On the Power of Message Passing for Learning on Graph-Structured Data

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Matthias Fey

Dortmund

2022

Tag der mündlichen Prüfung:	01.07.2022
Dekan / Dekanin:	Prof. Dr.-Ing. Gernot A. Fink
Gutachter / Gutachterinnen:	Priv.-Doz. Dr. Frank Weichert
	Ass.-Prof. Dr. Nils Morten Kriege

On the *Power* of **Message Passing** for *Learning* on **Graph-Structured Data**

Matthias Fey

Acknowledgements

The decision to pursue a PhD was one of the best decisions in my life. It gave me the opportunity to grow tremendously both in my knowledge and as a person. I am thankful to many people who made the last four years a truly enriching experience.

First and foremost, I would like to thank Frank Weichert for giving me the opportunity to pursue a PhD, your frequent advice and support, and for giving me the necessary freedom to follow a variety of research directions. Many thanks also go to Heinrich Müller, Petra Mutzel and Nils Kriege for co-advising me at the start of my PhD. I wish you all the best in retirement, in Bonn and in Vienna, respectively.

It has been a wonderful experience to be part of the chair of computer graphics, and I would like to thank all my colleagues. Special thanks go to Jan Eric Lenssen who constantly encouraged and motivated me, and who teached me to think "deep". Thank you for countless inspiring discussions and for being the most critical reviewer of all. You became a true friend over the years! Fortunately, we will still deal with each other moving forward. Further thanks go to Christopher Morris for the mutual support in our research. After all, Weisfeiler-Lehman have truly gone neural.

I also want to thank Jure Leskovec for taking me out of my bubble and putting me into the big wide world. You are a truly inspiring researcher and mentor. I am tremendously grateful for your continuous support and for giving me the opportunity to collaborate with fantastic researchers: Weihua Hu, Jiaxuan You, Rex Ying and Hongyu Ren. Final apologies for meeting at the craziest hours.

I am very grateful for many workshop invitations and the many opportunities to share and present my work. I am also extremely thankful to have won the lottery of exceptionally kind reviewer assignments. Furthermore, I am deeply impressed by the sheer amount of interest in my work, and I would like to thank everyone who got involved and reached out to me. Four years later, we truly have achieved something big and have advanced graph machine learning one step and one commit at a time.

Many thanks also go to my friends from both near and far: Arne, Simon, Mathis, Yannick, Haver, Daniel, Boro, Bastian, Janis, Tobias, Christian, Lukas, Annika, Isa, Steffi, Lennart, Marci, and others. Finally, I would also like to thank my family: To my brother Michael and to my parents Angelika and Norbert, thank you for your invaluable support and for always being or having been there for me.

Most importantly, I want to thank Sarah, the love of my life. I cannot thank you enough for your daily support and for enriching every day anew. I do not know where I would stand today without you. My biggest wish is to grow old with you, side by side.

You are my mountain
You are my sea

Abstract

This thesis proposes novel approaches for machine learning on irregularly structured input data such as graphs, point clouds and manifolds. Specifically, we are breaking up with the regularity restriction of conventional deep learning techniques, and propose solutions in designing, implementing and scaling up deep end-to-end representation learning on graph-structured data, known as *Graph Neural Networks* (*GNNs*).

Graph Neural Networks capture local graph structure and feature information by following a *neural message passing* scheme, in which node representations are recursively updated in a trainable and purely local fashion. We demonstrate the generality of message passing through a unified framework suitable for a wide range of operators and learning tasks. Our own specific GNN instantiation, the *Spline-Based Convolutional Neural Network* (*SplineCNN*), fits into this message passing scheme by conditioning messages via a continuous B-spline kernel formulation while resembling the traditional definition of Convolutional Neural Networks for discrete input.

We further analyze the limitations and inherent weaknesses of Graph Neural Networks and propose solutions to overcome them, both theoretically and in practice. For this, we relate the representational power of Graph Neural Networks to their ability to distinguish non-isomorphic (sub-)graphs, and leverage gained insights to propose a generalization of GNNs which allow them to reach maximal expressiveness. Additional solutions aim to enhance the power of Graph Neural Networks in task-specific scenarios. Specifically, we propose the *Dynamic Neighborhood Aggregation* (*DNA*) procedure, which allows for a selective and node-adaptive aggregation of neighbors from potentially differing locality, boosting model performance in heterophily graphs. Furthermore, we introduce the *Hierarchical Inter-Message Passing* (*HIMP*) architecture for learning on molecular graphs, in which its junction tree representation is used as a coarse-grained representation for exchanging messages between different hierarchies, leading to increased model power. Lastly, we present a two-stage neural *Deep Graph Matching Consensus* (*DGMC*) architecture for learning and refining structural correspondences between graphs. DGMC aims to reach a matching consensus in local neighborhoods by distributing global positional encodings, overcoming the limitations of locality in traditional GNNs.

In addition, we ensure that our proposed methods scale naturally to large input domains. In particular, we propose novel methods to eliminate the exponentially increasing dependency of nodes over layers inherent to message passing GNNs; a phenomenon framed as *neighbor explosion*. Specifically, while scalability techniques are indispensable for applying Graph Neural Networks to large graphs, alternative ap-

proaches based on graph sub-sampling weaken the expressive power of message passing. In order to overcome this restriction, we propose *GNNAutoScale* (*GAS*), a framework for scaling arbitrary message passing Graph Neural Networks to large graphs. GAS prunes entire sub-trees of the computation graph by utilizing historical node embeddings from prior training iterations. As a result, GAS is provably able to maintain the expressive power of the original architecture. Furthermore, we propose the *Open Graph Benchmark* (*OGB*) suite and organized the *Open Graph Benchmark Large-Scale Challenge* (*OGB-LSC*) in order to accelerate Graph Neural Network research on large-scale graphs. OGB includes a diverse set of challenging, realistic and large-scale graph benchmark datasets across three different learning tasks.

With the rise of Graph Neural Networks as a state-of-the-art technique for Graph Representation Learning, there also exists an urgent demand in both flexible and powerful libraries for accelerating research and putting existing models into production. However, meeting both requirements is challenging, as high GPU throughput needs to be achieved on highly sparse and irregular data of varying size across a wide range of different model implementations. Here, we introduce *PyTorch Geometric* (*PyG*), a deep learning library for implementing and working with graph-based neural network building blocks, built upon PyTorch. PyG leverages sparse GPU acceleration by providing dedicated CUDA kernels, and introduces efficient mini-batch handling for input examples of different size. In addition, PyG provides a general message passing interface to allow for rapid and clean prototyping of new research ideas. Further, we present *PyGAS*, an easy-to-use extension for PyG that converts common and custom GNN models into their scalable variants by utilizing our GAS framework. In particular, PyGAS optimizes the access pattern of historical embeddings in order to allow for both fast and memory-efficient mini-batch training.

Notation

Reference list of the most commonly used notation across this thesis:

Example	Explanation
	General
$\mathbb{N}^N$	The N-dimensional space of natural numbers
$\mathbb{R}^N$	The N-dimensional space of real numbers
x, y, z	A lowercase italic letter denotes a scalar
$\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z}$	A lowercase bold letter denotes a vector
$\boldsymbol{X}$, $\boldsymbol{Y}$, $\boldsymbol{Z}$	An uppercase bold letter denotes a matrix
$\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$	An uppercase calligraphic letter denotes a set
$\{x, y, z\}$	An unordered set
(x, y, z)	An ordered set
$\{\{x, y, y\}\}$	An unordered multiset
x_i	The i-th element of vector $\boldsymbol{x}$
$\boldsymbol{X}_{i,:}$ or $\boldsymbol{X}[i, :]$	The i-th row of matrix $\boldsymbol{X}$
$\boldsymbol{X}_{:,j}$ or $\boldsymbol{X}[:, j]$	The j-th col of matrix $\boldsymbol{X}$
$X_{i,j}$ or $X[i, j]$	The i, j-th element of matrix $\boldsymbol{X}$
$\boldsymbol{I}_N$	The $N \times N$ identity matrix
	Deep Learning
$f_{\boldsymbol{\theta}}$	A function f parametrized by $\boldsymbol{\theta}$
$\mathcal{L}$	A scalar-valued objective function
$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$	The partial derivate of a function $\mathcal{L}$ w.r.t. $\boldsymbol{\theta}$
	Graph Theory
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	A graph given by node set $\mathcal{V}$ and edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$
$\mathcal{N}(v)$	The neighborhood set $\{w \colon (w, v) \in \mathcal{E}\}$ of a node $v \in \mathcal{V}$
$\mathcal{G}[\mathcal{B}]$	The subgraph of $\mathcal{G}$ induced by the node set $\mathcal{B} \subseteq \mathcal{V}$

Contents

Acknowledgements i

Abstract iii

Notation v

1 Introduction 1
- 1.1 Motivation and Relevance 1
- 1.2 Research Questions 3
- 1.3 Main Contributions 4
- 1.4 Organization 6
- 1.5 List of Publications 7

2 Foundations of Graphs and Machine Learning 9
- 2.1 Graph Theory 9
- 2.2 Graph Machine Learning 10
- 2.3 Deep Neural Networks 13

3 Representation Learning on Graphs via Neural Message Passing 15
- 3.1 Introduction 15
- 3.2 Message Passing Graph Neural Networks 16
  - 3.2.1 Permutation Invariant and Equivariant Neural Networks 17
  - 3.2.2 A Recipe for Graph Neural Networks 19
  - 3.2.3 (Un)supervised Deep Learning on Graphs 21
  - 3.2.4 Design Principles of Graph Neural Networks 25
- 3.3 Edge-Conditioned Message Passing via B-Spline Kernels 29
  - 3.3.1 Graph Convolution via Continuous B-Spline Kernels 30
  - 3.3.2 Relation to Traditional CNNs 35
- 3.4 Maximally Expressive Graph Neural Networks 36
  - 3.4.1 Relation to the Weisfeiler-Lehman Isomorphism Test 37
  - 3.4.2 Provably Powerful Graph Neural Networks 40
  - 3.4.3 Trading Expressivity and Generalization 44
- 3.5 Evaluation 45
  - 3.5.1 Graph Classification and Regression 45
  - 3.5.2 Node Classification 48
  - 3.5.3 Superpixels Graph Classification 50
  - 3.5.4 Shape Correspondence 51

4 Task-Specific Design of Graph Neural Networks **55**
4.1 Introduction 55
4.2 Locality-Adaptive Neighborhood Aggregation 57
4.2.1 State-of-the-Art 57
4.2.2 Attentional Jumping Knowledge Neighborhood Aggregation . 59
4.2.3 Evaluation 61
4.3 Hierarchical Learning in Molecular Graphs 63
4.3.1 State-of-the-Art 65
4.3.2 Higher-Order Graph Coarsening via Tree Decompositions . . . 66
4.3.3 Inter-Message Passing with Junction Trees 67
4.3.4 Evaluation 70
4.4 Graph Matching via Differentiable Neighborhood Consensus 72
4.4.1 State-of-the-Art 73
4.4.2 Local Feature Matching 76
4.4.3 Iterative Message Passing for Neighborhood Consensus 78
4.4.4 Evaluation 83

