important hyperparameter. Here, we select $h$ only from $\{1, 2\}$, since on one hand we empirically verified that the performance typically does not increase after $h \geq 3$, which validates our theoretical results that the most useful information is within local structures. On the other hand, even $h = 3$ sometimes results in very large subgraphs if a hub node is included. This raises the idea of sampling nodes in subgraphs, which we leave to future work. The selection principle is very simple: If the second-order heuristic AA outperforms the first-order heuristic CN on 10% validation data, then we choose $h = 2$; otherwise we choose $h = 1$. For datasets PB and E.coli, we consistently use $h = 1$ to fit into the memory. We include more details about the baselines and hyperparameters in Appendix A.1.

Table 3.2 shows the results. Firstly, we observe that methods which learn from enclosing subgraphs (WLK, WLNM and SEAL) generally perform much better than predefined heuristics. This indicates that the learned “heuristics” are better at capturing the network properties than manually designed ones. Among learning-based methods, SEAL has the best
performance, demonstrating GNN’s superior graph feature learning ability over graph kernels and fully-connected neural networks. From the results on Power and Router, we can see that although existing heuristics perform similarly to random guess, learning-based methods still maintain high performance. This suggests that we can even discover new “heuristics” for networks where no existing heuristics work.

**Comparison to latent feature methods.** Next we compare SEAL with six state-of-the-art latent feature methods: matrix factorization (MF), stochastic block model (SBM) [5], node2vec (N2V) [56], LINE [155], spectral clustering (SPC), and variational graph auto-encoder (VGAE) [78]. Among them, VGAE uses a GNN too. Please note the difference between VGAE and SEAL: VGAE uses a node-level GNN to learn node embeddings that best reconstruct the network, while SEAL uses a graph-level GNN to classify enclosing subgraphs.
Therefore, VGAE still belongs to latent feature methods. For SEAL, we additionally include the 128-dimensional node2vec embeddings in the node information matrix $X$. Since the datasets do not have node attributes, explicit features are not included.

Table 3.3 shows the results. As we can see, SEAL shows significant improvement over latent feature methods. One reason is that SEAL learns from both graph structures and latent features simultaneously, thus augmenting those methods that only use latent features. We observe that SEAL with node2vec embeddings outperforms pure node2vec by large margins. This implies that network embeddings alone may not be able to capture the most useful link prediction information located in the local structures. It is also interesting that compared to SEAL without node2vec embeddings (Table 3.2), joint learning does not always improve the performance.

To evaluate SEAL’s scalability, we show its single-GPU inference time performance in Table 3.4. As we can see, SEAL has good scalability. For networks with over 1E7 potential links, SEAL took less than an hour to make all the predictions. One possible way to further scale SEAL to social networks with millions of users is to first use some simple heuristics such as common neighbors to filter out most unlikely links and then use SEAL to make further recommendations. Another way is to restrict the candidate friend recommendations to be those who are at most 2 or 3 hops away from the target user, which will vastly reduce the number of candidate links to infer for each user and thus further increase the scalability.

Table 3.4: Inference time of SEAL.

<table>
<thead>
<tr>
<th></th>
<th>USAir</th>
<th>NS</th>
<th>PB</th>
<th>Yeast</th>
<th>C.ele</th>
<th>Power</th>
<th>Router</th>
<th>E.coli</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of potential links</td>
<td>5.49E+04</td>
<td>1.26E+06</td>
<td>7.46E+05</td>
<td>2.82E+06</td>
<td>4.40E+04</td>
<td>1.22E+07</td>
<td>1.26E+07</td>
<td>1.39E+06</td>
</tr>
<tr>
<td>Inference time per link (s)</td>
<td>6.05E-04</td>
<td>2.55E-04</td>
<td>2.04E-04</td>
<td>3.96E-04</td>
<td>4.13E-04</td>
<td>1.35E-04</td>
<td>2.13E-04</td>
<td>2.40E-04</td>
</tr>
<tr>
<td>Inference time for all potential links (s)</td>
<td>31</td>
<td>321</td>
<td>146</td>
<td>1106</td>
<td>16</td>
<td>1640</td>
<td>2081</td>
<td>328</td>
</tr>
</tbody>
</table>
Table 3.5: Comparing SEAL with network embedding methods (AUC) on larger networks, OOM: out of memory.

<table>
<thead>
<tr>
<th></th>
<th>N2V</th>
<th>LINE</th>
<th>SPC</th>
<th>VGAE</th>
<th>WLNM</th>
<th>SEAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>arXiv</td>
<td>96.18±0.40</td>
<td>84.64±0.03</td>
<td>87.00±0.14</td>
<td>OOM</td>
<td>99.19±0.03</td>
<td>99.40±0.14</td>
</tr>
<tr>
<td>Facebook</td>
<td>99.05±0.07</td>
<td>89.63±0.06</td>
<td>98.59±0.11</td>
<td>98.21±0.22</td>
<td>99.24±0.03</td>
<td>99.40±0.08</td>
</tr>
<tr>
<td>BlogCatalog</td>
<td>85.97±1.56</td>
<td>90.92±2.05</td>
<td>96.74±0.31</td>
<td>OOM</td>
<td>96.55±0.08</td>
<td>98.10±0.60</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>76.59±2.06</td>
<td>74.44±0.66</td>
<td>99.54±0.04</td>
<td>89.74±0.18</td>
<td>99.05±0.03</td>
<td>99.63±0.05</td>
</tr>
<tr>
<td>PPI</td>
<td>70.31±0.79</td>
<td>72.82±1.53</td>
<td>92.27±0.22</td>
<td>85.86±0.43</td>
<td>88.79±0.38</td>
<td>93.52±0.37</td>
</tr>
</tbody>
</table>

We further conduct experiments with the setting of the node2vec paper [56] on five networks: arXiv (18,722 nodes and 198,110 edges) [92], Facebook (4,039 nodes and 88,234 edges) [92], BlogCatalog (10,312 nodes, 333,983 edges and 39 attributes) [178], Wikipedia (4,777 nodes, 184,812 edges and 40 attributes) [108], and Protein-Protein Interactions (PPI) (3,890 nodes, 76,584 edges and 50 attributes) [150]. For each network, 50% of random links are removed and used as testing data, while keeping the remaining network connected. For Facebook and arXiv, all remained links are used as positive training data. For PPI, BlogCatalog and Wikipedia, we sample 10,000 remained links as positive training data. We compare SEAL (\(h = 1\), 10 training epochs) with node2vec, LINE, SPC, VGAE, and WLNM (\(K = 10\)). For node2vec, we use the parameters provided in [56] if available. For SEAL and VGAE, the node attributes are used since only these two methods support explicit features.

Table 3.5 shows the results. As we can see, SEAL consistently outperforms all embedding methods. Especially on the last three networks, SEAL (with node2vec embeddings) outperforms pure node2vec by large margins. These results indicate that in many cases, embedding methods alone cannot capture the most useful link prediction information, while effectively combining the power of different types of features results in much better performance. SEAL also consistently outperforms WLNM.
3.1.6 Conclusion

Learning link prediction heuristics automatically is a new field. In this section, we presented theoretical justifications for learning from local enclosing subgraphs. In particular, we proposed a $\gamma$-decaying theory to unify a wide range of high-order heuristics and prove their approximability from local subgraphs. Motivated by the theory, we proposed a novel link prediction framework, SEAL, to simultaneously learn from local enclosing subgraphs, embeddings and attributes based on graph neural networks. Experimentally we showed that SEAL achieved unprecedentedly strong performance by comparing to various heuristics, latent feature methods, and network embedding algorithms. We hope SEAL can not only inspire link prediction research, but also open up new directions for other relational machine learning problems such as knowledge graph completion and recommender systems.

3.2 Inductive Matrix Completion Based on Graph Neural Networks

3.2.1 Introduction

Collaborative filtering (CF) for recommender systems leverages collected ratings of items by users to make new recommendations. These collected ratings can be written as entries of an $m \times n$ rating matrix, where $m$ is the number of users and $n$ is the number of items. Many modern CF-based recommender systems try to solve the matrix completion problem through matrix factorization techniques. By assuming a low-rank rating matrix, matrix factorization decomposes the rating $r_{ij}$ that user $i$ gives to item $j$ into $r_{ij} = w_i^T h_j$, the product of user
$i$’s and item $j$’s low-dimensional latent feature vectors $w_i$ and $h_j$, respectively, which has achieved great successes [3, 12, 82, 137]

However, matrix factorization is intrinsically transductive, meaning that the learned latent features (embeddings) for users/items are not generalizable to users/items unseen during the training. When the rating matrix has changed values or has new rows/columns added, it often requires a complete retraining to get the new embeddings. To make matrix completion inductive, Inductive Matrix Completion (IMC) has been invented, which leverages content (side information) of users and items [63, 169]. In IMC, a rating is decomposed by $r_{ij} = x_i^\top Q y_j$, where $x_i$ and $y_j$ are content feature vectors of user $i$ and item $j$, respectively, and $Q$ is a learnable matrix modeling the interactions between feature dimensions. To accurately predict the missing entries, IMC methods have strong constraints on the quality of the content, which often leads to inferior performance. Other content-based recommender systems [101] face similar problems, such as the lack of high-quality content and the inferior performance than transductive methods.

In this paper, we propose a novel inductive matrix completion method for recommender systems **without** using any side information, **while** achieving state-of-the-art performance. The key that frees us from using any side information is the **graph pattern**. If for each existing rating we add an edge between the corresponding user and item, we can build a bipartite graph, where nodes are either users or items and an edge only exists between a user and an item. Subsequently, predicting unknown ratings corresponds to predicting labeled links in this bipartite graph. This transforms the matrix completion problem into a link prediction problem [96], where graph patterns play a major role in determining link existences.

A major class of link prediction methods are heuristic methods, which compute some node similarity scores as the likelihood of links. For example, the common neighbors heuristic [96]
count the common neighbors between two nodes to predict links, while the Katz index [74] uses a weighted sum of all the paths between two nodes. These heuristics can be seen as some predefined graph structure features calculated based on the local or global graph patterns around links, which have achieved great successes due to their simplicity, interpretability and for some of them, scalability.

However, most existing heuristics only work for simple graphs where nodes are of the same type. Can we find some heuristics for bipartite graph link prediction? Intuitively, such heuristics should exist. For example, if a user $u_0$ likes an item $v_0$, we may expect to see very often that $v_0$ is also liked by some other user $u_1$ who shares a similar taste to $u_0$. By similar taste, we mean $u_1$ and $u_0$ have together both liked some other item $v_1$. In the bipartite graph, such a pattern is realized as a “like” path $(u_0 \rightarrow_{\text{like}} v_1 \rightarrow_{\text{liked by}} u_1 \rightarrow_{\text{like}} v_0)$. If there are many such paths between $u_0$ and $v_0$, we may infer that $u_0$ is highly likely to like $v_0$. Therefore, we may count the number of such paths as an indicator of how likely $u_0$ likes $v_0$. In fact, many neighborhood-based recommender systems [40] rely on such heuristics.

Of course we can try to manually define many such intuitive heuristics and test their effectiveness for recommender systems. In this work, however, we take a different approach by automatically learn suitable heuristics from the given bipartite graph. To do so, we first extract an $h$-hop enclosing subgraph for each training (user, item) pair $(u, v)$, which is defined to be the subgraph induced from the bipartite graph by nodes $u, v$ and the neighbors of $u$ and $v$ within $h$ hops. Such local subgraphs contain rich information about the rating that $u$ may give to $v$. For example, the number of $(u_0 \rightarrow_{\text{like}} v_1 \rightarrow_{\text{liked by}} u_1 \rightarrow_{\text{like}} v_0)$ paths can just be computed from the 1-hop enclosing subgraph around $(u_0, v_0)$. By feeding these enclosing subgraphs to a graph neural network (GNN) [39, 77, 136, 184], we train a graph regression model that maps each subgraph to the rating that its center user gives to its center item.
Figure 3.2: An illustration of our IGMC framework. We extract a local enclosing subgraph around each training rating, and use a GNN to learn graph patterns that are useful for rating prediction. Note that the features listed inside the box are only for illustration – the learned features can be much more complex. We use the trained GNN to complete other missing entries of the matrix.