5 Scalable Graph Neural Networks for Large-Scale Graph Learning **91**
5.1 Introduction 91
5.2 State-of-the-Art 93
5.2.1 Scalable Graph Neural Networks 93
5.2.2 Shortcomings of Existing Graph Benchmark Datasets 98
5.3 Scaling Up Graph Neural Networks via Historical Embeddings 100
5.3.1 Historical-based Sub-Tree Pruning 101
5.3.2 Analysis of Historical-caused Approximation Errors 104
5.3.3 Expressiveness of Historical-based Graph Neural Networks . . 107
5.4 The Open Graph Benchmark Datasets for Large-Scale Graph Learning 110
5.4.1 Benchmark Design Principles 111
5.4.2 Realistic Datasets for Diverse Task Categories 113
5.5 Evaluation 116
5.5.1 Open Graph Benchmark Analysis 116
5.5.2 Deep and Expressive GNNs on Large-Scale Graphs 124

6 Efficient Realization of Graph Neural Networks **129**
6.1 Introduction 129
6.2 State-of-the-Art 130
6.3 Graph Neural Networks within PyTorch Geometric 131
6.3.1 Overview of the Library 132
6.3.2 A Unified Interface for Graph Neural Networks 137
6.3.3 Efficient Sparse Tensor Arithmetic 141
6.3.4 Heterogeneous Graph Learning 148
6.3.5 Additional Features 151
6.4 PyGAS: Auto-Scaling Graph Neural Networks 152
6.5 Evaluation 154
6.5.1 A Uniform GNN Benchmark Analysis 155
6.5.2 Efficiency of GNN Design 157

7 Conclusion and Future Work **161**
7.1 Conclusion 162
7.2 Future Work 164

Bibliography **167**

Abbreviations **197**

A Appendix **199**

A.1 OGB Datasets 199

A.1.1 Node Property Prediction Datasets 199

A.1.2 Link Property Prediction Datasets 202

A.1.3 Graph Property Prediction Datasets 204

1

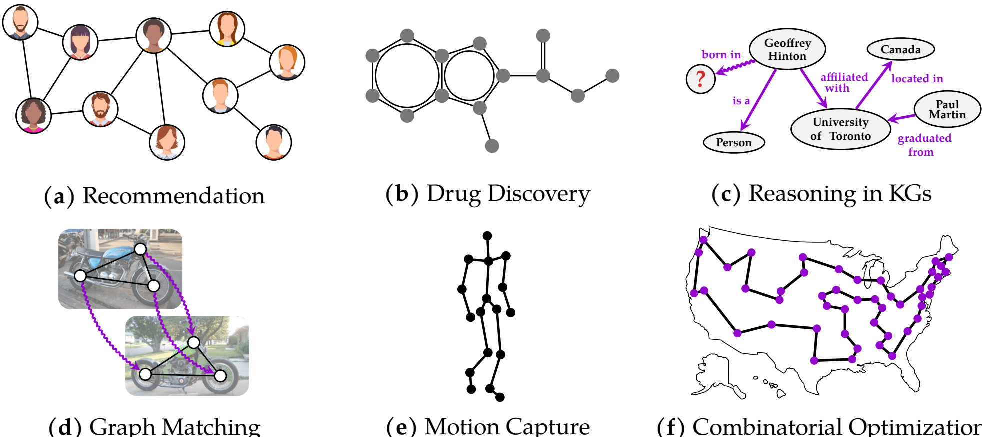
Introduction

1.1 Motivation and Relevance

Our world is highly rich in structure, composed of objects, their relations and hierarchies. Notably, this holds true for even the highest and lowest conceptual levels and scales in our world: Sentences can be described as sequences of words, maps can be decomposed into streets and their intersections, the world wide web connects websites via hyperlinks, and even chemical compounds can be described by a set of atoms and their interactions. A natural way to represent such structured information comes in the form of a *graph*, represented by a set of *nodes* and their pairwise *relationships*.

Despite the ubiquity of graphs in our world, most modern machine learning methods fail to properly handle such rich structural representations, as features are expected to be given by fixed-sized vectors, data points are treated as independent and identically distributed examples, and their sequence of computation is inherently fixed. As such, existing models are only able to process a narrow subset of all potential applications. This observation calls for the development of much more broadly applicable machine learning methods, which are eventually able to process, understand and reason about the diverse structure in our world by dynamically routing and exchanging information across entities to derive higher-level decisions (Dean, 2021).

Traditionally, machine learning models were applied on top of a set of features extracted by pre-defined procedures developed by domain experts. Such so-called *feature engineering* is considered to be notoriously difficult, and has been mostly replaced over recent years by a data-driven *representation learning* approach via *deep learning*. In contrast, deep learning is able to jointly learn the transformation of raw input data and the predictive model, thus shifting the efforts of manual feature engineering to the efforts of building better machine learning models, *e.g.*, by injecting certain forms of *inductive biases* into the model to let it learn better and more generalizable representations. Thus, a natural question to ask is how we can develop and extend *deep neural networks* to be able to learn from structured but highly irregular data, as given in the form of graphs. Due to the recent successes of deep learning in areas such as computer vision (LeCun *et al.*, 1998; Krizhevsky *et al.*, 2012; He *et al.*, 2016) or natural lan-



(**a**) Recommendation (**b**) Drug Discovery (**c**) Reasoning in KGs

(**d**) Graph Matching (**e**) Motion Capture (**f**) Combinatorial Optimization

Figure 1.1: Exemplary applications for machine learning on graphs (Morris *et al.*, 2021b), such as (**a**) recommendation and fake news detection in social networks (Easley & Kleinberg, 2010), (**b**) drug discovery via molecular graph property prediction (Gilmer *et al.*, 2017), protein folding (Jumper *et al.*, 2021), (**c**) completing and reasoning in knowledge graphs (Hu *et al.*, 2021b), (**d**) keypoint matching and network alignment (Yan *et al.*, 2016), (**e**) scene graph and motion capture understanding (Agarwal & Mangal, 2020; Chang *et al.*, 2021), and (**f**) data-driven combinatorial optimization, *e.g.*, for routing (Cappart *et al.*, 2021).

guage processing (Hochreiter & Schmidhuber, 1997; Bengio *et al.*, 2010; Vaswani *et al.*, 2017), such graph-based neural networks have the potential to significantly advance the state-of-the-art for a wide range of applications (*cf.* Figure 1.1), *e.g.*, in recommendation, drug discovery and protein folding, completing and reasoning in knowledge graphs, graph matching, scene and motion capture understanding, or combinatorial optimization. Irrespective of their broad range of applications, the injection of structural and compositional inductive biases into deep learning models ultimately manifests our understanding of a structured world, potentially leading to a much more general and powerful kind of artificial intelligence.

Despite their rich potential, the application of neural networks to graph-structured data also brings new challenges, in particular due to their underlying complex and rich topological structure, such as in the form of varying node degrees, arbitrary sizes or in the absence of fixed node orderings and reference points. As most building blocks in neural networks take fixed structure for granted (in particular due to the inherent grid-like nature of GPU processing), extending and generalizing their concepts to arbitrary structured data is difficult, both theoretically and implementation-wise.

Recently, a universal class of neural networks emerged that can seamlessly operate on graph-structured data, summarized under the umbrella term *Graph Neural Networks* (*GNNs*) (Hamilton, 2020). In its essence, GNNs capture both graph structure and feature information in a trainable fashion by following a differentiable neural *message passing* scheme (Gilmer *et al.*, 2017). In this thesis, we build upon this general framework of GNNs, analyze their capabilities, and design advanced structural and compositional inductive biases to boost their performances in general as well as in

task-specific applications. In light of the Collaborative Research Center SFB 876[1] — *Providing Information by Resource-Constrained Analysis* — this thesis, in particular, argues for the design of efficient and scalable Graph Neural Network architectures and training strategies to enable their application in resource-constrained environments.

1.2 Research Questions

This thesis is aims to explore the various aspects of deep end-to-end representation learning on graph-structured data. Due to the ubiquitousness of graphs and its wide range of applications, we focus on answering the following research questions:

Research Question 1: *How can we develop a broadly applicable class of deep neural models for learning on graph-structured data that inherits from successful principles of traditional neural network building blocks?*

Most of the breakthroughs in deep learning are due to the properties of Convolutional Neural Networks (CNNs) (LeCun *et al.*, 1998), *i.e.* local connectivity, weight sharing and shift invariance. Since those layers are defined on inputs with a grid-like structure, they are not trivially portable to non-Euclidean domains like graphs or discrete manifolds. Notably, message passing GNNs utilize highly similar concepts of the convolution operator, but can be applied to much more irregularly structured domains. However, in comparison to CNNs, commonly utilized GNN operators are also more restrictive in learning meaningful geometric patterns (Huszár, 2016). This observation gives natural rise to the question on how we can model a general class of GNNs that can incorporate anisotropy in the form of directional descriptors, while effectively resembling the traditional definition of CNNs for discrete input data.

Research Question 2: *What are the limitations and inherent weaknesses of Graph Neural Networks and how can we overcome them both theoretically and in practice?*

While GNNs are a popular tool for feature aggregation across all kinds of structured data, it is unclear how well they are actually doing in reasoning about and encoding structural properties of the underlying graph. Ideally, a maximally powerful GNN is able to map isomorphic graphs to the same representation in the embedding space, while it maps non-isomorphic ones to different representations (Xu *et al.*, 2019c). Such an ability, however, requires a GNN to solve the challenging graph isomorphism problem (Biggs *et al.*, 1986), which directly implies that there exists an upper bound for a GNN's representational power.

Furthermore, we are interested in how to overcome inherent weaknesses of GNNs by designing task-specific methods and injecting domain knowledge, *i.e.* mitigating oversmoothing effects in deep GNNs (Li *et al.*, 2018a), reasoning about higher-order graph structures such as cycles (Klicpera *et al.*, 2020b), or distributing global positional node information (You *et al.*, 2019).

Research Question 3: *How can we scale Graph Neural Networks to giant input domains, in particular, without compromising their theoretical properties?*

A major challenge of GNNs is the difficulty to scale them to large graphs. In particular, there exists a high inter-dependency between nodes that grows exponentially

[1]SFB 876: `https://sfb876.tu-dortmund.de` (last access: August 25, 2022)

with respect to the number of layers, since the embedding of a given node depends recursively on all its neighbor's embeddings (Frasca *et al.*, 2020). As a result, scalability techniques are indispensable for applying GNNs to large-scale graphs. However, existing approaches based on graph sub-sampling or non-trainable propagations weaken the expressive power of message passing in reasoning about structural graph properties at scale (Wu *et al.*, 2019a). Thus, a natural question to ask is how we can enable GNNs to learn structural graph properties while still being able to apply them to large-scale graphs.

Research Question 4: *How can we practically implement Graph Neural Networks in a simultaneously flexible and efficient way, and enable scalable, robust and reproducible graph machine learning research?*

With the rise of GNNs as a state-of-the-art technique for Graph Representation Learning, there exists an urgent demand in both flexible and powerful libraries for accelerating research and putting existing models into production. Notably, meeting both requirements is challenging, as high GPU throughput needs to be achieved on highly sparse and irregular data of varying size across a wide range of different model implementations. Furthermore, modern deep learning software libraries are heavily designed with regular structures and dense tensor computation in mind (Paszke *et al.*, 2019), which makes the efficient realization of GNNs even more challenging.

Lastly, most of the frequently-used datasets to evaluate the performance of graph machine learning models are extremely small compared to graphs found in real applications, leading to unstable and nearly statistically identical performance across different models (Shchur *et al.*, 2018). Since historically, high-quality and large-scale datasets have played significant roles in advancing machine learning research, we ask what is needed to facilitate scalable, robust, and reproducible Graph Representation Learning research.

1.3 Main Contributions

This thesis is based on several main contributions with the high-level goal to address and answer the aforementioned research questions. An overview of all the different main contributions is illustrated in Figure 1.2.