Due to the superior graph learning ability, a GNN can learn highly expressive graph structure features useful for rating prediction instead of only using predefined heuristics. Figure 3.2 illustrates the overall framework. Our Inductive Graph-based Matrix Completion (IGMC) model does not rely on any latent features associated with the training users/items. Thus, we can freely apply our trained model on unseen (user, item) pairs’ enclosing subgraphs without retraining. We can even transfer the model to other similar tasks. We evaluate our model on five benchmark matrix completion datasets, and show that it is highly competitive with state-of-the-art transductive methods. Without using any content, IGMC achieves the smallest RMSEs on four out of five datasets, beating those baselines using side information. Our model is also equipped with excellent transfer learning ability. We show that an IGMC model trained on the MovieLens-100k dataset can be directly used to predict Douban movie ratings and even outperform baselines trained specifically on Douban. We further demonstrate that our model is robust to sparse rating matrices. Our model outperforms transductive matrix factorization by a large margin under extremely sparse cases.
3.2.2 Related work

GNNs for matrix completion  The matrix completion problem has been studied from a graph point of view. [113] develops a multi-graph CNN (MGCNN) model to extract user and item latent features from their respective networks and use the latent features to predict the ratings. Later, [berg2017graph] proposes graph convolutional matrix completion (GC-MC) to directly operate on user-item bipartite graphs to extract user and item latent features using a GNN. Although using GNNs for matrix completion, both the two models are still transductive – the MGCNN model requires graph Laplacians which do not generalize to new tasks, while the GC-MC model uses one-hot encoding of node IDs as their initial features input to the GNN, thus cannot generalize to unseen users/items. A recent inductive graph-based recommender system, PinSage [172], replaces the one-hot encoding features in GC-MC with node content features, and is successfully used in recommending related pins in Pinterest. Although being inductive, PinSage relies heavily on the rich visual and text content information associated with the pins, which is rarely accessible in other recommendation tasks, especially for those tasks where user features are hard to collect. In comparison, our IGMC model is inductive and does not rely on any content information.

Link prediction based on graph patterns  Learning supervised heuristics (graph patterns) has been studied for link prediction in simple graphs. [182] propose Weisfeiler-Lehman Neural Machine (WLNM), which learns graph structure features using a fully-connected neural network on the subgraphs’ adjacency matrices. Later, they improve this work by replacing the fully-connected neural network with a GNN and achieves state-of-the-art link prediction results [181]. Our work generalizes this line of research from predicting link existence in simple graphs to predicting values on links in bipartite graphs (i.e., matrix completion), where multiple deep customized innovations are made in order to learn patterns
from bipartite graphs. In [22, 189], traditional link prediction heuristics are adapted to bipartite graphs and show promising performance for recommender systems. Our work differs in that we do not use any predefined heuristics, but learn general graph structure features using a GNN. Another similar work to ours is [93], where graph kernels are used to learn graph structure features. However, graph kernels require quadratic time and space complexity to compute and store the kernel matrices thus are unsuitable for modern recommender systems.

### 3.2.3 Inductive Graph-based Matrix Completion (IGMC)

We now present our Inductive Graph-based Matrix Completion (IGMC) framework. We use $G$ to denote the undirected bipartite graph constructed from the given rating matrix $R$. In $G$, a node is either a user (denoted by $u$, corresponding to a row in $R$) or an item (denoted by $v$, corresponding to a column in $R$). Edges only exist between user and item, but cannot exist between two users or two items. Each edge $(u, v)$ has a value $r$, corresponding to the rating that $u$ gives to $v$, i.e., the $(u, v)^{th}$ entry of $R$. We use $\mathcal{R}$ to denote the set of all possible ratings. We use $\mathcal{N}_r(u)$ to denote the set of $u$’s neighbors that connect to $u$ with edge type $r$.

#### Enclosing subgraph extraction

The first part of the IGMC framework is enclosing subgraph extraction. For each training (user, item, rating) tuple, we extract an $h$-hop enclosing subgraph around the (user, item) pair from $G$. We will feed these enclosing subgraphs to a GNN and regress on their ratings. Then, for each testing (user, item) pair, we again extract its $h$-hop enclosing subgraph, and use the trained GNN model to predict its rating. Algorithm 1 describes how we extract $h$-hop enclosing subgraphs. Note that after extracting a training enclosing subgraph for $(u, v)$, we must remove the target edge $(u, v)$ from it. This is because the rating on $(u, v)$ is what we want our model to predict, thus must be excluded from the subgraph.
Algorithm 1 Enclosing Subgraph Extraction

1: **input**: \( h \), target (user, item) \((u, v)\), the bipartite graph \( G \)
2: **output**: enclosing subgraph \( G_{u,v}^h \) for \((u, v)\)
3: \( U = U_{\text{fringe}} = \{u\}, V = V_{\text{fringe}} = \{v\} \)
4: **for** \( i = 1, 2, \ldots, h \) **do**
5: \( U_{\text{fringe}}' = \{u_i : u_i \sim V_{\text{fringe}}\} \setminus U \)
6: \( V_{\text{fringe}}' = \{v_i : v_i \sim U_{\text{fringe}}\} \setminus V \)
7: \( U_{\text{fringe}} = U_{\text{fringe}}', V_{\text{fringe}} = V_{\text{fringe}}' \)
8: \( U = U \cup U_{\text{fringe}}, V = V \cup V_{\text{fringe}} \)
9: Let \( G_{u,v}^h \) be the vertex-induced subgraph from \( G \) using vertices \( U, V \)
10: Remove edge \((u, v)\) from \( G_{u,v}^h \).
11: **end for**
12: **return** \( G_{u,v}^h \)

Notes: \( \{u_i : u_i \sim V_{\text{fringe}}\} \) is the set of nodes that are adjacent to at least one node in \( V_{\text{fringe}} \) with any edge type.

### Node labeling

The second part of IGMC is node labeling. Before we feed enclosing subgraphs to a GNN, we need to apply a node labeling to each enclosing subgraph. A node labeling is a function that returns an integer label for every node in the subgraph. The purpose is to use different labels to mark nodes’ different roles in a subgraph. Specifically, we need to: 1) distinguish the target user and the target item between which the target rating is located, and 2) differentiate user-type nodes from item-type nodes. To achieve these goals, we propose a node labeling scheme as follows: We first give label 0 to the target user and label 1 to the target item. Then for other nodes, we determine their labels according to at which hop they are included in the subgraph in Algorithm 1. If a user-type node is included at the \( i^{\text{th}} \) hop \((i = 1, \ldots, h)\), we will give it a label \( 2i \). If an item-type node is included at the \( i^{\text{th}} \) hop, we will give it \( 2i + 1 \). Such a node labeling can sufficiently discriminate: 1) target nodes from “context” nodes, 2) users from items (users always have even labels), and 3) nodes of different distances to the target user/item. Note that this is not the only possible way of node labeling, but we
empirically verified its excellent performance. The one-hot encoding of these node labels will be treated as the initial node features of a subgraph fed to the GNN. Note that our node labels are determined completely inside each enclosing subgraph, thus are independent of the global bipartite graph. Given a new enclosing subgraph, we can as well predict its rating even if all of its nodes are from a different bipartite graph. That is, we are using pure graph patterns for matrix completion instead of learning transductive latent feature models.

Our node labeling is also different from using global node IDs as in GC-MC [berg2017graph]. Using global IDs allows transforming the first GNN layer’s parameters into node embeddings associated with each particular ID. However, such a model cannot generalize to node IDs that are unseen during the training, thus is transductive. In PinSage [172], node content features are used as initial features. Our model does not use any content, thus is applicable to scenarios where content information is unavailable.

**Graph neural network architecture**

The third part of IGMC is to train a graph neural network (GNN) model predicting ratings from the enclosing subgraphs. That is, we use GNN as a function mapper mapping each enclosing subgraph to its target rating. There are two components in our GNN: 1) message passing layers which aggregate neighboring nodes’ features to the center to extract a feature vector for each node in the subgraph, and 2) a pooling layer to summarize a graph representation from node features. In our implementation, we use mini-batch training which feeds multiple enclosing subgraphs together to the GNN. For simplicity, here we introduce our GNN using a single subgraph as an example.
To handle different edge types, we adopt the relational graph convolutional operator (R-GCN) [138] as our GNN’s message passing layers. The R-GCN layer has the following form:

\[ x_{i}^{l+1} = W_0^l x_i^l + \sum_{r \in R} \sum_{j \in N_r(i)} \frac{1}{|N_r(i)|} W_r^l x_j^l, \]  

(3.11)

where \( x_i^l \) denotes node \( i \)'s feature vector at layer \( l \), \( x_i^{l+1} \) denotes its feature vector at layer \( l + 1 \), \( W_0^l \) and \( \{ W_r^l, r \in R \} \) are learnable parameter matrices. Since neighbors \( j \) connected to \( i \) with edge type \( r \) are processed by the parameter matrix \( W_r^l \) associated particularly with \( r \), we are able to learn rich graph patterns inside the edge types, such as the average rating the target user gives to items, the average rating the target item receives, and how two target nodes are connected by different paths etc. We apply \( L \) R-GCN layers with tanh activations between two layers. The node feature vectors from different layers are concatenated for node \( i \) as its final representation \( h_i \):

\[ h_i = \text{concat}(x_i^1, x_i^2, \ldots, x_i^L). \]  

(3.12)

Next, we pool the node representations into a graph-level feature vector. There are many choices such as the simple summing, or some advanced pooling layers such as SortPooling [184] and DiffPooling [173] etc. In this work, we use a different pooling layer which concatenates the final representations of only the target user and item as the graph representation:

\[ g = \text{concat}(h_u, h_v), \]  

(3.13)

where we use \( h_u \) and \( h_v \) to denote the final representations of the target user and the target item, respectively. Our particular choice is due to the extra importance that the these two center nodes carry compared to other context nodes. Although being very simple, we
empirically verified its better performance than summing and other advanced pooling layers for our tasks.

After getting the final graph representation, we use an MLP to output the predicted rating:

\[
\hat{r} = w^\top \sigma(Wg),
\]

where \( W \) and \( w \) are parameters of the MLP which map the graph representation \( g \) to a scalar rating \( \hat{r} \), and \( \sigma \) is an activation function (we take ReLU in this paper).

**Model training**

**Loss function** We minimize the mean squared error (MSE) between the predictions and the ground truth rating:

\[
\mathcal{L} = \frac{1}{|\{(u, v) | \Omega_{u,v} = 1\}|} \sum_{(u,v): \Omega_{u,v} = 1} (r_{u,v} - \hat{r}_{u,v})^2,
\]

where we use \( r_{u,v} \) and \( \hat{r}_{u,v} \) to denote the true rating and predicted rating of \((u, v)\), respectively, and \( \Omega_{u,v} \) is a 0/1 mask matrix indicating the observed entries of the rating matrix \( R \).

**Continuous rating regularization** The R-GCN layer (3.11) used in our GNN has different parameters \( W_r \) for different ratings. One drawback here is that it fails to to take the continuity of ratings into consideration. For instance, a rating of 4 and a rating of 5 in MovieLens both indicate that the user likes the movie, while a rating of 1 indicates that the user does not like the movie. Thus, we would like our model to be aware of the fact that a rating of 4 is more similar to 5 than 1 is. In R-GCN, however, ratings 1, 4 and 5 are all treated as discrete edge types associated with their own parameter matrices. The continuity information of the ratings are completely lost. To fix that, we propose a continuous rating regularization technique,
which encourages neighboring ratings to have similar parameter matrices. Assume the ratings in $\mathcal{R}$ exhibit an ordering $\{r_1, r_2, \ldots, r_{|\mathcal{R}|}\}$ which shows increasingly higher preference that users give to items. Then, the regularization loss is:

$$L_{\text{reg}} = \sum_{i=1,2,\ldots,|\mathcal{R}|-1} \| W_{r_{i+1}} - W_{r_i} \|_F^2,$$

(3.16)

where $\| \cdot \|_F$ denotes the Frobenius norm of a matrix. The above regularization restrain the parameter matrices of adjacent ratings from having too much differences, which not only embeds the continuity of ratings into the continuity of $W_r$, but also helps the optimization of those infrequent ratings by transferring knowledge from their nearby ratings. The final loss function is given by:

$$L + \lambda L_{\text{reg}},$$

(3.17)

where $\lambda$ trades-off the importance of the MSE loss and the regularization.

### 3.2.4 Experiments

We conduct experiments on five common matrix completion datasets: Flixster [64], Douban [105], YahooMusic [42], MovieLens-100K and MovieLens-1M [111]. For ML-100K, we train and evaluate on the canonical u1.base/u1.test train/test split. For ML-1M, we randomly split it into 90% and 10% train/test sets. For Flixster, Douban and YahooMusic we use the preprocessed subsets and splits provided by [113]. Dataset statistics are summarized in Table 3.6. We implemented IGMC using PyTorch_Geometric [47]. We tuned model hyperparameters based on cross validation results on ML-100K, and use them across all datasets. The final architecture uses 4 R-GCN layers with 32, 32, 32, 32 hidden dimensions. Basis decomposition with 4 bases is used to reduce the number of parameters in $W_r$ [138].
The final MLP has 128 hidden units and a dropout rate 0.5. We use 1-hop enclosing subgraphs for all datasets, and find them sufficiently good. We find using 2 or more hops can slightly increase the performance but take much longer training time. For each enclosing subgraph, we randomly dropout its adjacency matrix entries with a rate 0.2 during the training. We train our model using the Adam optimizer [76] with a batch size of 50 and an initial learning rate of 0.001, and multiply the learning rate by 0.1 every 20 epochs for ML-1M, and every 50 epochs for all other datasets. We will make all the code publicly available.