We introduce the unified framework of message passing GNNs and demonstrate its generality for a wide range of operators and learning tasks. In particular, we show that GNNs utilize and generalize key concepts such as locality or weight sharing that are well-known from successful building blocks of traditional neural networks (**Research Question 1**). Furthermore, we introduce the *SplineCNN* model in which message passing is conditioned based on a continuous B-spline kernel formulation (Fey *et al.*, 2018). SplineCNNs improve upon earlier work by endowing GNNs with an anisotropic aggregation scheme, which is able to resemble the traditional definition of CNNs for discrete input data (**Research Question 1**). Furthermore, SplineCNNs can be utilized in a variety of diverse domains, such as for learning on graphs, superpixels, point clouds and manifolds.

We analyze the representational power of GNNs from a theoretical point of view by relating them to the Weisfeiler-Lehman (WL) graph isomorphism heuristic (Morris *et al.*, 2019, 2021a). We show that GNNs have at most the same expressiveness as the

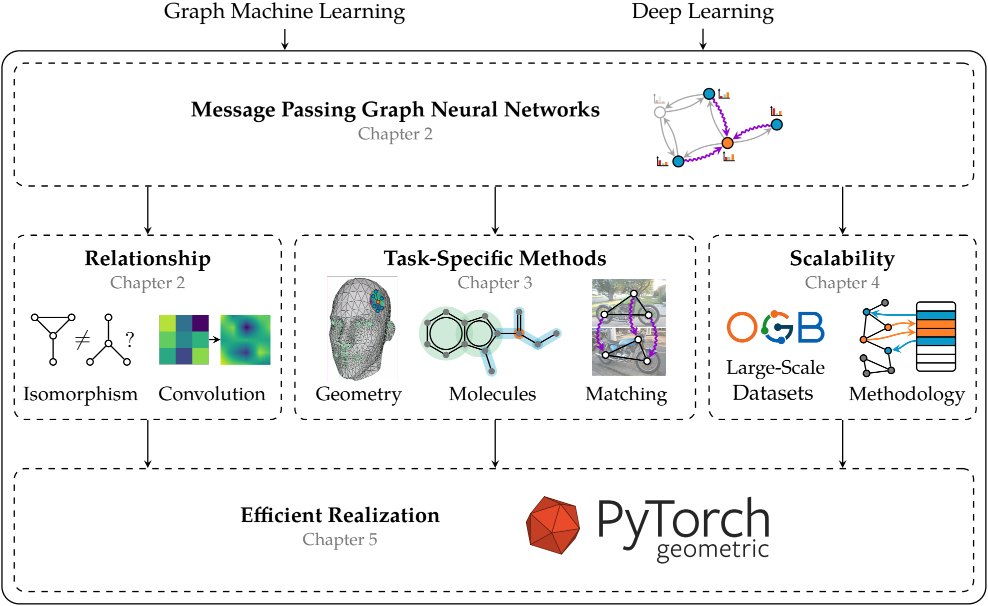


Figure 1.2: Illustrative overview of the main contributions of this thesis, sitting at the intersection of machine learning with graphs and deep learning.

WL test in terms of distinguishing non-isomorphic (sub-)graphs, and therefore obey the same shortcomings (**Research Question 2**). Based on these findings, we propose a generalization of GNNs, the so-called *k-dimensional GNNs* (*k-GNNs*), which can take higher-order graph structures at multiple scales into account, leading to provably more powerful Graph Neural Networks.

We identify additional inherent shortcomings of GNNs and show how to overcome them in practice (**Research Question 2**). In order to mitigate the problem of over-smoothing in deep GNNs, we propose the *Dynamic Neighborhood Aggregation* (*DNA*) procedure that allows for a selective and node-adaptive aggregation of neighbors of potentially differing locality, guided by attention (Fey, 2019). To allow for more expressive Graph Neural Networks in molecular learning, we propose the *Hierarchical Inter-Message Passing* (*HIMP*) model that is able to exchange information between different higher-order substructures in molecules, *e.g.*, between rings or bonds (Fey *et al.*, 2020b). Furthermore, we present a two-stage neural architecture for the task of graph matching, in which we learn to refine structural correspondences between nodes in different graphs. Our *Deep Graph Matching Consensus* (*DGMC*) model aims to reach a matching consensus in local neighborhoods by sparsely distributing global positional encodings (Fey *et al.*, 2020a).

We propose a general framework named *GNNAutoScale* (*GAS*) that is able to scale arbitrary message passing GNNs to giant graphs (Fey *et al.*, 2021). GAS prunes entire sub-trees of the computation graph by utilizing historical embeddings acquired in prior training iterations, leading to constant GPU memory consumption w.r.t. input node size. While prior scalability solutions weaken the expressive power of message passing due to sub-sampling of edges or non-trainable propagations, GAS is provably able to maintain the expressive power of the original GNN (**Research Question 3**).

We develop the *PyTorch Geometric* (*PyG*) library (Fey & Lenssen, 2019), a library for accelerating deep learning research on irregularly structured input data such as graphs, point clouds and manifolds, built upon PyTorch (Paszke *et al.*, 2019). PyG achieves high data throughput by leveraging sparse GPU acceleration and dedicated CUDA kernels, while being both easy and flexible to-use via a general message passing interface (**Research Question 4**). All our findings as well as related research are condensed into the PyG library in order to make state-of-the-art graph-based deep learning architectures broadly applicable. Furthermore, our PyG extension, *PyGAS*, allows to convert common and custom GNN models from PyG into a scalable, fast and memory-efficient variant (Fey *et al.*, 2021).

We introduce the *Open Graph Benchmark* (*OGB*) suite that bundles diverse, challenging and realistic graph benchmark datasets (Hu *et al.*, 2020a, 2021b). In contrast to prior frequently-used graph datasets, OGB datasets are orders of magnitude larger than existing ones, encompass multiple important graph machine learning tasks, and cover a diverse range of domains. We show that our datasets present significant challenges of scalability and out-of-distribution generalization under realistic data splits, indicating fruitful opportunities for future research (**Research Question 4**).

1.4 Organization

We start by introducing several background topics in Chapter 2, which will serve as a basis for the material presented in the rest of this thesis.

Afterwards, this thesis is structured into four main chapters, which are additionally highlighted in Figure 1.2: **Chapter 3** — *Representation Learning on Graphs via Neural Message Passing* — will introduce the overarching topic of this thesis. Here, we will study the central concepts of applying deep learning principles to graph-structured and highly irregular input data, such as equivariance and locality, leading to the common definition of Graph Neural Networks via a neural message passing formulation (Section 3.2). Further, we relate the concepts of GNNs to well-known deep learning techniques and graph isomorphism heuristics, namely CNNs (Section 3.3) and the WL heuristic (Section 3.4), respectively. Based on these findings, **Chapter 4** — *Task-Specific Design of Graph Neural Networks* — will explore the design of graph-based neural architectures that are suitable for learning on specific tasks or domains, leading to provably more powerful architectures that are able to overcome the inherent weaknesses of GNNs, such as over-smoothing (Section 4.2), expressivity (Section 4.3) and locality (Section 4.4). Furthermore, **Chapter 5** — *Scalable Graph Neural Networks for Large-Scale Graph Learning* — will study the issues of training GNNs on larger scale, and will explore advanced techniques to overcome this problem (Section 5.3). In order to facilitate GNN benchmarking and evaluation on larger scale and to advance research, we will further introduce a set of diverse, realistic and large-scale benchmark datasets (Section 5.4). Finally, **Chapter 6** — *Efficient Realization of Graph Neural Networks* — unifies the aforementioned topics and studies how Graph Neural Networks can be implemented flexibly and trained efficiently, leading to a general framework for realizing deep learning on graph-structured data (Section 6.3) and its extension for learning on graphs of larger scale (Section 6.4).

After presenting the main body of our work, we will conclude and outline interesting directions for future work in Chapter 7.

1.5 List of Publications

This thesis is based on research initially published by the author in a set of peer-reviewed publications. In the following, we list each publication and its original appearance, and outline the authors' contribution to the respective works. For this, we briefly divide the set of contributions into concept/idea finding, realization, evaluation and writing:

- **Matthias Fey***, Jan Eric Lenssen*, Frank Weichert and Heinrich Müller (2018). "SplineCNN: Fast Geometric Deep Learning with Continuous B-Spline Kernels." In: *Computer Vision and Pattern Recognition* (*CVPR*), *cf.* Chapter 3 (*equal contribution).
Contribution: The author contributed significantly to concept, realization and evaluation, and assisted in writing.

- **Matthias Fey** and Jan Eric Lenssen (2019). "Fast Graph Representation Learning with PyTorch Geometric." In: *ICLR Workshop on Representation Learning on Graphs and Manifolds* (*RLGM*) **(contributed talk)**, *cf.* Chapter 6.
Contribution: The author contributed the majority of work in respect to concept, realization, evaluation and writing.

- **Matthias Fey**, Jan Eric Lenssen, Christopher Morris, Jonathan Masci and Nils M. Kriege (2020). "Deep Graph Matching Consensus." In: *International Conference on Learning Representations* (*ICLR*), *cf.* Chapter 4.
Contribution: The author contributed the majority of work in respect to realization and evaluation, and contributed significantly to concept and writing.

- **Matthias Fey**, Jan Eric Lenssen, Frank Weichert and Jure Leskovec (2021). "GNNAutoScale: Scalable And Expressive Graph Neural Networks via Historical Embeddings." In: *International Conference on Machine Learning* (*ICML*), *cf.* Chapters 5 and 6.
Contribution: The author contributed the majority of work in respect to concept, realization, evaluation and writing.

- **Matthias Fey** (2019). "Just Jump: Dynamic Neighborhood Aggregation in Graph Neural Networks." In: *ICLR Workshop on Representation Learning on Graphs and Manifolds* (*RLGM*), *cf.* Chapter 4.
Contribution: The author contributed solely to concept, realization, evaluation and writing.

- **Matthias Fey***, Jan-Gin Yuen* and Frank Weichert (2020). "Hierarchical Inter-Message Passing for Learning on Molecular Graphs." In: *ICML Workshop on Graph Representation Learning and Beyond* (*GRL+*), *cf.* Chapter 4 (*equal contribution).
Contribution: The author contributed the majority of work in respect to concept, realization and writing, and contributed significantly to evaluation.

- Weihua Hu, **Matthias Fey**, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta and Jure Leskovec (2020). "Open Graph Benchmark: Datasets for Machine Learning on Graphs." In: *Advances in Neural Information Processing Systems* (*NeurIPS*) **(spotlight)**, *cf.* Chapter 5.

Contribution: The author contributed significantly to realization and evaluation, and assisted in writing.

- Weihua Hu, **Matthias Fey**, Hongyu Ren, Maho Nakata, Yuxiao Dong and Jure Leskovec (2021). "OGB-LSC: Large-Scale Challenge for Machine Learning on Graphs." In: *NeurIPS: Datasets and Benchmarks Track*, *cf.* Chapter 5.
  Contribution: The author contributed significantly to realization and evaluation, and assisted in writing.

- Christopher Morris, Martin Ritzert, **Matthias Fey**, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan and Martin Grohe (2019). "Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks." In: *Conference on Artificial Intelligence* (*AAAI*), *cf.* Chapter 3.
  Contribution: The author contributed the majority of work in respect to realization and evaluation, contributed to concept, and assisted in writing.

- Christopher Morris, **Matthias Fey**, Nils M. Kriege (2021). "The Power of the Weisfeiler-Leman Algorithm for Machine Learning with Graphs." In: *International Joint Conference on Artificial Intelligence* (*IJCAI*) *— Survey Track*, *cf.* Chapter 3.
  Contribution: The author contributed to writing.

In addition, the author has contributed to the following publications. These describe special applications of GNNs out of scope of this thesis or non peer-reviewed work.

- Jan Eric Lenssen, **Matthias Fey** and Pascal Libuschewski (2018). "Group Equivariant Capsule Networks." In: *Advances in Neural Information Processing Systems* (*NeurIPS*).
  Contribution: The author contributed significantly to realization and evaluation, contributed to concept, and assisted in writing.

- Marian Kleineberg, **Matthias Fey** and Frank Weichert (2020). "Adversarial Generation of Continuous Implicit Shape Representations." In: *Eurographics & Eurovis*.
  Contribution: The author contributed the majority of work in respect to writing, contributed significantly to concept and evaluation, and contributed to realization.

- Nils M. Kriege, **Matthias Fey**, Denis Fisseler, Petra Mutzel and Frank Weichert (2018). "Recognizing Cuneiform Signs Using Graph-Based Methods." In: *SDM Workshop on Cost-Sensitive Learning* (*COST*).
  Contribution: The author contributed significantly to realization and evaluation, and contributed to concept and writing.

- Christopher Morris, Yaron Lipman, Haggai Maron, Bastian Rieck, Nils M. Kriege, Martin Grohe, **Matthias Fey** and Karsten Borgwardt (2021). "Weisfeiler and Leman go Machine Learning: The Story so far." In: *CoRR, abs/2112.09992*.
  Contribution: The author assisted in writing.

2

Foundations of Graphs and Machine Learning

This chapter establishes notions and definitions, and provides a brief introduction to several background topics used throughout this thesis. In particular, we formalize graph theory notation in Section 2.1, review graph machine learning techniques in Section 2.2, and introduce the concepts of deep neural networks in Section 2.3. A more detailed overview of the state-of-the-art on the various research topics of this thesis can be found in the individual main chapters.

2.1 Graph Theory 9

2.2 Graph Machine Learning 10

2.3 Deep Neural Networks 13

2.1 Graph Theory

A *graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is defined by a finite *node* set $\mathcal{V} = \{1, \dots N\}$, $|\mathcal{V}| = N < \infty$, and a set of *edges* $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, defining the relations among its nodes. We say that $\mathcal{G}$ is *undirected* in case it holds that $(w, v) \in \mathcal{E}$ for every $(v, w) \in \mathcal{E}$. Otherwise, we call $\mathcal{G}$ a *directed* graph. A *self-loop* denotes an edge connecting a node with itself, *i.e.* $(v, v) \in \mathcal{E}$. For a node $v \in \mathcal{V}$, its *neighborhood* set is denoted by its *in-going* edges $\mathcal{N}(v) = \{w : (w, v) \in \mathcal{E}\}$. The *node degree* of $v \in \mathcal{V}$ is given by $|\mathcal{N}(v)|$. For a subset of nodes $\mathcal{B} \subset \mathcal{V}$, $\mathcal{G}[\mathcal{B}]$ refers to its *induced subgraph*, in which only nodes in $\mathcal{B}$ and edges among nodes in $\mathcal{B}$ are maintained, *i.e.* $\mathcal{E}[\mathcal{B}] = \mathcal{E} \cap (\mathcal{B} \times \mathcal{B})$ (Biggs *et al.*, 1986).

A graph $\mathcal{G}$ induces an *adjacency matrix* $\boldsymbol{A} \in \{0, 1\}^{|\mathcal{V}| \times |\mathcal{V}|}$ such that $A_{v,w} = 1$ in case $(v, w) \in \mathcal{E}$, and $A_{v,w} = 0$ otherwise. Importantly, the adjacency matrix $\boldsymbol{A}$ is treated *sparsely* such that only non-zero elements are maintained. As most real-world graphs are sparse by nature (*i.e.* nodes only connect to a small fraction of other nodes), this reduces both storage and computation overheads by a wide margin.

Two graphs $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ and $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$ are *isomorphic* or *topologically identical* in case they only differ by permutation, *i.e.* there exists an edge preserving bijection $\pi : \mathcal{V}_1 \to \mathcal{V}_2$ such that $(v, w) \in \mathcal{E}_1$ if and only if $(\pi(v), \pi(w)) \in \mathcal{E}_2$. Checking whether two graphs are isomorphic is a challenging problem, and no polynomial-time algorithm is known for it yet (Garey, 1979; Garey & Johnson, 2002; Babai, 2016). Apart from some corner cases (Arvind *et al.*, 2015), the *Weisfeiler-Lehman* (*WL*) algorithm (Weisfeiler & Lehman, 1968) is an effective and computationally efficient heuristic to solve the graph isomorphism test for a broad class of graphs, *cf.* Section 3.4.1.

We further allow the attachment of node-level and edge-level *feature vectors* to a given graph $\mathcal{G}$, *i.e.* $\boldsymbol{x} : \mathcal{V} \to \mathbb{R}^F$ and $\boldsymbol{e} : \mathcal{E} \to \mathbb{R}^D$ represent important characteristics of nodes and edges, respectively. Feature vectors of all the nodes and edges can be compactly stored in feature matrices $\boldsymbol{X} \in \mathbb{R}^{|\mathcal{V}| \times F}$ and $\boldsymbol{E} \in \mathbb{R}^{|\mathcal{E}| \times D}$, respectively. We refer to $\boldsymbol{X}_{:,i}$ as the *feature map* or *feature channel* at position $1 \leq i \leq F$. For edge-level features, we abuse notation and allow indexing via $\boldsymbol{e}_{v,w}$ if $(v, w) \in \mathcal{E}$.

In order to comprehensively characterize real-world applications, graphs may further incorporate multiple node and edge types, leading to heterogeneous graph information. A *heterogeneous graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T}_V, \mathcal{T}_E)$ holds additional *node type* and *edge type* mapping functions $\mathcal{T}_V : \mathcal{V} \to \Sigma_V$ and $\mathcal{T}_E : \mathcal{E} \to \Sigma_E$, respectively. The terms edge type and *relation* are used interchangeably throughout this thesis. Note that heterogeneous graphs resemble the classical definition of graphs in case $|\Sigma_V| = 1$ and $|\Sigma_E| = 1$. Importantly, feature distributions may vary across different node and edge types, leading to feature vectors of potentially varying dimensionalities, *i.e.* node-level feature $\boldsymbol{x}_v \in \mathbb{R}^{F(\mathcal{T}_V(v))}$ and edge-level feature $\boldsymbol{e}_{v,w} \in \mathbb{R}^{D(\mathcal{T}_E((v,w)))}$ dimensionalities depend on the given type. We refer to the *typed neighborhood* set of $v \in \mathcal{V}$ and edge type $r \in \Sigma_E$ as $\mathcal{N}_r(v) = \{w : (w, v) \in \mathcal{E} \wedge \mathcal{T}_E((w, v)) = r\}$ (Zhu *et al.*, 2019b).

2.2 Graph Machine Learning

Machine learning on complex networks has become an integral part of research in both natural and social sciences. In order to learn on such graph-structured data, machine learning algorithms need to, in particular, exploit its rich structural properties.

In this thesis, we are mostly interested in the task of *supervised* graph machine learning, in which either

- each node $v \in \mathcal{V}$ is associated with a label y_v, and the goal is to obtain a representation $\boldsymbol{h}_v$ from which y_v can be easily predicted (*node-level prediction*),
- each pair $(v, w) \in \mathcal{V} \times \mathcal{V}$ is associated with the existence of a link (*i.e.* $(v, w) \in \mathcal{E}$ or $(v, w) \notin \mathcal{E}$), and the goal is to obtain representations $\boldsymbol{h}_v$ and $\boldsymbol{h}_w$ from which the existence of a link can be easily predicted (*edge-level prediction*), or
- each graph $\mathcal{G} \in \{\mathcal{G}_1, \ldots, \mathcal{G}_N\}$ is associated with a label y, and the goal is to obtain a representation $\boldsymbol{h}_\mathcal{G}$ from which y can be easily predicted (*graph-level prediction*).

Importantly, the vectorial representations $\boldsymbol{h}_v$ or $\boldsymbol{h}_\mathcal{G}$ should be able to preserve structural graph information, either in a local or global fashion, respectively. Representations can also be obtained in an unsupervised learning scenario, and eventually used

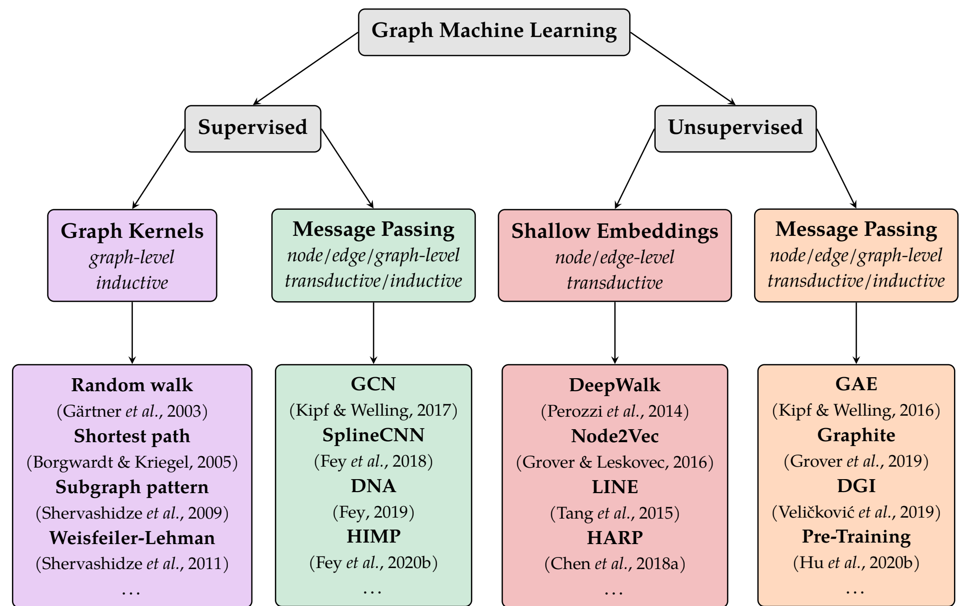


Figure 2.1: Taxonomy of graph machine learning models. We broadly categorize graph machine learning approaches into four categories: *graph kernels* (supervised learning), *shallow node embedding techniques* (unsupervised learning), and *message passing Graph Neural Networks* (supervised learning and unsupervised learning). Notably, message passing GNNs, which builds the focus of this thesis, excel in their flexibility: they can handle node/edge/graph-level tasks, are applicable to either transductive and inductive learning scenarios, and can be trained both in a supervised and unsupervised fashion.

for a supervised down-stream task (Veličković *et al.*, 2019; Hu *et al.*, 2020b). Furthermore, graph machine learning tasks can be further divided into *transductive* and *inductive* learning scenarios. In the transductive learning task, we are allowed to obverse the full data, but only use a small subset of ground-truth labels for supervision. This is the default learning scenario for obtaining node-level predictions on single graphs. In the inductive case, the model is restricted to observe data in a given training set, *e.g.*, a set of graphs, and is then applied to unseen examples afterwards.

This thesis focus on graph machine learning based on deep learning via message passing Graph Neural Networks. The most dominant approaches for graph machine learning prior to the deep learning era are known as *node embedding techniques* (Perozzi *et al.*, 2014; Grover & Leskovec, 2016; Tang *et al.*, 2015; Dong *et al.*, 2017a; Abu-El-Haija *et al.*, 2018; Chami *et al.*, 2020) and *graph kernels* (Kriege *et al.*, 2020). Figure 2.1 proposes a taxonomy of different graph machine learning models and compares their respective properties, *e.g.*, their application to supervised/unsupervised tasks, node/edge/graph-level tasks and transductive/inductive tasks. Notably, we see that alternative graph machine learning approaches are limited to specific tasks and properties, while message passing Graph Neural Networks (GNNs) do not have such restrictions.