Table 3.6: Statistics of each dataset.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Users</th>
<th>Items</th>
<th>Ratings</th>
<th>Density</th>
<th>Rating types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flixster</td>
<td>3,000</td>
<td>3,000</td>
<td>26,173</td>
<td>0.0029</td>
<td>0.5, 1, 1.5, ..., 5</td>
</tr>
<tr>
<td>Douban</td>
<td>3,000</td>
<td>3,000</td>
<td>136,891</td>
<td>0.0152</td>
<td>1, 2, 3, 4, 5</td>
</tr>
<tr>
<td>YahooMusic</td>
<td>3,000</td>
<td>3,000</td>
<td>5,335</td>
<td>0.0006</td>
<td>1, 2, 3, ..., 100</td>
</tr>
<tr>
<td>MovieLens</td>
<td>943</td>
<td>1,682</td>
<td>100,000</td>
<td>0.0630</td>
<td>1, 2, 3, 4, 5</td>
</tr>
</tbody>
</table>

Flixster, Douban and YahooMusic

For these three datasets, we compare our IGMC with GRALS [130], sRGCNN [113], GC-MC [berg2017graph], F-EAE [58], and PinSage [172]. Among them, GRALS is a graph regularized matrix completion algorithm. GC-MC and sRGCNN are GNN-assisted matrix completion methods, where GNNs are used to learn better user/item latent features to reconstruct the rating matrix. Thus, they are still transductive models. F-EAE uses a factorized exchangeable autoencoder to perform permutation-equivariant operations to reconstruct the rating matrix, which is an inductive model without using content, similar to our IGMC. PinSage is an inductive GNN-based model which relies on node content features. We further implemented an inductive GC-MC model (IGC-MC) which replaces the one-hot
Table 3.7: RMSE test results on Flixster, Douban and YahooMusic.

<table>
<thead>
<tr>
<th>Model</th>
<th>Inductive</th>
<th>Content</th>
<th>Flixster</th>
<th>Douban</th>
<th>YahooMusic</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRALS</td>
<td>no</td>
<td>yes</td>
<td>1.245</td>
<td>0.833</td>
<td>38.0</td>
</tr>
<tr>
<td>sRGCNN</td>
<td>no</td>
<td>yes</td>
<td>0.926</td>
<td>0.801</td>
<td>22.4</td>
</tr>
<tr>
<td>GC-MC</td>
<td>no</td>
<td>yes</td>
<td>0.917</td>
<td>0.734</td>
<td>20.5</td>
</tr>
<tr>
<td>IGC-MC</td>
<td>yes</td>
<td>yes</td>
<td>0.999±0.062</td>
<td>0.990±0.082</td>
<td>21.3±0.989</td>
</tr>
<tr>
<td>F-EAE</td>
<td>yes</td>
<td>no</td>
<td>0.908</td>
<td>0.738</td>
<td>20.0</td>
</tr>
<tr>
<td>PinSage</td>
<td>yes</td>
<td>no</td>
<td>0.954±0.005</td>
<td>0.739±0.002</td>
<td>22.9±0.629</td>
</tr>
<tr>
<td>IGMC (ours)</td>
<td>yes</td>
<td>no</td>
<td>0.872±0.001</td>
<td>0.721±0.001</td>
<td>19.1±0.138</td>
</tr>
</tbody>
</table>

encoding of node IDs with the content features, in order to show the advantages of learning subgraph features compared to learning subtree features.

We train our model for 40 epochs, and save the model parameters every 10 epochs. The final predictions are given by averaging the predictions from epochs 10, 20, 30 and 40. We repeat the experiment five times and report the average RMSEs. Table 3.7 shows the results. The baseline results are taken from [58]. Our model achieves state-of-the-art RMSE results on all three datasets, significantly outperforming the compared transductive and inductive baselines. Note that all baselines here except F-EAE use side information such as user-user or item-item graphs, while IGMC and F-EAE do not use any side information. This further highlights IGMC’s great performance advantages even using less information.

**ML-100K and ML-1M**

We further conduct experiments on MovieLens datasets. For ML-100K, we compare against matrix completion (MC) [20], inductive matrix completion (IMC) [63], and geometric matrix completion (GMC) [72], as well as GRALS, sRGCNN, GC-MC, F-EAE and PinSage. User/item side information are used in baselines if possible. For IGMC, we train our model
Table 3.8: RMSE test results on MovieLens-100K.

<table>
<thead>
<tr>
<th>Model</th>
<th>Inductive</th>
<th>Content</th>
<th>ML-100K</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC</td>
<td>no</td>
<td>no</td>
<td>0.973</td>
</tr>
<tr>
<td>IMC</td>
<td>no</td>
<td>yes</td>
<td>1.653</td>
</tr>
<tr>
<td>GMC</td>
<td>no</td>
<td>yes</td>
<td>0.996</td>
</tr>
<tr>
<td>GRALS</td>
<td>no</td>
<td>yes</td>
<td>0.945</td>
</tr>
<tr>
<td>sRGCNN</td>
<td>no</td>
<td>yes</td>
<td>0.929</td>
</tr>
<tr>
<td>GC-MC</td>
<td>no</td>
<td>yes</td>
<td>0.905</td>
</tr>
<tr>
<td>IGC-MC</td>
<td>yes</td>
<td>yes</td>
<td>1.142</td>
</tr>
<tr>
<td>F-EAE</td>
<td>yes</td>
<td>no</td>
<td>0.920</td>
</tr>
<tr>
<td>PinSage</td>
<td>yes</td>
<td>yes</td>
<td>0.951</td>
</tr>
<tr>
<td>IGMC</td>
<td>yes</td>
<td>no</td>
<td><strong>0.904</strong></td>
</tr>
</tbody>
</table>

for 80 epochs and report the ensemble performance of epochs 50, 60, 70 and 80. Results are summarized in Table 3.8. For ML-1M, besides the baselines GC-MC, F-EAE and PinSage, we further include many other state-of-the-art algorithms including PMF [112], I-RBM [134], NNMF [44], I-AutoRec [141] and CF-NADE [187]. We train IGCM for 40 epochs and report the ensemble performance of epochs 25, 30, 35 and 40. Results are summarized in Table 3.9. As we can see, IGMC achieves the best performance on ML-100K, which beats state-of-the-art methods such as GC-MC. Note also that our model uses purely local subgraph structure features and is inductive, while GC-MC is transductive and additionally uses content features. For ML-1M, IGMC cannot catch up with state-of-the-art models such as CF-NADE and GC-MC, but outperforms all other inductive models.

Transfer learning

To verify the transferability of the learned IGMC model, we conduct a transfer learning experiment by applying the model trained on ML-100K to Flixster, Douban and YahooMusic. Among the three datasets, only Douban has exactly the same rating types as ML-100K.
Table 3.9: RMSE test results on MovieLens-1M.

<table>
<thead>
<tr>
<th>Model</th>
<th>Inductive</th>
<th>Content</th>
<th>ML-1M</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMF</td>
<td>no</td>
<td>no</td>
<td>0.883</td>
</tr>
<tr>
<td>I-RBM</td>
<td>no</td>
<td>no</td>
<td>0.854</td>
</tr>
<tr>
<td>NNMF</td>
<td>no</td>
<td>no</td>
<td>0.843</td>
</tr>
<tr>
<td>I-AutoRec</td>
<td>no</td>
<td>no</td>
<td>0.831</td>
</tr>
<tr>
<td>CF-NADE</td>
<td>no</td>
<td>no</td>
<td>0.829</td>
</tr>
<tr>
<td>GC-MC</td>
<td>no</td>
<td>no</td>
<td>0.832</td>
</tr>
<tr>
<td>IGC-MC</td>
<td>yes</td>
<td>yes</td>
<td>1.259</td>
</tr>
<tr>
<td>F-EAE</td>
<td>yes</td>
<td>no</td>
<td>0.860</td>
</tr>
<tr>
<td>PinSage</td>
<td>yes</td>
<td>yes</td>
<td>0.906</td>
</tr>
<tr>
<td>IGMC</td>
<td>yes</td>
<td>no</td>
<td>0.857</td>
</tr>
</tbody>
</table>

Table 3.10: RMSE of transferring models trained on ML-100K to Flixster, Douban and YahooMusic.

<table>
<thead>
<tr>
<th>Model</th>
<th>Inductive</th>
<th>Content</th>
<th>Flixster</th>
<th>Douban</th>
<th>YahooMusic</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGC-MC</td>
<td>yes</td>
<td>no</td>
<td>1.290</td>
<td>1.144</td>
<td>25.7</td>
</tr>
<tr>
<td>F-EAE</td>
<td>yes</td>
<td>no</td>
<td>0.987</td>
<td>0.766</td>
<td>23.3</td>
</tr>
<tr>
<td>IGMC (ours)</td>
<td>yes</td>
<td>no</td>
<td>0.906</td>
<td>0.759</td>
<td>20.1</td>
</tr>
</tbody>
</table>

(1,2,3,4,5). For Flixster and YahooMusic, we bin their edge types into 1-5 before feeding into the ML-100K model, and multiply the YahooMusic predictions by 20 to account for the different scales. Despite all the compromises, the transferred ML-100K model achieves excellent performance on the three datasets (Table 3.10), even outperforming many transductive baselines trained especially on each dataset (Table 3.7). We further show the transfer learning results of other two inductive models, IGC-MC and F-EAE, in Table 3.10. Note that an inductive model using content features (such as PinSage) is not transferrable, due to the different feature dimensions between the source and target domains. Thus, we replace
the content features in IGC-MC with node degrees here. As we can see, IGMC outperforms the other two models by large margins in terms of transfer learning ability.

### 3.2.5 Conclusion

In this section, we have proposed Inductive Graph-based Matrix Completion (IGMC) for recommender systems. Instead of learning transductive latent features, IGMC learns local graph patterns related to ratings inductively based on graph neural networks. IGMC shows highly competitive performance compared with traditional matrix completion baselines. In addition, the inductive model is transferrable to new tasks without any retraining, a property much desired in those recommendation tasks that have few training data. We believe IGMC will open a new direction for matrix completion and recommender systems.
Chapter 4

Graph Neural Networks for Graph Structure Optimization

In this chapter, we describe our final contribution of the dissertation – exploring graph neural networks’ use cases for graph structure optimization. We focus on one specific types of graphs, directed acyclic graphs (DAGs), since DAG structure optimization is most common in the real world. DAGs are often used to represent computations. For example, neural networks, Bayesian networks and electronic circuit blocks are all DAGs representing some computation tasks. The optimization of their structures for better performance or efficiency is a nontrivial and very important problem.

To optimize DAG structures, we propose a variational autoencoder (VAE) model for DAGs, named D-VAE [185]. Our D-VAE model can not only generate DAGs from its latent space, but also optimize DAG structures by performing Bayesian optimization in the latent space. To encode DAGs, we use a graph neural network based on a novel asynchronous message passing scheme which specifically works for DAGs. Instead of performing message passing
simultaneously for all nodes like in most GNNs, we perform message passing asynchronously respecting the partial order of a DAG. This is similar to the hierarchical attention propagation model proposed in Chapter 2, which is designed for embedding medical knowledge DAGs. In addition, here we prove our proposed asynchronous message passing can injectively encode computations defined on DAGs. The theorem ensures that every computation can be uniquely embedded into the latent space, so that instead of performing discrete optimization in graph space, we can equivalently perform continuous optimization in the VAE latent space. The computation encoding property also promotes the smoothness of the learned latent space w.r.t. computation graph’s performance, facilitating the optimization. We apply our model to two types of DAGs: neural network architectures and Bayesian networks. Experimental results show that our model not only generates valid and novel DAGs, but also achieves promising results in neural architecture search (NAS) and Bayesian network structure learning (BNSL), two important DAG optimization tasks.

4.1 Introduction

Many real-world problems can be posed as optimizing of a directed acyclic graph (DAG) representing some computational task. For example, the architecture of a neural network is a DAG. The problem of searching optimal neural architectures is essentially a DAG optimization task. Similarly, one critical problem in learning graphical models – optimizing the connection structures of Bayesian networks [80], is also a DAG optimization task. DAG optimization is pervasive in other fields as well. In electronic circuit design, engineers need to optimize DAG circuit blocks not only to realize target functions, but also to meet specifications such as power usage and operating temperature.
DAG optimization is a hard problem. Firstly, the evaluation of a DAG’s performance is often time-consuming (e.g., training a neural network). Secondly, state-of-the-art black-box optimization techniques such as simulated annealing and Bayesian optimization primarily operate in a continuous space, thus are not directly applicable to DAG optimization due to the discrete nature of DAGs. In particular, to make Bayesian optimization work for discrete structures, we need a kernel to measure the similarity between discrete structures as well as a method to explore the design space and extrapolate to new points. Principled solutions to these problems are still lacking.

Is there a way to circumvent the trouble from discreteness? The answer is yes. If we can embed all DAGs to a continuous space and make the space relatively smooth, we might be able to directly use principled black-box optimization algorithms to optimize DAGs in this space, or even use gradient methods if gradients are available. Recently, there has been increased interest in training generative models for discrete data types such as molecules [54, 88], arithmetic expressions [87], source code [52], undirected graphs [95], etc. In particular, [88] developed a grammar variational autoencoder (GVAE) for molecules, which is able to encode and decode molecules into and from a continuous latent space, allowing one to optimize molecule properties by searching in this well-behaved space instead of a discrete space. Inspired by this work, we propose to also train a variational autoencoder for DAGs, and optimize DAG structures in the latent space via Bayesian optimization.

To encode DAGs, we leverage graph neural networks (GNNs) [166]. Traditionally, a GNN treats all nodes symmetrically, and extracts local features around nodes by simultaneously passing all nodes’ neighbors’ messages to themselves. However, such a simultaneous message passing scheme is designed to learn local structure features. It might not be suitable for DAGs, since in a DAG: 1) nodes are not symmetric, but intrinsically have some ordering
based on its dependency structure; and 2) we are more concerned about the computation represented by the entire graph, not the local structures.

In this chapter, we propose an \textbf{asynchronous message passing scheme} to encode the computations on DAGs. The message passing no longer happens at all nodes simultaneously, but respects the computation dependencies (the partial order) among the nodes. For example, suppose node A has two predecessors, B and C, in a DAG. Our scheme does not perform feature learning for A until the feature learning on B and C are both finished. Then, the aggregated message from B and C is passed to A to trigger A’s feature learning. We incorporate this feature learning scheme in both our encoder and decoder, and propose the \textit{DAG variational autoencoder} (D-VAE). D-VAE has an excellent theoretical property for modeling DAGs – we prove that D-VAE can \textbf{injectively encode computations} on DAGs. This means, we can build a mapping from the discrete space to a continuous latent space so that every DAG computation has its \textbf{unique} embedding in the latent space, which \textbf{justifies} performing optimization in the latent space instead of the original design space.

Our contributions in this chapter are: 1) We propose D-VAE, a variational autoencoder for DAGs using a novel asynchronous message passing scheme, which is able to injectively encode computations. 2) Based on D-VAE, we propose a new DAG optimization framework which performs Bayesian optimization in a continuous latent space. 3) We apply D-VAE to two problems, neural architecture search and Bayesian network structure learning. Experiments show that D-VAE not only generates novel and valid DAGs, but also learns smooth latent spaces effective for optimizing DAG structures.
4.2 Related Work

Variational autoencoder (VAE) [75, 133] provides a framework to learn both a probabilistic generative model $p_\theta(x|z)$ (the decoder) as well as an approximated posterior distribution $q_\phi(z|x)$ (the encoder). VAE is trained through maximizing the evidence lower bound

$$
\mathcal{L}(\phi, \theta; x) = \mathbb{E}_{z \sim q_\phi(z|x)}[\log p_\theta(x|z)] - \text{KL}[q_\phi(z|x)\|p(z)].
$$

(4.1)

The posterior approximation $q_\phi(z|x)$ and the generative model $p_\theta(x|z)$ can in principle take arbitrary parametric forms whose parameters $\phi$ and $\theta$ are output by the encoder and decoder networks. After learning $p_\theta(x|z)$, we can generate new data by decoding latent space vectors $z$ sampled from the prior $p(z)$. For generating discrete data, $p_\theta(x|z)$ is often decomposed into a series of decision steps.

Deep graph generative models use neural networks to learn distributions over graphs. There are mainly three types: token-based, adjacency-matrix-based, and graph-based. Token-based models [36, 54, 88] represent a graph as a sequence of tokens (e.g., characters, grammar rules) and model these sequences using RNNs. They are less general since task-specific graph grammars such as SMILES for molecules [164] are required. Adjacency-matrix-based models [14, 37, 106, 146, 175] leverage the proxy adjacency matrix representation of a graph, and generate the matrix in one shot or generate the columns/entries sequentially. In contrast, graph-based models [69, 95, 100, 174] seem more natural, since they operate directly on graph structures (instead of proxy matrix representations) by iteratively adding new nodes/edges to a graph based on the the existing graph and node states. In addition, the graph and node states are learned by graph neural networks (GNNs), which have already shown their powerful graph representation learning ability on various tasks [43, 57, 77, 94, 121, 181, 184].
Neural architecture search (NAS) aims at automating the design of neural network architectures. It has seen major advances in recent years [45, 99, 128, 131, 191, 192]. See [62] for an overview. NAS methods can be mainly categorized into: 1) reinforcement learning methods [128, 191, 192] which train controllers to generate architectures with high rewards in terms of validation accuracy, 2) Bayesian optimization based methods [73] which define kernels to measure architecture similarity and extrapolate the architecture space heuristically, 3) evolutionary approaches [98, 110, 131] which use evolutionary algorithms to optimize neural architectures, and 4) differentiable methods [19, 99, 103] which use continuous relaxation/mapping of neural architectures to enable gradient-based optimization. In Appendix A.2, we include more discussion on several most related works.

Bayesian network structure learning (BNSL) is to learn the structure of the underlying Bayesian network from observed data [32, 50, 51, 97]. Bayesian network is a probabilistic graphical model which represents conditional dependencies among variables via a DAG [80]. One main approach for BNSL is score-based search, i.e., we define some “goodness-of-fit” score for network structures, and search for one with the optimal score in the discrete design space. Commonly used scores include BIC and BDeu, mostly based on marginal likelihood [80]. Due to the NP-hardness [24], however, exact algorithms such as dynamic programming [147] or shortest path approaches [176, 177] can only solve small-scale problems. Thus, people have to resort to heuristic methods such as local search and simulated annealing, etc. [25]. In general, BNSL is still a hard problem with much research ongoing.

4.3 DAG Variational Autoencoder (D-VAE)

In this section, we describe our proposed DAG variational autoencoder (D-VAE). D-VAE uses an asynchronous message passing scheme to encode and decode DAGs. In contrast to
the simultaneous message passing in traditional GNNs, D-VAE allows encoding *computations* rather than *structures*.

**Definition 4.1. (Computation)** Given a set of elementary operations $\mathcal{O}$, a computation $C$ is the composition of a finite number of operations $o \in \mathcal{O}$ applied to an input signal $x$, with the output of each operation being the input to its succeeding operations.

![Diagram of computations represented by DAGs.](image)

Figure 4.1: Computations can be represented by DAGs. Note that the left and right DAGs represent the same computation.

The set of elementary operations $\mathcal{O}$ depends on specific applications. For example, when we are interested in computations given by a calculator, $\mathcal{O}$ will be the set of all the operations defined on the functional buttons, such as $+$, $-$, $\times$, $\div$, etc. When modeling neural networks, $\mathcal{O}$ can be a predefined set of basic layers, such as $3 \times 3$ convolution, $5 \times 5$ convolution, $2 \times 2$ max pooling, etc. A computation can be represented as a directed acyclic graph (DAG), with directed edges representing signal flow directions among node operations. The graph must be acyclic, since otherwise the input signal will go through an infinite number of operations so that the computation never stops. Figure 4.1 shows two examples. Note that the two DAGs in Figure 4.1 represent the same computation, as the input signal goes through exactly the same operations.

**Computation vs. Function.** Have defined what is a computation, it is worth discussing the difference between a computation and a function. A computation $C_1 := x + 1 - 1$ defines
a function \( f(x) = x \). However, computations \( C_2 := x - 1 + 1 \) and \( C_3 := x \) also define the same function \( f(x) = x \), but \( C_1 \), \( C_2 \) and \( C_3 \) are different computations. In other words, a computation is (informally speaking) a process which focuses on the course of how the input is processed into the output, while a function is a mapping which cares about the results. Different computations can define the same function.

Sometimes, the same computation can also define different functions, e.g., two identical neural architectures will represent different functions given they are trained differently (since the weights of their layers will be different). In D-VAE, we model computations instead of functions, since 1) modeling functions is much harder than modeling computations (requires understanding the semantic meaning of each operation, such as the cancelling out of + and −), and 2) modeling functions additionally requires knowing the parameters of some operations, which are unknown before training.

Note also that in Definition 4.1, we only allow one single input signal. But in real world a computation sometimes has multiple initial input signals. However, the case of multiple input signals can be reduced to the single input case by adding an initial assignment operation that assigns the combined input signal to their corresponding next-level operations. For ease of presentation, we uniformly assume single input throughout this chapter.

### 4.3.1 Encoding

We first introduce D-VAE’s encoder. The D-VAE encoder can be seen as a graph neural network (GNN) using an asynchronous message passing scheme. Given a DAG, we assume there is a single starting node which does not have any predecessors (e.g., the input layer of a neural architecture). If there are multiple such nodes, we add a virtual starting node connecting to all of them.
Similar to standard GNNs, we use an update function $U$ to compute the hidden state of each node based on its neighbors’ incoming message. The hidden state of node $v$ is given by:

$$h_v = U(x_v, h_v^{in}),$$

(4.2)

where $x_v$ is the one-hot encoding of $v$’s type, and $h_v^{in}$ represents the incoming message to $v$. $h_v^{in}$ is given by aggregating the hidden states of $v$’s predecessors using an aggregation function $A$:

$$h_v^{in} = A(\{h_u : u \rightarrow v\}),$$

(4.3)

where $u \rightarrow v$ denotes there is a directed edge from $u$ to $v$, and $\{h_u : u \rightarrow v\}$ represents a multiset of $v$’s predecessors’ hidden states. If an empty set is input to $A$ (corresponding to the case for the starting node without any predecessors), we let $A$ output an all-zero vector.

Compared to the traditional simultaneous message passing, in D-VAE the message passing for a node must wait until all of its predecessors’ hidden states have already been computed. This simulates how a computation is really performed – to execute some operation, we also need to wait until all its input signals are ready. To make sure the required hidden states are available when a new node comes, we can perform message passing for nodes sequentially following a topological ordering of the DAG.

In Figure 4.2, we use a real neural architecture to illustrate the encoding process. After all nodes’ hidden states are computed, we use $h_{v_n}$, the hidden state of the ending node $v_n$ without any successors, as the output of the encoder. Then we feed $h_{v_n}$ to two multi-layer perceptrons (MLPs) to get the mean and variance parameters of $q_φ$ in (4.1) which is a normal distribution in our experiments. If there are multiple nodes without successors, we again add a virtual ending node connecting from all of them.
Figure 4.2: An illustration of the encoding procedure for a neural architecture. Following a topological ordering, we iteratively compute the hidden state for each node (red) by feeding in its predecessors’ hidden states (blue). This simulates how an input signal goes through the DAG, with $h_v$ simulating the output signal at node $v$.

Note that although topological orderings are usually not unique for a DAG, we can take any one of them as the message passing order while ensuring the encoder output is always the same, formalized by the following theorem.

**Theorem 4.1.** The D-VAE encoder is invariant to node permutations of the input DAG if the aggregation function $\mathcal{A}$ is invariant to the order of its inputs (e.g., summing, averaging, etc.).

**Proof.** Let $v_1$ be the starting node with no predecessors. By assumption, $v_1$ is the single starting node no matter how we permute the nodes of the input DAG. For $v_1$, the aggregation function $\mathcal{A}$ always outputs a zero vector. Thus, $h_{v_1}^{in}$ is invariant to node permutations. Subsequently, the hidden state $h_{v_1} = U(x_{v_1}, h_{v_1}^{in})$ is also invariant to node permutations.

Now we prove the theorem by structural induction. Consider node $v$. Suppose for every predecessor $u$ of $v$, the hidden state $h_u$ is invariant to node permutations. We will show that $h_v$ is also invariant to node permutations. Notice that in (4.3), the output $h_v^{in}$ by $\mathcal{A}$ is invariant to node permutations, since $\mathcal{A}$ is invariant to the order of its inputs $h_u$, and all $h_u$ are invariant to node permutations. Subsequently, node $v$’s hidden state $h_v = U(x_v, h_v^{in})$ is invariant to node permutations. By induction, we know that every node’s hidden state is
invariant to node permutations, including the ending node’s hidden state. Thus, the D-VAE encoder is invariant to node permutations.

Theorem 4.1 means isomorphic DAGs will have the same encoding result, no matter how we reorder/reindex the nodes. It also indicates that so long as we encode a DAG complying with its partial order, the real message passing order and node order do not influence the encoding result.

The next theorem shows another property of D-VAE that is crucial for its success in modeling DAGs, i.e., it is able to injectively encode computations on DAGs.

**Theorem 4.2.** Let $G$ be any DAG representing some computation $C$. Let $v_1, \ldots, v_n$ be its nodes following a topological order each representing some operation $o_i$, $1 \leq i \leq n$, where $v_n$ is the ending node. Then, the encoder of D-VAE maps $C$ to $h_{v_n}$ injectively if $A$ is injective and $U$ is injective.

**Proof.** Suppose there is an arbitrary input signal $x$ fed to the starting node $v_1$. For convenience, we will use $C_i(x)$ to denote the output signal at vertex $v_i$, where $C_i$ represents the composition of all the operations along the paths from $v_1$ to $v_i$.