Specifically, node embedding techniques rely on embedding nodes into low-dimensional vectorial representations $\boldsymbol{h}_v$ via a *shallow embedding lookup table* such that the likelihood of preserving neighborhoods is maximized, *i.e.* nearby nodes should receive similar embeddings while distant nodes should receive distinct embeddings (Perozzi *et al.*, 2014). These techniques generalize the famous SkipGram model for obtaining low-dimensional word embeddings (Mikolov *et al.*, 2013), in which sequences of words are now interpreted as sequences of nodes, *e.g.*, given via randomly-generated walks. Specifically, given a *random walk* $\mathcal{W} = (v_{\pi(1)}, \ldots, v_{\pi(k)})$ of length k starting at node $v \in \mathcal{V}$, the objective is to maximize the likelihood of observing node $v_{\pi(i)}$ given node v. This objective can be efficiently trained via stochastic gradient descent in a contrastive learning scenario

$$\mathcal{L} = \sum_{w \in \mathcal{W}} - \log \left(\sigma\left(\boldsymbol{h}_v^\top \boldsymbol{h}_w \right) \right) + \sum_{w \sim \mathcal{V} \setminus \mathcal{W}} - \log \left(1 - \sigma\left(\boldsymbol{h}_v^\top \boldsymbol{h}_w \right) \right), \tag{2.1}$$

in which "non-existent walks" (so called *negative examples*) are sampled and trained jointly, and σ denotes the $\mathrm{sigmoid}$ function. Alternative formulations mostly differ in the way random walks are drawn, *e.g.*, Grover & Leskovec (2016) allow flexible control via biased random walk procedures based on breadth-first or depth-first samplers. Importantly, shallow node embeddings are trained in an *unsupervised* fashion, and can eventually be used as input for a given down-stream task, *e.g.*, in node-level tasks $\boldsymbol{h}_v$ can directly be used as input to a final classifier. For edge-level tasks, edge-level representations can be obtained via averaging $\frac{1}{2}(\boldsymbol{h}_v + \boldsymbol{h}_w)$ or the Hadamard product $(\boldsymbol{h}_v \odot \boldsymbol{h}_w)$. Despite the simplicity of node embedding techniques, they are also subject to certain shortcomings. In particular, they fail to incorporate rich feature information attached to nodes and edges, and cannot be trivially applied to unseen graphs as learnable parameters are fixed to the nodes of a particular graph.

Alternative techniques for learning on graph-structured data are known as graph kernels, which finds their application in graph-level prediction tasks. Graph kernels define functions that measure the similarity between graphs, plugged into a *Support-Vector Machine* (*SVM*) to obtain graph-level predictions (Kriege *et al.*, 2020). A *kernel* maps each pair of examples to a real number that corresponds to an inner product between two vectors in a (usually high-dimensional) Hilbert space, such that they are linearly separable (Schölkopf & Smola, 2002; Shawe-Taylor & Cristianini, 2004). Over the last few years, numerous graph kernels have been proposed, which can be broadly categorized into neighborhood aggregation approaches, assignment- and matching-based approaches, subgraph pattern approaches, and walk- or path-based approaches (Kriege *et al.*, 2020). Ultimately, graph kernels rely on hand-crafted features in order to obtain meaningful similarities between graphs, and are thus subject to manual feature engineering. For example, random walk kernels count the number of walks that two graphs have in common (Gärtner *et al.*, 2003; Kashima *et al.*, 2003). Shortest path kernels compare the length of shortest paths between all pairs of nodes in two graphs (Borgwardt & Kriegel, 2005; Hermansson *et al.*, 2015). Subgraph pattern kernels ignore the global graph structure altogether and represent graphs as bags of subgraph patterns or *graphlets* (Shervashidze *et al.*, 2009). Neighborhood aggregation kernels obtain node-level labels based on the local structure around them, *e.g.*, obtained via the WL algorithm, and define similarities via the graph-level histogram of node labels (Shervashidze *et al.*, 2011; Morris *et al.*, 2017). Finally, assignment- and matching-based kernels define a measure of similarity based on the optimal assignment between the nodes in two graphs (Fröhlich *et al.*, 2005), *e.g.*, obtained from the

WL color refinement procedure (Kriege *et al.*, 2016). As the aforementioned graph kernels are of discrete nature, recent efforts in graph kernel research aim to integrate continuous feature information as well (Kriege & Mutzel, 2012; Feragen *et al.*, 2013; Orsini *et al.*, 2015).

Recently, *Graph Neural Networks* (*GNNs*) were proposed as an alternative approach for machine learning on graphs (Gori *et al.*, 2005b; Scarselli *et al.*, 2009; Li *et al.*, 2016b; Kipf & Welling, 2017; Gilmer *et al.*, 2017; Battaglia *et al.*, 2018), *cf.* Chapter 3. In contrast to the aforementioned methods, GNNs are able to jointly capture local graph structure and feature information in a trainable and fully end-to-end fashion. They do so by following a simple yet powerful differentiable neighborhood aggregation scheme, motivated from two major perspectives: The generalization of classical Convolutional Neural Networks to irregular domains (Section 3.3), and their strong relations to the WL algorithm (Section 3.4).

2.3 Deep Neural Networks

The availability of massively parallel co-processors opened the door for training large models and dramatically changed the way we approach machine learning in general. The increase in computational capabilities was most noticable in the field of *deep learning*. In particular, deep neural networks utilize a data-driven approach that jointly learn representations *and* predictions of the data. This has led to major improvements to the state-of-the-art in the fields of computer vision (LeCun *et al.*, 1998; Krizhevsky *et al.*, 2012; He *et al.*, 2016) and natural language processing (Vaswani *et al.*, 2017).

In its simplest form, *deep neural networks* can be described as the composition of parametrized linear functions $f_{\boldsymbol{\theta}}^{(\ell)}$ and element-wise non-linear transformations σ:

$$f_{\boldsymbol{\theta}}^{(L)} \circ \sigma \circ f_{\boldsymbol{\theta}}^{(L-1)} \circ \sigma \circ \cdots \circ \sigma \circ f_{\boldsymbol{\theta}}^{(2)} \circ \sigma \circ f_{\boldsymbol{\theta}}^{(1)}, \tag{2.2}$$

where $\circ$ denotes function composition (Goodfellow *et al.*, 2016). Deep neural networks act as non-linear function approximators that are able to adapt to a given learning task by adjusting its parameters $\boldsymbol{\theta}$ and by iteratively computing and *learning* intermediate *hidden* representations $\boldsymbol{h}^{(\ell)}$ of the raw input data. The most simple neural network is the *Multi-Layer Perceptron* (*MLP*) (Rosenblatt, 1958), which iteratively applies the parametrized affine linear transformation

$$f_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{h}^{(\ell-1)}) = \boldsymbol{W}^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)} \tag{2.3}$$

for L *layers*, where $\boldsymbol{W}^{(\ell)}$ and $\boldsymbol{b}^{(\ell)}$ denote the learnable *weight* matrix and *bias* vector of the ℓ-th *layer*, respectively. We refer to either $\boldsymbol{x}$ or $\boldsymbol{h}^{(0)}$ as the neural network's input, and to $\boldsymbol{h}^{(\ell)}$ as its hidden feature representation obtained from $f_{\boldsymbol{\theta}}^{(\ell)}$. The *Rectified Linear Unit* (*ReLU*) function $\sigma(\cdot) = \max(0, \cdot)$ is the non-linearity function of choice in most modern MLPs. In order to train neural networks, we obtain the gradients $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(\ell)}}$ and $\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(\ell)}}$ of the differentiable function $f_{\boldsymbol{\theta}}^{(\ell)}$ via *backpropagation* (Werbos, 1982), and minimize the *objective function* $\mathcal{L}$ via variants of stochastic gradient descent optimization, *e.g.*, Adam (Kingma & Ba, 2015). Modern deep learning software frameworks (Abadi *et al.*, 2015; Paszke *et al.*, 2019) utilize *automatic differentiation* to automatically backpropagate gradients without any user intervention.

Although MLPs are known to be universal approximators (Hornik, 1991; Hornik *et al.*, 1989), they usually fail to generalize well to unseen high-dimensional input data. As such, numerous neural network variants have been proposed that inject useful inductive biases into their architectures. These variants mostly differ in their elementary building blocks $f_{\boldsymbol{\theta}}^{(\ell)}$ and how these building blocks are combined as part of a *computation graph*, such as in Recurrent Neural Networks (Hochreiter & Schmidhuber, 1997), Convolutional Neural Networks (LeCun *et al.*, 1998) and Transformer Networks (Vaswani *et al.*, 2017), *cf.* Goodfellow *et al.* (2016). Importantly, modern neural networks are becoming much more complex and eventually assemble networks of a vast number of functional building blocks that potentially change dynamically dependent on the input fed to them. As a result, deep neural networks are nowadays more viewed just as regular programs, except that they are parameterized, automatically differentiated, and optimizable; a concept framed as *differentiable programming*. This is also reflected in the design of message passing Graph Neural Networks (Chapter 3) and their similarities to the dynamic programming paradigm (Xu *et al.*, 2020).

3

Representation Learning on Graphs via Neural Message Passing

Graph Neural Networks capture local graph structure and feature information by following a differentiable neural message passing scheme. We demonstrate the generality of message passing through a unified framework suitable for a wide range of operators and learning tasks. In addition, we propose SplineCNNs which fit into this message passing scheme by conditioning messages via a continuous B-spline kernel formulation, while resembling the traditional definition of CNNs for discrete input. We further relate the representational power of Graph Neural Networks to their ability to distinguish non-isomorphic (sub-)graphs, and propose effective solutions to allow them to reach maximal expressiveness.

3.1 Introduction 15

3.2 Message Passing Graph Neural Networks 16

3.3 Edge-Conditioned Message Passing via B-Spline Kernels 29

3.4 Maximally Expressive Graph Neural Networks 36

3.5 Evaluation 45

3.1 Introduction

Traditional graph machine learning (Ma & Tang, 2020; Hamilton, 2020) relies on feature engineering techniques in the form of hand-designed graph statistics in order to obtain a vectorial graph representation suitable for a given down-stream task, *cf.* Section 2.2. However, the process of manual feature engineering is considered to be notoriously difficult, as it requires an immense amount of human efforts, and leads to solving the problem based on trial and error. After all, manual feature engineering might even be suboptimal as we often lack the prior knowledge about the essential features for a given task. Furthermore, graphs may come in many different forms,

sometimes even allowing for multiple types of nodes or edges, which makes identifying and integrating the essential parts of a graph via engineering efforts rather challenging (Battaglia *et al.*, 2018).

As a result, over the past few years, there has been made a tremendous amount of efforts in ultimately *learning* such a representation of a graph, typically guided by a application-specific down-stream task (Defferrard *et al.*, 2016; Kipf & Welling, 2016; Hamilton *et al.*, 2017; Veličković *et al.*, 2018). In particular, deep learning has been proven to be a powerful tool for representation learning that led to significant advancements in domains such as computer vision, speech recognition and natural language processing (LeCun *et al.*, 1998; Hochreiter & Schmidhuber, 1997; He *et al.*, 2016; Vaswani *et al.*, 2017). However, the primary challenge in developing complex neural building blocks for graph-structured data is that our conventional deep learning architectures do not apply, as traditional deep learning approaches have been designed with regular structured data in mind, such as images and sequences. In contrast, graphs are highly irregular structures, with nodes in a graph being unordered and having distinct and varying sized neighborhoods. Furthermore, nodes in a graph are inherently connected, leading to inter-dependency of data, whereas traditional machine learning techniques often assume that data is independent and identically distributed.