For the starting node $v_1$, remember we feed a fixed $h_{v_1}^{\text{in}} = 0$ to (4.2), thus $h_{v_1}$ is also fixed. Since $C_1$ also represents a fixed input operation, we know that the mapping from $C_1$ to $h_{v_1}$ is injective. Now we prove the theorem by induction. Assume the mapping from $C_j$ to $h_{v_j}$ is injective for all $1 \leq j < i$. We will prove that the mapping from $C_i$ to $h_{v_i}$ is also injective.
Let $\phi_j(C_j) = h_{v_j}$ where $\phi_j$ is injective. Consider the output signal $C_i(x)$, which is given by feeding $\{C_j(x) : v_j \rightarrow v_i\}$ to $o_i$. Thus,

$$C_i(x) = o_i(\{C_j(x) : v_j \rightarrow v_i\}). \quad (4.4)$$

In other words, we can write $C_i$ as

$$C_i = \psi(o_i, \{C_j : v_j \rightarrow v_i\}), \quad (4.5)$$

where $\psi$ is an injective function used for defining the composite computation $C_i$ based upon $o_i$ and $\{C_j : v_j \rightarrow v_i\}$. Note that $\{C_j : v_j \rightarrow v_i\}$ can be either unordered or ordered depending on the operation $o_i$. For example, if $o_i$ is some symmetric operations such as adding or multiplication, then $\{C_j : v_j \rightarrow v_i\}$ can be unordered. If $o_i$ is some operation like subtraction or division, then $\{C_j : v_j \rightarrow v_i\}$ must be ordered.

With (4.2) and (4.3), we can write the hidden state $h_{v_i}$ as follows:

$$h_{v_i} = U(x_{v_i}, A(\{h_{v_j} : v_j \rightarrow v_i\}))$$

$$= U(O(o_i), A(\{\phi_j(C_j) : v_j \rightarrow v_i\})), \quad (4.6)$$

where $O$ is the injective one-hot encoding function mapping $o_i$ to $x_{v_i}$. In the above equation, $U, O, A, \phi_j$ are all injective. Since the composition of injective functions is injective, there exists an injective function $\varphi$ so that

$$h_{v_i} = \varphi(o_i, \{C_j : v_j \rightarrow v_i\}). \quad (4.7)$$
Then combining (4.5) we have:

\[ h_{vi} = \varphi \circ \psi^{-1}(\psi(o_i, \{C_j : v_j \rightarrow v_i\})) \]

\[ = \varphi \circ \psi^{-1}(C_i). \tag{4.8} \]

\( \varphi \circ \psi^{-1} \) is injective since the composition of injective functions is injective. Thus, we have proved that the mapping from \( C_i \) to \( h_{vi} \) is injective.

The significance of Theorem 4.2 is that it provides a way to injectively encode computations on DAGs, so that every computation has a unique embedding in the latent space. Therefore, instead of performing optimization in the original discrete space, we may equivalently perform optimization in the continuous latent space. In this well-behaved Euclidean space, distance is well defined, and principled Bayesian optimization can be applied to search for latent points with high performance scores, which transforms the discrete optimization problem into an easier continuous problem.

Note that Theorem 4.2 states D-VAE injectively encodes computations on graph structures, rather than graph structures themselves. Being able to injectively encode graph structures is a very strong condition, as it might provide an efficient algorithm to solve the challenging graph isomorphism (GI) problem. Luckily, here what we really want to injectively encode are computations instead of structures, since we do not need to differentiate two different structures \( G_1 \) and \( G_2 \) as long as they represent the same computation. Figure 4.1 shows such an example. Our D-VAE can identify that the two DAGs in Figure 4.1 actually represent the same computation by encoding them to the same vector, while those encoders focusing on encoding structures might fail to capture the underlying computation and output different vectors.
To model and learn the injective functions $A$ and $U$, we resort to neural networks thanks to the universal approximation theorem [61]. For example, we can let $A$ be a gated sum:

$$h_v^{in} = \sum_{u \rightarrow v} g(h_u) \odot m(h_u),$$

(4.9)

where $m$ is a mapping network and $g$ is a gating network. Such a gated sum can model injective multiset functions [168], and is invariant to input order. To model the injective update function $U$, we can use a gated recurrent unit (GRU) [26], with $h_v^{in}$ treated as the input hidden state:

$$h_v = \text{GRU}_e(x_v, h_v^{in}).$$

(4.10)

Here the subscript $e$ denotes “encoding”.

Using a GRU also allows reducing our framework to traditional sequence to sequence modeling frameworks [153], as discussed in 4.3.3.

The above aggregation and update functions can be used to encode general computation graphs. For neural architectures, depending on how the outputs of multiple previous layers are aggregated as the input to a next layer, we will make a modification to (4.9), which is discussed in section 4.3.4. For Bayesian networks, we also make some modifications to their encoding due to the special d-separation properties of Bayesian networks, which is discussed in section 4.3.5.

### 4.3.2 Decoding

We now describe how D-VAE decodes latent vectors to DAGs (the generative part). The D-VAE decoder uses the same asynchronous message passing scheme as in the encoder to learn intermediate node and graph states. Similar to (4.10), the decoder uses another GRU,
denoted by GRU_d, to update node hidden states during the generation. Given the latent vector z to decode, we first use an MLP to map z to h_0 as the initial hidden state to be fed to GRU_d. Then, the decoder constructs a DAG node by node. For the i^{th} generated node v_i, the following steps are performed:

1. Compute v_i’s type distribution using an MLP f_{add_vertex} (followed by a softmax) based on the current graph state h_G := h_{v_{i-1}}.

2. Sample v_i’s type. If the sampled type is the ending type, stop the decoding, connect all loose ends (nodes without successors) to v_i, and output the DAG; otherwise, continue the generation.

3. Update v_i’s hidden state by h_{v_i} = GRU_d(x_{v_i}, h_{in}^{v_i}), where h_{in}^{v_i} = h_0 if i = 1; otherwise, h_{in}^{v_i} is the aggregated message from its predecessors’ hidden states given by equation (4.9).

4. For j=i−1, i−2, . . . , 1: (a) compute the edge probability of (v_j, v_i) using an MLP f_{add_edge} based on h_{v_j} and h_{v_i}; (b) sample the edge; and (c) if a new edge is added, update h_{v_i} using step 3.

The above steps are iteratively applied to each new generated node, until step 2 samples the ending type. For every new node, we first predict its node type based on the current graph state, and then sequentially predict whether each existing node has a directed edge to it based on the existing and current nodes’ hidden states. Figure 4.3 illustrates this process. Since edges always point to new nodes, the generated graph is guaranteed to be acyclic. Note that
we maintain hidden states for both the current node and existing nodes, and keep updating them during the generation. For example, whenever step 4 samples a new edge between \(v_j\) and \(v_i\), we will update \(h_{v_i}\) to reflect the change of its predecessors and thus the change of the computation so far. Then, we will use the new \(h_{v_i}\) for the next prediction. Such a dynamic updating scheme is flexible, computation-aware, and always uses the up-to-date state of each node to predict next steps. In contrast, methods based on RNNs [88, 175] do not maintain states for old nodes, and only use the current RNN state to predict the next step.

In step 4, when sequentially predicting incoming edges from previous nodes, we choose the reversed order \(i - 1, \ldots, 1\) instead of \(1, \ldots, i - 1\) or any other order. This is based on the prior knowledge that a new node \(v_i\) is more likely to firstly connect from the node \(v_{i-1}\) immediately before it. For example, in neural architecture design, when adding a new layer, we often first connect it from the last added layer, and then decide whether there should be skip connections from other previous layers. Note that however, such an order is not fixed and can be flexible according to specific applications.

### 4.3.3 Model extensions

**Relation with RNNs.** The D-VAE encoder and decoder can be reduced to ordinary RNNs when the input DAGs are reduced to linked lists. Although we propose D-VAE from a GNN’s perspective, our model can also be seen as a generalization of traditional sequence modeling frameworks [16, 153] where a timestamp depends only on the timestamp immediately before it, to the DAG case where a timestamp has multiple previous dependencies. As special DAGs, similar ideas have been explored for trees [69, 154], where a node can have multiple incoming edges yet only one outgoing edge.
**Bidirectional encoding.** D-VAE’s encoding process can be seen as simulating how an input signal goes through a DAG, with $h_v$ simulating the output signal at each node $v$. This is also known as *forward propagation* in neural networks. Inspired by the bidirectional RNN [139], we can also use another GRU to reversely encode a DAG (i.e., reverse all edge directions and encode the DAG again), thus simulating the *backward propagation* too. After reverse encoding, we get two ending states, which are concatenated and linearly mapped to their original size as the final output state. We find this bidirectional encoding can increase the performance and convergence speed on neural architectures.

**Incorporating vertex semantics.** Note that D-VAE currently uses one-hot encoding of node types as $x_v$, which does not consider the semantic meanings of different node types. For example, a $3 \times 3$ convolution layer might be functionally very similar to a $5 \times 5$ convolution layer, while being functionally distinct from a max pooling layer. We expect incorporating such semantic meanings of node types to be able to further improve D-VAE’s performance. For example, we can use pretrained embeddings of node types to replace the one-hot encoding. We leave it for future work.

### 4.3.4 Encoding neural architectures

According to Theorem 2.2, to ensure D-VAE injectively encodes computations, we need the aggregation function $A$ to be injective. Remember $A$ takes the multiset $\{h_u : u \rightarrow v\}$ as input. If the order of its elements does not matter, then the gated sum in (4.9) can model this injective multiset function without issues. However, if the order matters (i.e., permuting the elements of $\{h_u : u \rightarrow v\}$ makes $A$ output different results), we need a different aggregation function that can encode such orders.
Whether the order should matter for $\mathcal{A}$ depends on whether the input order matters for the operations $o$ (see the proof for Theorem 2.2 for more details). For example, if multiple previous layers’ outputs are summed or averaged as the input to a next layer in the neural networks, then $\mathcal{A}$ can be modeled by the gated sum in (4.9) as the order of inputs does not matter. However, if these outputs are concatenated as the next layer’s input, then the order does matter. In our experiments, the neural architectures use the second way to aggregate outputs from previous layers. The order of concatenation depends on a global order of the layers in a neural architecture. For example, if layer-2 and layer-4’s outputs are input to layer-5, then layer-2’s output will be before layer-4’s output in their concatenation.

Since the gated sum in (4.9) can only handle the unordered case, we can slightly modify (4.9) in order to make it order-aware thus more suitable for our neural architectures. Our scheme is as follows:

$$h_{\text{in}}^v = \sum_{u \rightarrow v} g(\text{Concat}(h_u, x_{uid})) \odot m(\text{Concat}(h_u, x_{uid})), \quad (4.11)$$

where $x_{uid}$ is the one-hot encoding of layer $u$’s global ID (1,2,3,...). Such an aggregation function respects the concatenation order of the layers. We empirically observed that this aggregation function can increase D-VAE’s performance on neural architectures compared to the plain aggregation function (4.9). However, even using (4.9) still outperformed all baselines.
4.3.5 Encoding Bayesian networks

We also make some modifications when encoding Bayesian networks. One modification is that the aggregation function \( (4.9) \) is changed to:

\[
\mathbf{h}_v^{\text{in}} = \sum_{u \rightarrow v} g(x_u) \odot m(x_u). \tag{4.12}
\]

Compared to (4.9), we replace \( h_u \) with the node type feature \( x_u \). This is due to the differences between computations on a neural architecture and on a Bayesian network. In a neural network, the signal flow follows the network architecture, where the output signal of a layer is fed as the input signals to its succeeding layers. Also in a neural network, what we are interested in is the result output by the final layer. In contrast, for a Bayesian network, the graph represents a set of conditional dependencies among variables instead of a computational flow. In particular, for Bayesian network structure learning, we are often concerned about computing the (log) marginal likelihood score of a dataset given a graph structure, which is often decomposed into individual variables given their parents (see Definition 18.2 in [80]). For example, in Figure 4.4, the overall score can be decomposed into \( s(X_1) + s(X_2) + s(X_3 \mid X_1, X_2) + s(X_4) + s(X_5 \mid X_3, X_4) \). To compute the score \( s(X_5 \mid X_3, X_4) \) for \( X_5 \), we only need the values of \( X_3 \) and \( X_4 \); its grandparents \( X_1 \) and \( X_2 \) should have no influence on \( X_5 \). Based on this intuition, when computing the hidden state of a node, we

![Figure 4.4: An example Bayesian network and its encoding.](image-url)
use the features \( x_u \) of its parents \( u \) instead of \( h_u \), which “d-separates” the node from its grandparents. For the update function, we still use (4.10).

Also based on the decomposibility of the score, we make another modification for encoding Bayesian networks by using the sum of all node states as the final output state instead of only using the ending node state. Similarly, when decoding Bayesian networks, the graph state \( h_G := \sum_{j=1,\ldots,i-1} h_{v_j} \).

Note that the combination of (4.12) and (4.10) can injectively model the conditional dependence between \( v \) and its parents \( u \). In addition, using summing can model injective set functions [168, Lemma 5]. Therefore, the above encoding scheme is able to injectively encode the complete conditional dependencies of a Bayesian network, thus also the overall score function \( s \) of the network.

### 4.3.6 Advantages of Encoding Computations for DAG Optimization

Here we discuss why D-VAE’s ability to injectively encode computations (Theorem 2.2) is of great benefit to performing DAG optimization in the latent space. Firstly, our target is to find a DAG that achieves high performance (e.g., accuracy of neural network, BIC score of Bayesian network) on a given dataset. The performance of a DAG is directly related to its computation. For example, given the same set of layer parameters, two neural networks with the same computation will have the same performance on a given test set. Since D-VAE encodes computations instead of structures, it allows embedding DAGs with similar performances to the same regions in the latent space, rather than embedding DAGs with merely similar structure patterns to the same regions. Subsequently, the latent space can be smooth w.r.t. performance instead of structure. Such smoothness can greatly facilitate
searching for high-performance DAGs in the latent space, since similar-performance DAGs tend to locate near each other in the latent space instead of locating randomly, and modeling a smoothly-changing performance surface is much easier.