In this chapter, we will systematically introduce the *Graph Neural Network* (*GNN*) formalism (Gilmer *et al.*, 2017) — a general framework for defining deep neural networks on graphs based upon a *neural message passing* scheme (Section 3.2). The key idea of GNNs is that they learn representations of nodes in a graph that actually depend on the structure of the graph as well as any feature information attached to it. We start to derive the GNN formalism from the principles of designing invariant and equivariant function approximators. Notably, they can also be seen as a more general class of Convolutional Neural Networks (CNNs) (LeCun *et al.*, 1998), as GNNs are subject to well-known principles of traditional deep learning techniques, *e.g.*, locality and weight sharing. In particular, this observation leads to the development of our specialized anisotropic Graph Neural Network instantiation called *Spline-Based Convolutional Neural Network* (*SplineCNN*). SplineCNNs are introduced in Section 3.3 and Fey *et al.* (2018). Notably, they resemble the traditional definition of CNNs for discrete input data, while directly being applicable on more diverse and general domains as well, *e.g.*, for learning on either simple or embedded graphs, reaching state-of-the-art performance on all of these tasks. Furthermore, we motivate Graph Neural Networks from a graph theoretical point of view by relating them to well-known graph isomorphism approximators. Specifically, in Section 3.4, we study the expressive power of Graph Neural Networks by relating them to the Weisfeiler-Lehman (WL) graph isomorphism heuristic (Weisfeiler & Lehman, 1968), and propose effective solutions to allow them to reach maximal expressiveness.

3.2 Message Passing Graph Neural Networks

We start by introducing the unified framework of message passing GNNs and demonstrate its generality for a wide range of operators and learning tasks. In particular, we show that GNNs utilize and generalize key concepts such as locality or weight sharing that are well-known from successful building blocks of traditional neural networks.

$$\begin{array}{ccc} x & \xrightarrow{f} & y \\ \big\downarrow T_g^{\mathcal{X}} & & \big\downarrow T_g^{\mathcal{Y}} \\ x' & \xrightarrow{f} & y' \end{array}$$

Figure 3.1: An equivariant function $f\colon \mathcal{X} \to \mathcal{Y}$ commutes with respect to transformations $T_g^{(\cdot)}$ for all elements g of a certain group G.

3.2.1 Permutation Invariant and Equivariant Neural Networks

The concepts of invariance and equivariance play a crucial role in designing powerful neural building blocks. Invariance and equivariance are typically motivated by the intuition that incorporating task-relevant symmetries increases sample efficiency and provides generalization benefits, since a model without said symmetries baked-in will need to learn to ignore such spurious transformations (Cohen & Welling, 2016). Instead of relying on heavy data augmentations to let neural networks learn to be independent of transformations of a certain type, baked-in constraints of invariance or equivariance let models utilize these symmetries by design. We briefly recap their definitions and recent applications before we derive powerful graph-based neural networks from those concepts:

Let $T_g^{\mathcal{X}}\colon \mathcal{X} \to \mathcal{X}$ be a set of transformations on $\mathcal{X}$ for an element g of the abstract group G. A function $f\colon \mathcal{X} \to \mathcal{Y}$ is said to be *invariant* to G if (Satorras *et al.*, 2021)

$$f(T_g^{\mathcal{X}}(x)) = f(x) \quad \text{for all} \quad g \in G \text{ and } x \in \mathcal{X}. \tag{3.1}$$

As such, invariance refers to the independence of outcome w.r.t. to any transformations $T_g^{\mathcal{X}}$ of a certain type. Similarly, we can define the concept of equivariance.

A function $f\colon \mathcal{X} \to \mathcal{Y}$ is said to be *equivariant* to G if there exists an transformation $T_g^{\mathcal{Y}}\colon \mathcal{Y} \to \mathcal{Y}$ to $T_g^{\mathcal{X}}$ on its output space such that

$$f(T_g^{\mathcal{X}}(x)) = T_g^{\mathcal{Y}}(f(x)) \quad \text{for all} \quad g \in G \text{ and } x \in \mathcal{X}, \tag{3.2}$$

i.e. the diagram given in Figure 3.1 has to commute (Satorras *et al.*, 2021). Informally, the mapping f is equivariant if first transforming the input and then applying the function f on it will deliver the same result as first running the function f and then applying a transformation on its output.

Note that $T_g^{\mathcal{X}}$ and $T_g^{\mathcal{Y}}$ do not necessarily need to be the same. Specifically, invariant functions are a special case of equivariant functions where $T_g^{\mathcal{Y}}$ is set to be the identity transformation (Cohen & Welling, 2016; Satorras *et al.*, 2021). Furthermore, a function composition of equivariant functions preserves the property of equivariance, *i.e.* $g(f(T_g^{\mathcal{X}}(x))) = T_g^{\mathcal{Z}}(g(f(x)))$ for any composable equivariant functions $f\colon \mathcal{X} \to \mathcal{Y}$ and $g\colon \mathcal{Y} \to \mathcal{Z}$. Practically, equivariance aims to preserve transformations and can be realized in many different ways, such as (Satorras *et al.*, 2021; Maron *et al.*, 2019b; Keriven & Peyré, 2019):

- **Translation equivariance:** Translating the input results in an equivalent translation of the output, *i.e.* $f(x+g) = f(x) + g$ for all elements g of the translation group $(\mathbb{R}, +)$.

- **Rotation equivariance:** Rotating the input results in an equivalent rotation of the output, *e.g.* $f(\boldsymbol{M}\boldsymbol{x}) = \boldsymbol{M}f(\boldsymbol{x})$ for all elements $\boldsymbol{M}$ of the 3-dimensional rotation group $SO(3)$ (the set of 3×3 orthogonal matrices with determinant 1).

- **Permutation equivariance:** Permuting the input results in an equivalent permutation of the output, *i.e.* $f(\boldsymbol{\pi} \circ \boldsymbol{x}) = \boldsymbol{\pi} \circ f(\boldsymbol{x})$ for all elements $\boldsymbol{\pi}$ of the symmetric group S_N, *i.e.* the set of all $N!$ possible bijections from $\{1, \ldots, N\}$ to itself, with $\boldsymbol{\pi} \circ [x_1, x_2, \ldots, x_N] = [x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(N)}]$. Note that there exists alternative formulations for representing elements of the symmetric group S_N. For example, any permutation $\boldsymbol{\pi} \in S_N$ can also be described by its corresponding *permutation matrix* $\boldsymbol{P} \in \{0,1\}^{N \times N}$ with $P_{i,j} = 1$ only if $\pi(i) = j \; \forall i \in [N]$. Utilizing this notation, permutation equivariance is defined as $f(\boldsymbol{P}\boldsymbol{x}) = \boldsymbol{P}(f(\boldsymbol{x}))$.

Convolutional Neural Networks (CNNs) (LeCun *et al.*, 1998) are a designated class of translation equivariant neural networks, since, intuitively, patterns of objects are interesting irrespective of where they are located in an image. Recent efforts enhance CNNs by allowing them to be equivariant to a wider range of symmetries (*e.g.*, w.r.t. rotation and reflection), known as Group Equivariant Convolutional Networks (GECNs) (Cohen & Welling, 2016), leading to improved performance on real-world applications such as digital pathology segmentation (Veeling *et al.*, 2018).

For operating on graph-structured data $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ we are particularly seeking for invariance and equivariance w.r.t. permutations, since nodes of a graph are typically not assumed to be given in any meaningful order (Satorras *et al.*, 2021; Maron *et al.*, 2019b; Keriven & Peyré, 2019). Specifically, two graphs are considered to be the same in case they only differ by permutation of nodes, a concept known as graph isomorphism, *cf.* Section 2.1. Naturally, we expect our neural network building blocks to preserve such isomorphisms, *i.e.* $f(\mathcal{G}_1) = f(\mathcal{G}_2)$ or $\boldsymbol{\pi} \circ f(\mathcal{G}_1) = f(\mathcal{G}_2)$ for any two isomorphic graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ with $\mathcal{G}_1 = \boldsymbol{\pi} \circ \mathcal{G}_2$, $\boldsymbol{\pi} \in S_N$.

For the following explanation, we assume that the graph $\mathcal{G}$ does not contain any edges, *i.e.* $\mathcal{E} = \emptyset$, which leads to the simplified setting of learning over a set of nodes $\mathcal{V} = \{v_1, v_2, \ldots, v_N\}$. Here, we additionally associate each node $v_i \in \mathcal{V}$ with a F-dimensional feature vector $\boldsymbol{x}_i \in \mathbb{R}^F$. The union of node features can then either be represented as a multiset $\mathcal{X} = \{\!\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}\!\}$ (allowing potential duplicates) or as a node feature matrix $\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N]^\top \in \mathbb{R}^{N \times F}$. In order to derive any permutation invariant set function, *i.e.* $f(\boldsymbol{P}\boldsymbol{X}) = f(\boldsymbol{X})$, we can decompose f into the form

$$f(\mathcal{X}) = p\left(\sum_{\boldsymbol{x} \in \mathcal{X}} q(\boldsymbol{x})\right) \quad \text{or} \quad f(\boldsymbol{X}) = p\left(\sum_{i=1}^{N} q(\boldsymbol{x}_i)\right), \tag{3.3}$$

as proposed in the *DeepSet* model (Zaheer *et al.*, 2017). Here, $q(\cdot)$ operates *locally* over each element in the set, after which all elements are *globally* aggregated or *pooled* into a single representation before finally transformed via $p(\cdot)$. As shown in Zaheer *et al.* (2017), this decomposition is in fact known to be universal[1], *i.e.* every permutation invariant function f operating on sets can be represented by this decomposition. Training such a permutation invariant set function can be achieved by parametrizing both p and q, *i.e.* $p_{\boldsymbol{\theta}}$ and $q_{\boldsymbol{\theta}}$, typically done by realizing them as two separate Multi-Layer Perceptrons (MLPs) (Zaheer *et al.*, 2017). Notably, permutation invariance is

[1]Universality generally holds for countable sets and uncountable sets of fixed size.

achieved via the usage of the permutation invariant summation operator, however, other options are applicable as well as long as they are differentiable, *e.g.*, taking the mean or maximum over a multiset of node features (although universality will no longer hold, *cf.* Section 3.4.2). For example, Qi *et al.* (2017a) propose to use $f_{\boldsymbol{\theta}}(\mathcal{X}) = p_{\boldsymbol{\theta}}\big(\max_{\boldsymbol{x} \in \mathcal{X}} q_{\boldsymbol{\theta}}(\boldsymbol{x})\big)$ for learning on unordered point clouds $\mathcal{X} = \{\!\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}\!\}$, $\boldsymbol{x}_i \in \mathbb{R}^3$.

The aforementioned decomposition is a powerful blueprint for obtaining permutation invariant set-level outputs. However, it is not applicable for obtaining permutation equivariant node-level outputs, *i.e.* $f(\boldsymbol{P}\boldsymbol{X}) = \boldsymbol{P}f(\boldsymbol{X})$, due to the aggregation of element-wise representations into a global representation. Therefore, without assuming or inferring any additional structure, equivariance dictates that each node's features need to be transformed in isolation, *i.e.* $f(\boldsymbol{x}_i) = q_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$, discarding any information outside the current element under consideration.

3.2.2 A Recipe for Graph Neural Networks

Building upon the principles of learning on unordered sets in Section 3.2.1, we now consider the case of learning on graphs $\mathcal{G}$ with non-empty edge sets, *i.e.* $\mathcal{E} \neq \emptyset$, and derive a powerful recipe for defining equivariant Graph Neural Networks, framed as neural message passing (Gilmer *et al.*, 2017). By considering non-empty edge sets $\mathcal{E}$, our developed model will be able to exploit structural and feature-based information between distinct elements of $\mathcal{V}$. In particular, the edge $(v, w) \in \mathcal{E}$ denotes a relationship between two nodes $v, w \in \mathcal{V}$ which can be utilized in order to derive more accurate predictions. Considering such underlying structural properties in our neural network is typically referred to as injecting a *relational bias* into our function approximator (Battaglia *et al.*, 2018).

Let once again each node $v \in \mathcal{V}$ be associated with a multi-dimensional feature representation $\boldsymbol{x}_v \in \mathbb{R}^F$. Our Graph Neural Network model $f_{\boldsymbol{\theta}}(\boldsymbol{X}, \boldsymbol{A})$ operates on both stacked node features $\boldsymbol{X} \in \mathbb{R}^{|\mathcal{V}| \times F}$ as well as (sparse) adjacency matrix information $\boldsymbol{A} = \{0,1\}^{|\mathcal{V}| \times |\mathcal{V}|}$ induced by $\mathcal{G}$, and generates a set of output node embeddings $\boldsymbol{Z} \in \mathbb{R}^{|\mathcal{V}| \times F'}$. We will examine how to handle more sophisticated graph representations (such as additional edge features) in Section 3.3. In particular, we require $f_{\boldsymbol{\theta}}(\boldsymbol{X}, \boldsymbol{A})$ to be permutation equivariant, *i.e.*

$$\boldsymbol{P} f_{\boldsymbol{\theta}}(\boldsymbol{X}, \boldsymbol{A}) = f_{\boldsymbol{\theta}}(\boldsymbol{P}\boldsymbol{X}, \boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^\top) \quad \text{for all } \boldsymbol{P} \in S_N. \tag{3.4}$$

Here, $\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^\top$ permutes both rows and columns of $\boldsymbol{A}$ according to $\boldsymbol{P}$ simultaneously, *i.e.* $(\boldsymbol{P}\boldsymbol{A}\boldsymbol{P})_{i,j} = A_{\pi(i),\pi(j)}$, which is a spurious transformation that does not change the actual topology of $\mathcal{G}$.

The concrete recipe of message passing Graph Neural Networks can be derived from the DeepSet model (Zaheer *et al.*, 2017) in which the data information flow is further constrained to operate in a purely *local* fashion, exploiting the local sub-structure information around each node in $\mathcal{G}$, *cf.* Figure 3.2. Specifically, to guarantee permutation equivariance, $f_{\boldsymbol{\theta}}$ is applied *node-wise* according to

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}_v, \{\!\{\boldsymbol{x}_w : w \in \mathcal{N}(v)\}\!\}) = \textsc{Update}_{\boldsymbol{\theta}}\Big(\boldsymbol{x}_v, \bigoplus_{w \in \mathcal{N}(v)} \textsc{Message}_{\boldsymbol{\theta}}(\boldsymbol{x}_w, \boldsymbol{x}_v)\Big), \tag{3.5}$$

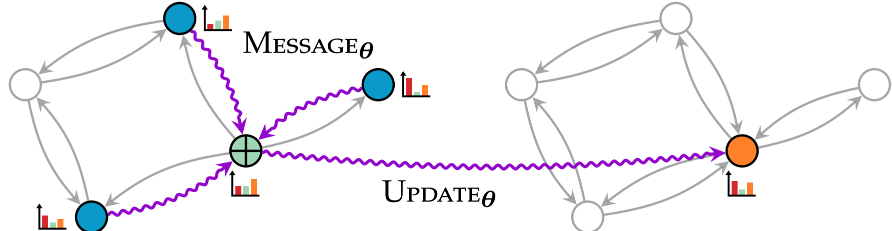


Figure 3.2: Illustration of the neural message passing scheme. For a given node ■, its neighbors ■ craft individual messages based on their representations (as indicated by the bar charts), which get then passed along their edges to the target node ■. Finally, aggregated messages are used to derive a new target node representation ■.

decomposed into a MESSAGE$_{\boldsymbol{\theta}}$ function, an aggregative function $\bigoplus$, and an UPDATE$_{\boldsymbol{\theta}}$ function (Hamilton, 2020).[2] Here, neural message passing refers to the process of exchanging information between nearby nodes, as each neighbor crafts an individual *message* that gets *passed* along its edge. In particular, MESSAGE$_{\boldsymbol{\theta}}$ and UPDATE$_{\boldsymbol{\theta}}$ denote arbitrary differentiable and parametrized functions, *i.e.* neural networks, and $\bigoplus$ defines how the information of each neighbor (as given by MESSAGE$_{\boldsymbol{\theta}}$) is aggregated into a vectorial representation $\boldsymbol{m}_{\mathcal{N}(v)} = \bigoplus_{w \in \mathcal{N}(v)} \textsc{Message}_{\boldsymbol{\theta}}(\boldsymbol{x}_w, \boldsymbol{x}_v)$. Finally, UPDATE$_{\boldsymbol{\theta}}(\boldsymbol{x}_v, \boldsymbol{m}_{\mathcal{N}(v)})$ refines the node's representation $\boldsymbol{x}_v$ based on $\boldsymbol{m}_{\mathcal{N}(v)}$. For learning meaningful patterns in the graph, the message passing functions need to be trainable and differentiable, but each node utilizes the *same* function for updating its representations, *i.e.* the set of parameters is shared across across all data. This does not only lead to faster convergence, but also allows the model to be applicable to unseen nodes or graphs as well, *cf.* Section 3.2.3.

It is easy to see that $f_{\boldsymbol{\theta}}$ is permutation equivariant by design as long as $\bigoplus$ defines a permutation-invariant aggregative function. However, while $f_{\boldsymbol{\theta}}$ is permutation equivariant, the aggregation of messages is constrained to be invariant w.r.t. the ordering of the underlying multiset of neighborhood features $\{\!\{\boldsymbol{x}_w \in w \in \mathcal{N}(v)\}\!\}$. Notably, there exists numerous works (Murphy *et al.*, 2019a,b; Maron *et al.*, 2019b; de Haan *et al.*, 2020; Satorras *et al.*, 2021) that allow for obtaining less constrained set-level representations during neighborhood aggregation. However, they are not the main focus of this dissertation.

In a Graph Neural Network, message passing operators $f_{\boldsymbol{\theta}}$ are typically stacked and enhanced by non-linear activation functions, *cf.* Figure 3.3, just like in a conventional deep neural network, *i.e.*

$$\boldsymbol{h}_v^{(\ell)} = f_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{h}_v^{(\ell-1)}, \{\!\{\boldsymbol{h}_w^{(\ell-1)} : w \in \mathcal{N}(v)\}\!\}) \tag{3.6}$$

with $\ell \in \{1, \ldots, L\}$ and L denoting the number of layers. We set $\boldsymbol{h}_v^{(0)} = \boldsymbol{x}_v$. Here, at each iteration, every node aggregates the current information from its local neighborhood. By recursively applying this scheme, the embeddings of a node from subsequent layers contain more and more information from farther nodes in the graph. For example, in the first layer ($\ell = 1$), the message passing phase will enrich the feature representation of each node by the features of its immediate neighbors. After the

[2] MESSAGE and UPDATE are common notations in literature (Hamilton, 2020).

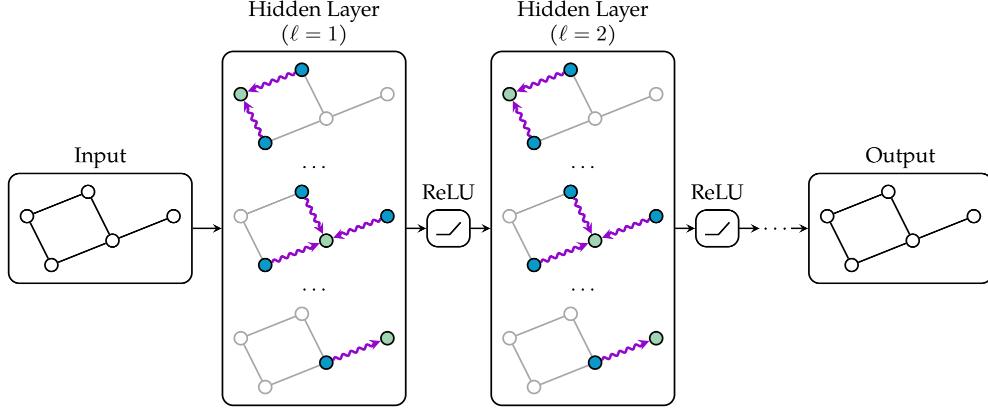


Figure 3.3: A Graph Neural Network model architecture. Message passing operators are stacked and enhanced by non-linear activation functions to obtain a final embedding $\boldsymbol{h}_v^{(L)}$ for all nodes $v \in \mathcal{V}$ that encodes structural and feature information of its L-hop neighborhood.

second layer ($\ell = 2$), each node embedding will therefore contain information about its 2-hop neighborhood. In general, a node has aggregated its L-hop structural and feature information after exactly L layers. The final node representation $\boldsymbol{h}_v^{(L)}$ can then be trained and utilized for a given down-stream task in an end-to-end fashion.

3.2.3 (Un)supervised Deep Learning on Graphs

Graph Neural Networks, as described in Section 3.2.2, can be viewed as a process of learning node representations by embedding their individual local L-hop subgraphs. These representations can be either learned via end-to-end (semi-)supervised or unsupervised training (Kipf & Welling, 2017), and are useful for a variety of tasks. For node-level tasks, such as node classification, the output embeddings produced by a GNN can directly be used for deriving node-level predictions. For other tasks, such as graph-level or edge-level predictions, representation learning of node features is typically implemented as an intermediate step.

3.2.3.1 (Semi-)Supervised Node Classification. The task of (semi-)supervised node classification is one of the most popular applications for GNNs (Kipf & Welling, 2017; Veličković *et al.*, 2018; Wu *et al.*, 2019a; Chen *et al.*, 2020b), *e.g.*, classifying academic papers in citation networks (Sen *et al.*, 2008; Yang *et al.*, 2016) or predicting product categories in co-purchase graphs (Hu *et al.*, 2020a). Here, each node $v \in \mathcal{V}$ is associated with a label y_v, and the goal is to learn a representation $\boldsymbol{h}_v^{(L)}$ from which y_v can be easily predicted, *i.e.* $\phi_{\boldsymbol{\theta}}(\mathcal{G}, v) = \psi_{\boldsymbol{\theta}}(\boldsymbol{h}_v^{(L)}) = y_v$, where $\phi_{\boldsymbol{\theta}}$ denotes the machine learning model and $\psi_{\boldsymbol{\theta}}$ represents a final classifier. Typically, the classifier $\psi_{\boldsymbol{\theta}}$ is either omitted (utilizing $\boldsymbol{h}_v^{(L)}$ directly as predictions) or modeled as an MLP, such that $\phi_{\boldsymbol{\theta}}$ denotes an end-to-end deep neural network pipeline. Algorithm 1 presents the general blueprint for training a GNNs for (semi-)supervised node classification tasks, in which node embeddings are first learned through a GNN (line 1) and inputted into

Algorithm 1 GNN: Node Classification Training

Require: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, Node features $\boldsymbol{X}$, Training labels $\mathcal{Y}_{\text{train}}$
1: $\{\!\{\boldsymbol{h}_1^{(L)}, \boldsymbol{h}_2^{(L)}, \ldots, \boldsymbol{h}_{|\mathcal{V}|}^{(L)}\}\!\} \leftarrow \text{GNN}_{\boldsymbol{\theta}}(\boldsymbol{X}, \mathcal{G})$
2: $\mathcal{L} \leftarrow \frac{1}{|\mathcal{V}_{\text{train}}|} \sum_{v \in \mathcal{V}_{\text{train}}} - \log \psi_{\boldsymbol{\theta}}\big(\boldsymbol{h}_v^{(L)}\big)_{y_v}$
3: Update model parameters $\boldsymbol{\theta}$ of $\text{GNN}_{\boldsymbol{\theta}}$ and $\psi_{\boldsymbol{\theta}}$ w.r.t. $\partial \mathcal{L} / \partial \boldsymbol{\theta}$

a classifier (line 2). Model training is then be performed via negative log-likelihood (line 2), back-propagation and gradient descent (line 3).