Note that Theorem 2.2 is a necessary condition for the latent space to be smooth w.r.t. performance, because if D-VAE cannot injectively encode computations, it might map two DAGs representing completely different computations to the same encoding, making this point of the latent space arbitrarily unsmooth. Although there yet is no theoretical guarantee that the latent space must be smooth w.r.t. DAGs’ performances, we do empirically observe that the predictive performance and Bayesian optimization performance of D-VAE’s latent space are significantly better than those of baselines, which is indirect evidence that D-VAE’s latent space is smoother w.r.t. performance. Our visualization results also confirm the smoothness. See Section 4.4.2, 4.4.3, 4.4.4 for details.

4.4 Experiments

We validate the proposed DAG variational autoencoder (D-VAE) on two DAG optimization tasks:

- **Neural architecture search.** Our neural network dataset contains 19,020 neural architectures from the ENAS software [128]. Each neural architecture has 6 layers (excluding input and output layers) sampled from: \(3 \times 3\) and \(5 \times 5\) convolutions, \(3 \times 3\) and \(5 \times 5\) depthwise-separable convolutions [30], \(3 \times 3\) max pooling, and \(3 \times 3\) average pooling. We evaluate each neural architecture’s weight-sharing accuracy [128] (a proxy of the true accuracy) on CIFAR-10 [86] as its performance measure. We split the dataset into 90% training and 10% held-out test sets. We use the training set for VAE training, and use the test set only for evaluation. More details are in Appendix A.3.
- **Bayesian network structure learning.** Our Bayesian network dataset contains 200,000 random 8-node Bayesian networks from the bnlearn package [140] in R. For each network, we compute the Bayesian Information Criterion (BIC) score to measure the performance of the network structure for fitting the Asia dataset [90]. We split the Bayesian networks into 90% training and 10% test sets. For more details, please refer to Appendix A.4.

Following [88], we do four experiments for each task:

- **Basic abilities of VAE models.** In this experiment, we perform standard tests to evaluate the reconstructive and generative abilities of a VAE model for DAGs, including reconstruction accuracy, prior validity, uniqueness and novelty.

- **Predictive performance of latent representation.** We test how well we can use the latent embeddings of neural architectures and Bayesian networks to predict their performances.

- **Bayesian optimization.** This is the motivating application of D-VAE. We test how well the learned latent space can be used for searching for high-performance DAGs through Bayesian optimization.

- **Latent space visualization.** We visualize the latent space to qualitatively evaluate its smoothness.

Since there is little previous work on DAG generation, we compare D-VAE with three generative baselines adapted for DAGs: S-VAE, GraphRNN and GCN. Among them, S-VAE [16] and GraphRNN [175] are adjacency-matrix-based methods, and GCN [77] uses simultaneous message passing to encode DAGs. We include more details about these baselines and discuss D-VAE’s advantages over them in Appendix A.5. The training details are in Appendix A.6. The link to our code is https://github.com/muhanzhang/D-VAE.
4.4.1 Reconstruction accuracy, prior validity, uniqueness and novelty

Being able to accurately reconstruct input examples and generate valid new examples are basic requirements for VAE models. In this experiment, we evaluate the models by measuring 1) how often they can reconstruct input DAGs perfectly (Accuracy), 2) how often they can generate valid neural architectures or Bayesian networks from the prior distribution (Validity), 3) the portion of unique DAGs out of the valid generations (Uniqueness), and 4) the portion of valid generations that are never seen in the training set (Novelty).

We first evaluate each model’s reconstruction accuracy on the test sets. Following previous work [69, 88], we regard the encoding as a stochastic process. That is, after getting the mean and variance parameters of the posterior approximation $q_\phi(z|G)$, we sample a $z$ from it as $G$’s latent vector. To estimate the reconstruction accuracy, we sample $z$ 10 times for each $G$, and decode each $z$ 10 times too. Then we report the average portion of the 100 decoded DAGs that are identical to the input.

To calculate prior validity, we sample 1,000 latent vectors $z$ from the prior distribution $p(z)$ and decode each latent vector 10 times. Then we report the portion of these 10,000 generated DAGs that are valid. A generated DAG is valid if it can be read by the original software which generated the training data. More details about the validity experiment are in Appendix A.7.

We show the results in Table 4.1. Among all the models, D-VAE and S-VAE generally have the highest performance. We find that D-VAE, S-VAE and GraphRNN all have near perfect reconstruction accuracy, prior validity and novelty. However, D-VAE and S-VAE show higher uniqueness, meaning that they generate more diverse examples. We find that GCN is not suitable for modeling neural architectures as it only reconstructs 5.42% unseen inputs. This
Table 4.1: Reconstruction accuracy, prior validity, uniqueness and novelty (%).

<table>
<thead>
<tr>
<th>Methods</th>
<th>Neural architectures</th>
<th>Bayesian networks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accuracy</td>
<td>Validity</td>
</tr>
<tr>
<td>D-VAE</td>
<td>99.96</td>
<td>100.00</td>
</tr>
<tr>
<td>S-VAE</td>
<td>99.98</td>
<td>100.00</td>
</tr>
<tr>
<td>GraphRNN</td>
<td>99.85</td>
<td>99.84</td>
</tr>
<tr>
<td>GCN</td>
<td>5.42</td>
<td>99.37</td>
</tr>
</tbody>
</table>

is not surprising, since the simultaneous message passing scheme in GCN focuses on learning local graph structures, but fails to encode the computation represented by the entire neural network. Besides, the sum pooling after the message passing might also lose some global topology information which is important for the reconstruction.

4.4.2 Predictive performance of latent representation.

In this experiment, we evaluate how well the learned latent embeddings can predict the corresponding DAGs’ performances, which tests a VAE’s unsupervised representation learning ability. Being able to accurately predict a latent point’s performance also makes it much easier to search for high-performance points in this latent space. Thus, the experiment is also an indirect way to evaluate a VAE latent space’s suitability for DAG optimization. Following [88], we train a sparse Gaussian Process (SGP) regression model [148] with 500 inducing points on the training data’s embeddings to predict the unseen test data’s performances. We include the SGP training details in Appendix A.8.

We use two metrics to evaluate the predictive performance of the latent embeddings (given by the mean of the posterior approximations). One is the RMSE between the SGP predictions and the true performances. The other is the Pearson correlation coefficient (or Pearson’s $r$), measuring how well the prediction and real performance tend to go up and down together. A small RMSE and a large Pearson’s $r$ indicate a better predictive performance. Table 4.2

122
Table 4.2: Predictive performance of encoded means.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Neural architectures</th>
<th>Bayesian networks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RMSE</td>
<td>Pearson’s $r$</td>
</tr>
<tr>
<td>D-VAE</td>
<td>0.384±0.002</td>
<td>0.920±0.001</td>
</tr>
<tr>
<td>S-VAE</td>
<td>0.478±0.002</td>
<td>0.873±0.001</td>
</tr>
<tr>
<td>GraphRNN</td>
<td>0.726±0.002</td>
<td>0.669±0.001</td>
</tr>
<tr>
<td>GCN</td>
<td>0.832±0.001</td>
<td>0.527±0.001</td>
</tr>
</tbody>
</table>

shows the results. All the experiments are repeated 10 times and the means and standard deviations are reported.

From Table 4.2, we find that both the RMSE and Pearson’s $r$ of D-VAE are significantly better than those of the other models. A possible explanation is D-VAE encodes the computation, which is directly related to a DAG’s performance. S-VAE follows closely by achieving the second best performance. GraphRNN and GCN have less satisfying performances in this experiment. The better predictive power of D-VAE’s latent space means performing Bayesian optimization in it may be more likely to find high-performance points.

4.4.3 Bayesian optimization

We perform Bayesian optimization using the two best models, D-VAE and S-VAE, validated by previous experiments. Based on the SGP model from the last experiment, we perform 10 iterations of batch Bayesian optimization, and average results across 10 trials. A batch size of 50 and the expected improvement (EI) heuristic [71] are used, following [88]. Concretely speaking, we start from the training data’s embeddings, and iteratively propose new points from the latent space that maximize the EI acquisition function. For each batch of selected points, we evaluate their decoded DAGs’ real performances and add them back to the SGP
to select the next batch. Finally, we check the best-performing DAGs found by each model to evaluate its DAG optimization performance.

Neural architectures. For neural architectures, we select the top 15 found architectures in terms of their weight-sharing accuracies, and fully train them on CIFAR-10’s train set to evaluate their true test accuracies. More details can be found in Appendix A.3. We show the 5 architectures with the highest true test accuracies in Figure 4.5. As we can see, D-VAE in general found much better neural architectures than S-VAE. Among the selected architectures, D-VAE achieved a highest accuracy of 94.80%, while S-VAE’s highest accuracy was only 92.79%. In addition, all the 5 architectures of D-VAE have accuracies higher than 94%, indicating that D-VAE’s latent space can stably find many high-performance architectures. Although not outperforming state-of-the-art NAS techniques such as NAONet [103] (2.11% error rate on CIFAR-10), our search space was much smaller, and we did not apply any data augmentation techniques nor did we copy multiple folds or add more filters after finding the architecture. We emphasize that in this chapter, we mainly focus on idea illustration rather than record breaking, since achieving state-of-the-art NAS results typically requires enormous computation resources beyond our capability. Nevertheless, D-VAE does provide a promising new direction for neural architecture search based on graph generation, alternative to existing approaches.
Bayesian networks. We similarly report the top 5 Bayesian networks found by each model ranked by their BIC scores in Figure 4.6. D-VAE generally found better Bayesian networks than S-VAE. The best Bayesian network found by D-VAE achieved a BIC of -11125.75, which is better than the best network in the training set with a BIC of -11141.89 (a higher BIC score is better). Considering BIC is in log scale, the probability of our found network to explain the data is actually 1E7 times larger than that of the best training network. For reference, the true Bayesian network used to generate the Asia data has a BIC of -11109.74. Although we did not exactly find the true network, our found network is close to it and outperforms all training data. Our experiments show that searching in an embedding space is a promising direction for Bayesian network structure learning.

Bayesian optimization vs. random search. To validate that Bayesian optimization (BO) in the latent space does provide guidance in searching better DAGs, we compare BO with Random (which randomly samples points from the latent space of D-VAE). Figure 4.7 and 4.8 show the results (averaged across 10 trials). In each figure, the left plot shows the average performance of all the points found in each BO round, and the right plot shows the highest performance of all the points found so far. As we can see, BO consistently selects points with better average performance in each round than random search, which is expected. However, for the highest performance results, BO tends to fall behind Random in the initial few rounds. This might be because our batch expected improvement heuristic aims to take advantage of the currently most promising regions by selecting most points of the batch in the same region (exploitation), while Random more evenly explores the entire
Figure 4.7: Comparing BO with random search on neural architectures. Left: average weight-sharing accuracy of the selected points in each iteration. Right: highest weight-sharing accuracy of the selected points over time.

space (exploration). Nevertheless, BO seems to quickly catch up after a few rounds and shows long-term advantages.

4.4.4 Latent space visualization

In this experiment, we visualize the latent spaces of the VAE models to get a sense of their smoothness.

For neural architectures, we visualize the decoded architectures from points along a great circle in the latent space. We start from the latent embedding of a straight network without skip connections. Imagine this point as a point on the surface of a sphere (visualize the earth). We randomly pick a great circle starting from this point and returning to itself around the sphere. Along this circle, we evenly pick 35 points and visualize their decoded nets in Figure 4.9. As we can see, both D-VAE and S-VAE show relatively smooth interpolations by changing only a few node types or edges each time. Visually speaking, S-VAE’s structural changes are even more smooth. This is because S-VAE treats DAGs purely as strings, thus tending to embed DAGs with few differences in string representations to similar regions of the latent
Figure 4.8: Comparing BO with random search on Bayesian networks. Left: average BIC score of the selected points in each iteration. Right: highest BIC score of the selected points over time.

Figure 4.9: Great circle interpolation starting from a point and returning to itself. Upper: D-VAE. Lower: S-VAE.

space without considering their computational differences (see Appendix A.5). In contrast, D-VAE models computations, and focuses more on the smoothness w.r.t. computation rather than structure.

For Bayesian networks, we aim to directly visualize the BIC score distribution of the latent space. To do so, we reduce its dimensionality by choosing a 2-D subspace of the latent space spanned by the first two principal components of the training data’s embeddings. In this low-dimensional subspace, we compute the BIC scores of all the points evenly spaced within a $[-0.3, 0.3]$ grid and visualize the scores using a colormap in Figure 4.10. As we can see, D-VAE seems to better differentiate high-score points from low-score ones and shows more
smoothly changing of BIC scores, while S-VAE shows sharp boundaries and seems to mix high-score and low-score points more severely. We suspect this helps Bayesian optimization find high-performance Bayesian networks more easily in D-VAE.

We include more visualization results of the generated neural architectures and Bayesian networks in appendix A.9 and A.10.

4.5 Conclusion

In this chapter, we have proposed D-VAE, a deep generative model for DAGs. D-VAE uses a novel asynchronous message passing scheme to explicitly model computations on DAGs. By performing Bayesian optimization in D-VAE’s latent spaces, we offer promising new directions to two important problems, neural architecture search and Bayesian network structure learning. We hope D-VAE can inspire more research on extending graph generative models’ applications on structure optimization.
Chapter 5

Conclusions

Graph learning has received much attention among the machine learning and data mining communities due to the abundance and importance of graph structured data in the real world. Deep neural networks, on the other hand, have achieved remarkable performance on numerous machine learning tasks and are continuously changing people’s lives. In this dissertation, we study an emerging new field, graph deep learning, which aims at applying deep learning, in particular graph neural networks (GNNs), to a series of graph learning problems. Through innovations in algorithms, architectures, theories, and applications, we have greatly extended GNNs’ boundary to a broad range of graph learning problems, including graph classification, medical ontology embedding, link prediction, recommender systems, graph generation, and graph structure optimization.

For graph classification, we identify that existing GNNs’ less satisfying performance results from the summing-based pooling operation that aggregates node features into a graph representation in one step. To address this issue, we have proposed an end-to-end GNN architecture with a novel SortPooling layer to replace the summing in previous works. Inspired
by CNN’s successes on images, we designed to also give nodes an order and apply convolutional filters on the ordered node sequences like convolutional filters on image pixels. Our SortPooling layer sorts node features using the soft WL colors extracted by previous graph convolution layers. With this meaningful and consistent order, we are able to apply 1-D convolutional filters to the sorted node sequences and hierarchically extract graph representations in a more refined way. Experimental results have demonstrated the superior graph classification performance compared to existing methods. Our work has since inspired many follow-up works studying how to better aggregate node features into graph representations [49, 91, 173], and has become a common baseline for graph classification.

For medical ontology embedding, we have proposed a novel hierarchical attention propagation model to absorb structural information from the entire medical ontology into the node embeddings. We have proved our medical concept embeddings’ superior expressive power in terms of encoding and recovering the ontology. We have successfully used the learned embeddings on two sequential procedure/diagnosis prediction tasks using real patient data and achieved superior prediction performance. Our work on medical ontology embedding has extended GNNs’ applications to healthcare domains. We expect it will encourage more research on studying the various types of graph-structured data in medical domains, such as medical ontologies and patient networks.

For link prediction, we have developed a $\gamma$-decaying theory, which unifies existing link prediction heuristics into a single framework and proves their local approximability. In particular, we have proved that three most popular high-order heuristics, Katz index, rooted PageRank and SimRank, intrinsically share the same form, and can all be approximated from local enclosing subgraphs with the approximation error decreasing exponentially with the hop number. The theory inspired us to develop SEAL, a framework to learn general graph structure features for link prediction from local enclosing subgraphs around links based on a
GNN, instead of using predefined heuristics. The learned graph structure features outperform traditional heuristics dramatically. Our ground-breaking work has been gradually shifting the trend of designing link prediction heuristics manually towards learning graph structural features automatically from graphs.

For recommender systems, we have leveraged a similar framework to SEAL to formulate matrix completion as predicting labeled links in a bipartite network. Through a novel GNN architecture and an effective regularization technique, our model achieved highly competitive results with traditional matrix factorization baselines. Further, we have demonstrated that our model is inductive and transferrable without using any content information, a property which previous inductive matrix completion methods do not have. Experiments have shown that our model trained on MovieLens can be directly used to predict Douban movie ratings with performance even superior than many baseline methods trained exclusively for Douban. This transfer learning ability has great potential for recommendation tasks lack of enough training data. We expect to see our work make more impact and encourage more new ideas for recommender systems in the future.

Lastly, we have explored GNNs’ applicability for graph generation and graph structure optimization. Focusing on directed acyclic graphs (DAGs), we have proposed a DAG variational autoencoder (D-VAE) model that can encode and decode DAGs to and from a latent space based on GNNs. To better encode the computations on DAGs, we have designed a novel asynchronous message passing scheme that can injectively map computations to the latent space, which transforms the hard discrete structure optimization problem into an easier continuous space optimization problem. We have tested D-VAE on two types of common DAGs in machine learning, neural architectures and Bayesian networks. Experimental results show that our D-VAE not only generates novel and valid DAGs, but also produces a smooth latent space effective for performing Bayesian optimization to optimize DAG structures on.
Our model has achieved very promising results on two important DAG optimization tasks, neural architecture search and Bayesian network structure learning. We hope our work can provide an orthogonal idea to existing algorithms of both problems and inspire more research on studying graph generative models based on GNNs.

We believe graphs are a very important type of data to study, not only because there exist various interesting and important graph learning problems, but also because graphs naturally model our densely connected, competitive and collaborative world. We also believe studying graph neural networks is a crucial step towards strong AI, as our human brain is also a connected graph which can leverage related memories to learn new things quickly. In the future, we plan to keep investigating graphs and GNNs. In particular, we plan to apply GNNs to model brain activities, which could potentially help us understand brains better, and perhaps eventually, simulate human brains.
References


Appendix A

A.1 Additional details about link prediction baselines

Hyperparameters of heuristic and latent feature methods Most hyperparameters are inherited from the original paper of each method. For Katz, we set the damping factor $\beta$ to 0.001. For PageRank, we set the damping factor $\alpha$ to 0.85. For SimRank, we set $\gamma$ to 0.8. For stochastic block model (SBM), we use the implementation of [4] using a latent group number 12. For matrix factorization (MF), we use the libFM [132] software with the default parameters. For node2vec, LINE, and spectral clustering, we first generate 128-dimensional embeddings from the observed networks with default parameters of each software. Then, we use the Hadamard product of two nodes’ embeddings as a link’s embedding as suggested in [56], and train a logistic regression model with Liblinear [46] using automatic hyperparameter selection. For VGAE, we use its default setting.

WLNM. Weisfeiler-Lehman Neural Machine (WLNM) [182] is our previously proposed link prediction method that learns general graph structure features. It achieves state-of-the-art performance on various networks, outperforming all handcrafted heuristics. WLNM has three steps: enclosing subgraph extraction, subgraph pattern encoding, and neural network
training. In the enclosing subgraph extraction step: for each node pair \((x, y)\), WLNM iteratively extracts \(x\) and \(y\)’s one-hop neighbors, two-hop neighbors, and so on, until the enclosing subgraph has \textbf{more than} \(K\) vertices, where \(K\) is a user-defined integer. In the subgraph pattern encoding step, WLNM uses the Weisfeiler-Lehman algorithm to define an order for nodes within each enclosing subgraph, so that the neural network can read different subgraphs’ nodes in a consistent order and learn meaningful patterns. To unify the sizes of the enclosing subgraphs, after getting the vertex order, the last few vertices are deleted so that all the truncated enclosing subgraphs have the same size \(K\). These truncated enclosing subgraphs are reordered and their fixed-size adjacency matrices are fed into the fully-connected neural network to train a link prediction model. Due to the truncation, WLNM cannot consistently learn from each link’s full \(h\)-hop neighborhood. The loss of structural information limits WLNM’s performance and restrict it from learning complete \(h\)-order graph structure features. Following [182], we use \(K = 10\) (the best performing \(K\)) in our experiments.

\textbf{WLK.} Weisfeiler-Lehman graph kernel (WLK) [142] is a state-of-the-art graph kernel. Graph kernels make kernel machines feasible for graph classification by defining some positive semidefinite graph similarity scores. Most graph kernels measure graph similarity by decomposing graphs into small substructures and adding up the pair-wise similarities between these components. Common types of substructures include walks [152, 159], subgraphs [34, 85], paths [15], and subtrees [116, 142]. WLK is based on counting common rooted subtrees between two graphs. In our experiments, we train a SVM on the WL kernel matrix. We feed the same enclosing subgraphs as in SEAL to WLK. We search the subtree depth from \(\{0, 1, 2, 3, 4, 5\}\) on 10\% validation links. WLK does not support continuous node information, but supports integer node labels. Thus, we feed the same structural node labels from (3.10) to WLK too.
We compare the characteristics of different link prediction methods in Table A.1.

### Table A.1: Comparison of different link prediction methods

<table>
<thead>
<tr>
<th></th>
<th>Heuristics</th>
<th>Latent features</th>
<th>WLK</th>
<th>WLN</th>
<th>SEAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph structure features</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Learn from full ( h )-hop</td>
<td>No</td>
<td>n/a</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Latent/explicit features</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Model</td>
<td>n/a</td>
<td>LR/inner product</td>
<td>SVM</td>
<td>NN</td>
<td>GNN</td>
</tr>
</tbody>
</table>

### A.2 More related work on neural architecture search and Bayesian network structure learning

Both neural architecture search (NAS) and Bayesian network structure learning (BNSL) are subfields of AutoML. See [190] for a survey. We have given a brief overview of NAS and BNSL in section 4.2. Below we discuss several works most related to our work in detail.

[103] proposed a novel NAS approach called Neural Architecture Optimization (NAO). The basic idea is to jointly learn an encoder-decoder between networks and a continuous space, and also a performance predictor \( f \) that maps the continuous representation of a network to its performance on a given dataset; then they perform two or three iterations of gradient descent on \( f \) to find better architectures in the continuous space, which are then decoded to real networks to evaluate. This methodology is similar to that of [54] and [69] for molecule optimization; also similar to [115] for slightly revising a sentence.

There are several key differences comparing to our approach. First, they use strings (e.g. “node-2 conv 3x3 node1 max-pooling 3x3”) to represent neural architectures, whereas we directly use graph representations, which is more natural, and generally applicable to other...
graphs such as Bayesian network structures. Second, they use supervised learning instead of unsupervised learning. That means they need to first evaluate a considerable amount of randomly sampled graphs on a typically large dataset (e.g. train many neural networks), and use these results to supervise the training of the autoencoder. Given a new dataset, the autoencoder needs to be completely retrained. In contrast, we train our variational autoencoder in a fully unsupervised manner, so the model is of general purposes.

[48] proposed a novel AutoML algorithm also using model embedding, but with a matrix factorization approach. They first construct a matrix of performances of thousands of ML pipelines on hundreds of datasets; then they use a probabilistic matrix factorization to get the latent representations of the pipelines. Given a new dataset, Bayesian optimization with the expected improvement heuristic is used to find the best pipeline. This approach only allows us to choose from predefined off-the-shelf ML models, hence its flexibility is somewhat limited.

[73] use Bayesian optimization for NAS; they define a kernel that measures the similarities between networks by solving an optimal transport problem, and in each iteration, they use some evolutionary heuristics to generate a set of candidate networks based on making small modifications to existing networks, and use expected improvement to choose the next one to evaluate. This work is similar to ours in the application of Bayesian optimization. However, defining a kernel to measure the similarities between discrete structures is a non-trivial problem. In addition, the discrete search space is heuristically extrapolated near existing architectures, which makes the search essentially local. In contrast, we directly fit a Gaussian process over the entire continuous latent space, enabling more global optimization.

Using Gaussian process (GP) for Bayesian network structure learning has also been studied before. [170] analyzed the smoothness of BDe score, showing that a local change (e.g. adding
an edge) can change the score by at most $O(\log n)$, where $n$ is the number of training points. They proposed to use GP as a proxy for the score to accelerate the search. [6] used GP to model the BDe score, and showed that the probability of improvement is higher than that of using hill climbing to guide the local search. However, these methods still heuristically and locally operate in the discrete space, whereas our latent space makes both local and global methods such as gradient descent and Bayesian optimization applicable in a principled manner.

A.3 More details about neural architecture search

We use the efficient neural architecture search (ENAS)’s software [128] to generate the training and testing neural architectures. With these seed architectures, we can train a VAE model and thus search for new high-performance architectures in the latent space.

ENAS alternately trains two components: 1) a RNN-based controller which is used to propose new architectures, and 2) the shared weights of the proposed architectures. It uses a weight-sharing (WS) scheme to obtain a quick but rough estimate of how good an architecture is. That is, it forces all the proposed architectures to use the same set of shared weights, instead of fully training each neural network individually. It assumes that an architecture with a high validation accuracy using the shared weights (i.e., the weight-sharing accuracy) is more likely to have a high test accuracy after fully retraining its weights from scratch.

We first run ENAS in the macro space (section 2.3 of [128]) for 1000 epochs with 20 architectures proposed in each epoch. For all the proposed architectures excluding the first 1000 burn-in ones, we evaluate their weight-sharing accuracies using the shared weights from the last epoch. We further split the data into 90% training and 10% held-out test sets. Then our task becomes to train a VAE on the training neural architectures, and then generate
new high-performance architectures from the latent space based on Bayesian optimization. Note that our target performance measure here is the weight-sharing accuracy, not the true validation/test accuracy after fully retraining the architecture. This is because the weight-sharing accuracy takes around 0.5 second to evaluate, while fully training a network takes over 12 hours. In consideration of our limited computational resources, we choose the weight-sharing accuracy as our optimization target in the Bayesian optimization experiments.

After the Bayesian optimization finds a final set of architectures with high weight-sharing accuracies, we will fully train them to evaluate their true test accuracies on CIFAR-10. To fully train an architecture, we follow the original setting of ENAS to train each architecture on CIFAR-10’s training set for 310 epochs, and report the last epoch’s net’s test accuracy. See [128, section 3.2] for details.

Due to our constrained computational resources, we choose not to perform Bayesian optimization to optimize the true validation accuracy (after fully training), which would be a more principled way for searching neural architectures. Nevertheless, we describe its procedure here for future explorations: After training the D-VAE, we have no architectures at all to initialize a Gaussian process regression on the true validation accuracy. Thus, we need to randomly pick up some points in the latent space, decode them into neural architectures, and get their true validation accuracies after full training. Then with these initial points, we start the Bayesian optimization similarly to section 4.4.3, with the optimization target replaced by the true validation accuracy. Finally, we will find a set of architectures with the highest true validation accuracies, and report their true test accuracies. This experiment will take much longer time (months of GPU time). Thus, making the training parallel is very necessary.

One might wonder why we train another generative model after we already have ENAS. Firstly, ENAS is not general-purpose, but task specific. It leverages the validation accuracy
signals to train the controller based on reinforcement learning. For any new NAS tasks, ENAS needs to be completely retrained. In contrast, D-VAE is unsupervised. It only needs to be trained once, and can be applied to other NAS tasks. Secondly, D-VAE also provides a way to learn neural architecture embeddings, which can be used for downstream tasks such as visualization, classification, clustering etc.

In the Bayesian optimization experiments (section 4.4.3), the best architecture found by D-VAE achieves a test accuracy of 94.80% on CIFAR-10. Although not outperforming state-of-the-art NAS techniques which has an error rate of 2.11%, our architecture only contains 3 million parameters compared to the state-of-the-art NAONet + Cutout which has 128 million parameters [103]. In addition, NAONet used 200 GPUs to fully train 1,000 architectures for 1 day, and stacked the final found cell for 6 times as well as adding 4 times more filters after optimization. In comparison, we only used 1 GPU to evaluate the weight-sharing accuracy, and never used any data augmentation techniques or architecture stacking to boost the performance, since achieving new state-of-the-art NAS results (through using great resources and heavy engineering) are beyond the main purpose of our paper.

A.4 More details about Bayesian network structure learning

We consider a small synthetic problem called Asia [90] as our target Bayesian network structure learning problem. The Asia dataset is composed of 5,000 samples, each is generated by a true network with 8 binary variables\(^4\). Bayesian Information Criteria (BIC) score is used to evaluate how well a Bayesian network fits the 5,000 samples. To train a VAE model to generate Bayesian network structures, we sample 200,000 random 8-node Bayesian networks

\(^4\)http://www.bnlearn.com/documentation/man/asia.html
from the bnlearn package [140] in R, which are split into 90% training and 10% testing sets. Our task is to train a VAE model on the training Bayesian networks, and search in the latent space for Bayesian networks with high BIC scores using Bayesian optimization. In this task, we consider a simplified case where the topological order of the true network is known – we let the sampled training and test Bayesian networks have topological orders consistent with the true network of Asia. This is a reasonable assumption for many practical applications, e.g., when the variables have a temporal order [80]. When sampling a network, the probability of a node having an edge with a previous node (as specified by the order) is set to the default option \(2/(k - 1)\), where \(k = 8\) is the number of nodes, which results in sparse graphs where the number of edges is in the same order of the number of nodes.

A.5 Baselines for D-VAE

As discussed in the related work, there are other types of graph generative models that can potentially work for DAGs. We explore three possible approaches and contrast them with D-VAE.

**S-VAE.** The S-VAE baseline treats a DAG as a sequence of node strings, which we call string-based variational autoencoder (S-VAE). In S-VAE, each node is represented as the one-hot encoding of its type number concatenated with a 0/1 indicator vector indicating which previous nodes have directed edges to it (i.e., a column of the adjacency matrix). For example, suppose there are two node types and five nodes, then node 4’s string “0 1, 0 1 1 0 0” means this node has type 2, and has directed edges from previous nodes 2 and 3. S-VAE leverages a standard GRU-based RNN variational autoencoder [16] on the topologically sorted node sequences, with each node’s string treated as its input bit vector.
GraphRNN. One similar generative model is GraphRNN [175]. Different from S-VAE, it further decomposes an adjacency column into entries and generates the entries one by one using another edge-level GRU. GraphRNN is a pure generative model which does not have an encoder, thus cannot optimize DAG performance in a latent space. To compare with GraphRNN, we equip it with S-VAE’s encoder and use it as another baseline. Note that the original GraphRNN feeds nodes using a BFS order (for undirected graphs), yet we find that it is much worse than using a topological order here. Note also that although GraphRNN seems more expressive than S-VAE, we find that in our applications GraphRNN tends to have more severe overfitting and generates less diverse DAGs.

Both GraphRNN and S-VAE treat DAGs as bit strings and use RNNs to model them. This representation has several drawbacks. Firstly, since the topological ordering is often not unique for a DAG, there might be multiple string representations for the same DAG, which all result in different encoded representations. This will violate the permutation invariance in Theorem 4.1. Secondly, the string representations can be very brittle in terms of modeling DAGs’ computational purposes. In Figure A.1, the left and right DAGs’ string representations are only different by two bits, i.e., the edge (2,3) in the left is changed to the edge (1,3) in the right. However, the two bits of change in structure greatly changes the signal flow, which makes the right DAG always output 1. In S-VAE and GraphRNN, since the bit representations of the left and right DAGs are very similar, they are highly likely to be...
encoded to similar latent vectors. In particular, the only difference between encoding the left and right DAGs is that, for node 3, the encoder RNN will read an adjacency column of \([0, 1, 0, 0, 0, 0]\) in the left, and read \([1, 0, 0, 0, 0, 0]\) in the right, while all the remaining encoding is exactly the same. By embedding two DAGs serving very different computational purposes to the same region of the latent space, S-VAE and GraphRNN tend to have less smooth latent spaces which make optimization on them more difficult. In contrast, D-VAE can better differentiate such subtle differences, as the change of edge \((2,3)\) to \((1,3)\) completely changes what aggregated message node 3 receives in D-VAE (hidden state of node 2 vs. hidden state of node 1), which greatly affects node 3 and all its successors’ feature learning.

**GCN.** The graph convolutional network (GCN) [77] is one representative graph neural network with a simultaneous message passing scheme. In GCN, all the nodes take their neighbors’ incoming messages to update their own states simultaneously instead of following an order. After message passing, the summed node states is used as the graph state. We include GCN as the third baseline. Since GCN can only encode graphs, we equip GCN with D-VAE’s decoder to make it a VAE model.

Using GCN as the encoder can ensure permutation invariance, since node ordering does not matter in GCN. However, GCN’s message passing focuses on propagating the neighboring nodes’ features to each center node to encode the local substructure pattern around each node. In comparison, D-VAE’s message passing simulates how the computation is performed along the directed paths of a DAG and focuses on encoding the computation. Although learning local substructure features is essential for GCN’s successes in node classification and graph classification, here in our tasks, modeling the entire computation is much more important than modeling the local features. Encoding only local substructures may also lose important information about the global DAG topology, thus making it more difficult to reconstruct the DAG.
We omit other possible approaches such as GraphVAE [146], GraphSAGE [57], and other graph-based models [69, 95, 100, 174] etc., either because they share similar characteristics to the compared baselines, or because they lack official code or target specific graphs (such as molecules).

A.6 VAE training details

We use the same settings and hyperparameters (where applicable) for all the four models to be as pair as possible. Many hyperparameters are inherited from [88]. Single-layer GRUs are used in all models requiring recurrent units, with the same hidden state size of 501. We set the dimension of the latent space to be 56 for all models. All VAE models use $\mathcal{N}(0, I)$ as the prior distribution $p(z)$, and take $q_{\phi}(z|G)$ ($G$ denotes the input DAG) to be a normal distribution with a diagonal covariance matrix, whose mean and variance parameters are output by the encoder. The two MLPs used to output the mean and variance parameters are all implemented as single linear layer networks.

For the decoder network of D-VAE, we let $f_{\text{add}_\text{vertex}}$ and $f_{\text{add}_\text{edge}}$ be two-layer MLPs with ReLU nonlinearities, where the hidden layer sizes are set to two times of the input sizes. Softmax activation is used after $f_{\text{add}_\text{vertex}}$, and sigmoid activation is used after $f_{\text{add}_\text{edge}}$. For the gating network $g$, we use a single linear layer with sigmoid activation. For the mapping function $m$, we use a linear mapping without activation. The bidirectional encoding discussed in section 4.3.3 is enabled for D-VAE on neural architectures, and disabled for D-VAE on Bayesian networks and other models where it gets no better results. To measure the reconstruction loss, we use teacher forcing [69]: following the topological order with which the input DAG’s nodes are consumed, we sum the negative log-likelihood of each decoding step by forcing them to generate the ground truth node type or edge at each step. This
ensures that the model makes predictions based on the correct histories. Then, we optimize the VAE loss (the negative of (4.1)) using gradient descent following [69].

When optimizing the VAE loss, we use ReconstructLoss + αKLDivergence as the loss function. In original VAE framework, α is set to 1. However, we found that it led to poor reconstruction accuracies, similar to the findings of previous work [36, 69, 88]. Following the implementation of [69], we set α = 0.005. Mini-batch SGD with Adam optimizer [76] is used for all models. For neural architectures, we use a batch size of 32 and train all models for 300 epochs. For Bayesian networks, we use a batch size of 128 and train all models for 100 epochs. We use an initial learning rate of 1E-4, and multiply the learning rate by 0.1 whenever the training loss does not decrease for 10 epochs. We use PyTorch to implement all the models.

A.7 More details of the prior validity experiment

Since different models can have different levels of convergence w.r.t. the KLD loss in (4.1), their posterior distribution $q_\phi(z \mid x)$ may have different degrees of alignment with the prior distribution $p(z) = \mathcal{N}(0, I)$. If we evaluate prior validity by sampling from $p(z)$ for all models, we will favor those models that have a higher-level of KLD convergence. To remove such effects and focus purely on models’ intrinsic ability to generate valid DAGs, when evaluating prior validity, we apply $z = z \odot \text{std}(Z_{\text{train}}) + \text{mean}(Z_{\text{train}})$ for each model (where $Z_{\text{train}}$ are encoded means of the training data by the model), so that the latent vectors are scaled and shifted to the center of the training data’s embeddings. If we do not apply such transformations, we find that we can easily control the prior validity results by optimizing for more or less epochs or putting more or less weight on the KLD loss.

For a generated neural architecture to be read by ENAS, it has to pass the following validity checks: 1) It has one and only one starting node (the input layer); 2) It has one and only
one ending type (the output layer); 3) Other than the input node, there are no nodes which
do not have any predecessors (no isolated paths); 4) Other than the output node, there are
no nodes which do not have any successors (no blocked paths); 5) Each node must have a
directed edge from the node immediately before it (the constraint of ENAS), i.e., there is
always a main path connecting all the nodes; and 6) It is a DAG.

For a generated Bayesian network to be read by `bnlearn` and evaluated on the Asia dataset,
it has to pass the following validity checks: 1) It has exactly 8 nodes; 2) Each type in
'ASTLBEXD' appears exactly once; and 3) It is a DAG.

Note that the training graphs generated by the original software all satisfy these validity
constraints.

### A.8 SGP training details

We use sparse Gaussian process (SGP) regression as the predictive model. We use the open
sourced SGP implementation in [88]. Both the training and testing data’s performances
are standardized according to the mean and std of the training data’s performances before
feeding to the SGP. And the RMSE and Pearson’s $r$ in Table 4.2 are also calculated on the
standardized performances. We use the default Adam optimizer to train the SGP for 100
epochs constantly with a mini-batch size of 1,000 and learning rate of 5E-4.

For neural architectures, we use all the training data to train the SGP. For Bayesian networks,
we randomly sample 5,000 training examples each time, due to two reasons: 1) using all
the 180,000 examples to train the SGP might not be realistic for a typical scenario where
network/dataset is large and evaluating a network is expensive; and 2) we found using
a smaller sample of training data results in more stable BO performance due to the less
probability of duplicate rows which might result in ill conditioned matrices. Note also that, when training the variational autoencoders, all the training data are used, since the VAE training is purely unsupervised.

A.9 The generated neural architectures

We randomly pick a neural architecture and use its encoded mean as the starting point. We then generate two random orthogonal directions, and move in the combination of these two directions from the starting point to render a 2-D visualization of the decoded architectures in Figure A.2.

Figure A.2: 2-D visualization of decoded neural architectures. Left: D-VAE. Right: S-VAE.
A.10 The generated Bayesian networks

We similarly show the 2-D visualization of decoded Bayesian networks in Figure A.3. Both D-VAE and S-VAE show smooth latent spaces.

Figure A.3: 2-D visualization of decoded Bayesian networks. Left: D-VAE. Right: S-VAE.