One typically distinguishes between *transductive* and *inductive* node property prediction tasks, resulting in a semi-supervised or supervised training scenario, respectively (Hamilton, 2020). In the transductive learning task, we are only given a small subset of labeled ground-truth nodes as training data $\mathcal{V}_{\text{train}} \subseteq \mathcal{V}$, while we utilize the full graph for learning meaningful node representations, *i.e.* unlabeled nodes are not used during loss computation, but are still involved in the GNN's message passing phases. In the inductive case, the model does not observe any data outside the given training set. This typically represents the case in which a model is trained in a fully-supervised manner, and then applied to unseen examples afterwards.

It is of further importance to see that the training procedure given in Algorithm 1 operates on the *complete* graph in a *full-batch* fashion, *i.e.* it computes node embeddings $\{\!\{\boldsymbol{h}_1^{(L)}, \boldsymbol{h}_2^{(L)}, \ldots, \boldsymbol{h}_N^{(L)}\}\!\}$ of all nodes $v \in \mathcal{V}$, not just for the nodes $\mathcal{V}_{\text{train}}$ involved in the actual loss computation. While not strictly necessary, this procedure is common in Graph Neural Network training (Kipf & Welling, 2017). In particular, it allows us access to *all* hidden node embeddings in *all* layers which is necessary to resolve the inter-dependencies of node embeddings, *i.e.* $\boldsymbol{h}_v^{(\ell+1)}$ with $v \in \mathcal{V}_{\text{train}}$ may well depend on node embedding $\boldsymbol{h}_w^{(\ell)}$ where $w \notin \mathcal{V}_{\text{train}}$. Since such a computation flow is not feasible in large-scale graphs due to memory limitations and slow convergence (Ma & Tang, 2020), we will take a closer look at mini-batching techniques for GNNs in Chapter 5.

3.2.3.2 Link Prediction. Link prediction describes the process of finding missing or future links in an incomplete or ever-evolving graph, which has wide practical applications, *e.g.*, in providing friend recommendations in social networks (Adamic & Adar, 2003) or in knowledge graph completion (Nickel *et al.*, 2016). One common practice for tackling the task of link prediction is to first derive node representations through a GNN, after which source node and target node representations are aggregated to predict the probability of corresponding link, *i.e.* $\phi_{\boldsymbol{\theta}}(\mathcal{G}, v, w) = \psi_{\boldsymbol{\theta}}(\boldsymbol{h}_v^{(L)}, \boldsymbol{h}_w^{(L)})$ (Kipf & Welling, 2016; Zhang & Chen, 2018; Zhang *et al.*, 2020a). Here, the final predictor $\psi_{\boldsymbol{\theta}}$ is typically represented as an MLP operating on the concatenated node representations, *i.e.* $\psi_{\boldsymbol{\theta}}(\boldsymbol{h}_v^{(L)}, \boldsymbol{h}_w^{(L)}) = \text{MLP}([\boldsymbol{h}_v^{(L)}, \boldsymbol{h}_w^{(L)}]) \in [0, 1]$, where its output denotes the probability of a link between $(v, w) \in \mathcal{V} \times \mathcal{V}$. In case of operating on an undirected graph $\mathcal{G}$, $\psi_{\boldsymbol{\theta}}$ can be further constrained to be *commutative*, *e.g.*, via averaging $(\boldsymbol{h}_v^{(L)} + \boldsymbol{h}_w^{(L)})/2$ or element-wise multiplication $\boldsymbol{h}_v^{(L)} \odot \boldsymbol{h}_w^{(L)}$ (Grover & Leskovec, 2016).

In contrast to the aforementioned node classification task, link prediction is usually framed as a *contrastive* learning task (Hamilton, 2020) that learns to distinguish between positive and negative edges, *cf.* Algorithm 2. In particular, given a distinct sub-

Algorithm 2 GNN: Link Prediction Training

Require: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, Node features $\boldsymbol{X}$, Training edges $\mathcal{E}_{\text{train}}$
1: $\bar{\mathcal{E}}_{\text{train}} \sim (\mathcal{V} \times \mathcal{V}) \setminus (\mathcal{E} \cup \mathcal{E}_{\text{train}})$
2: $\{\!\{\boldsymbol{h}_1^{(L)}, \boldsymbol{h}_2^{(L)}, \ldots, \boldsymbol{h}_N^{(L)}\}\!\} \leftarrow \mathrm{GNN}_{\boldsymbol{\theta}}(\boldsymbol{X}, \mathcal{G})$
3: $\mathcal{L} \leftarrow \frac{1}{|\mathcal{E}_{\text{train}}|} \sum_{(v,w) \in \mathcal{E}_{\text{train}}} - \log \psi_{\boldsymbol{\theta}}\big(\boldsymbol{h}_v^{(L)}, \boldsymbol{h}_w^{(L)}\big) + \frac{1}{|\bar{\mathcal{E}}_{\text{train}}|} \sum_{(v,w) \in \bar{\mathcal{E}}_{\text{train}}} - \log \big(1 - \psi_{\boldsymbol{\theta}}\big(\boldsymbol{h}_v^{(L)}, \boldsymbol{h}_w^{(L)}\big)\big)$
4: Update model parameters $\boldsymbol{\theta}$ of $\mathrm{GNN}_{\boldsymbol{\theta}}$ and $\psi_{\boldsymbol{\theta}}$ w.r.t. $\partial \mathcal{L} / \partial \boldsymbol{\theta}$

set of training edges $\mathcal{E}_{\text{train}} \cap \mathcal{E} = \emptyset$, our model is trained to separate positive edges $\mathcal{E}_{\text{train}}$ from negative ones $(\mathcal{V} \times \mathcal{V}) \setminus (\mathcal{E} \cup \mathcal{E}_{\text{train}})$ via a binary cross entropy formulation

$$\mathcal{L} = \sum_{(v,w) \in \mathcal{E}_{\text{train}}} - \log \psi_{\boldsymbol{\theta}}\big(\boldsymbol{h}_v^{(L)}, \boldsymbol{h}_w^{(L)}\big) + \sum_{(v,w) \sim (\mathcal{V} \times \mathcal{V}) \setminus (\mathcal{E} \cup \mathcal{E}_{\text{train}})} - \log \big(1 - \psi_{\boldsymbol{\theta}}\big(\boldsymbol{h}_v^{(L)}, \boldsymbol{h}_w^{(L)}\big)\big), \tag{3.7}$$

(line 3), in which negative edges are uniformly sampled from the complete set of negative edges (line 1). It is worth noting that, given the standard machine learning scenario of hold-out validation and test sets, it might well happen that positive validation edges and test edges appear as negative samples during training. We are not allowed to acknowledge their existence during training to prevent any data leakage. Although this seems to be counter-intuitive at first glance, it is to be expected that such a noise in the learning signal is negligible.

3.2.3.3 Graph Classification. The task of graph classification refers to predicting the properties of an entire graph. Specifically, molecular learning is one the most popular real-world applications for graph classification and used to infer certain properties of a molecule/molecular graph (Hu *et al.*, 2020a). Here, each graph within a set of graphs $\mathcal{G} \in \{\mathcal{G}_1, \mathcal{G}_2, \ldots\}$ is associated with a label $y_{\mathcal{G}}$, and the goal is to learn a graph-level representation $\boldsymbol{h}_{\mathcal{G}} \in \mathbb{R}^F$ from which $y_{\mathcal{G}}$ can be easily predicted. Similar to the task of link prediction, GNNs are utilized as an intermediate step that first learn meaningful node representations, *i.e.* $\phi_{\boldsymbol{\theta}}(\mathcal{G}) = \psi_{\boldsymbol{\theta}}(\textsc{Readout}(\{\!\{\boldsymbol{h}_1^{(L)}, \boldsymbol{h}_2^{(L)}, \ldots, \boldsymbol{h}_N^{(L)}\}\!\})) = y_{\mathcal{G}}$ (Zhang *et al.*, 2018; Xu *et al.*, 2019c; Morris *et al.*, 2019), where the READOUT function is used to derive $\boldsymbol{h}_{\mathcal{G}}$ from a set of features in a permutation-invariant fashion, drawing further analogies to the DeepSet model (Zaheer *et al.*, 2017) introduced in Section 3.2.1. Often, READOUT is implemented as a simple averaging of node representations (Hamilton, 2020)

$$\boldsymbol{h}_{\mathcal{G}} = \textsc{Readout}(\{\!\{\boldsymbol{h}_1^{(L)}, \boldsymbol{h}_2^{(L)}, \ldots, \boldsymbol{h}_N^{(L)}\}\!\}) = \frac{1}{|\mathcal{V}|} \sum_{v \in \mathcal{V}} \boldsymbol{h}_v^{(L)}, \tag{3.8}$$

although other, more sophisticated operators are applicable as well, *e.g.* guided by attention (Li *et al.*, 2016b; Vinyals *et al.*, 2016; Zhang *et al.*, 2018). We will introduce alternative READOUT functions in Section 3.2.4. Training a graph-level model $\phi_{\boldsymbol{\theta}}$ is done in a supervised, end-to-end fashion via negative log-likelihood, *cf.* Algorithm 3, similar to the task of node classification.

One limitation of the global READOUT approach is that it does not explicitly exploit the global graph structure while predicting a label associated with an entire graph: This task is solely attached to the preceding GNN, which is, however, only able to detect patterns in local substructures around each node. As a result, recent works propose to

Algorithm 3 GNN: Graph Classification Training

Require: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, Node features $\boldsymbol{X}$, Training label y
1: $\{\!\{\boldsymbol{h}_1^{(L)}, \boldsymbol{h}_2^{(L)}, \ldots, \boldsymbol{h}_N^{(L)}\}\!\} \leftarrow \mathrm{GNN}_{\boldsymbol{\theta}}(\boldsymbol{X}, \mathcal{G})$
2: $\boldsymbol{h}_{\mathcal{G}}^{L} \leftarrow \textsc{Readout}\big(\{\!\{\boldsymbol{h}_1^{(L)}, \boldsymbol{h}_2^{(L)}, \ldots, \boldsymbol{h}_N^{(L)}\}\!\}\big)$
3: $\mathcal{L} \leftarrow -\log \psi_{\boldsymbol{\theta}}\big(\boldsymbol{h}_{\mathcal{G}}^{(L)}\big)_y$
4: Update model parameters $\boldsymbol{\theta}$ of $\mathrm{GNN}_{\boldsymbol{\theta}}$ and $\psi_{\boldsymbol{\theta}}$ w.r.t. $\partial\mathcal{L}/\partial\boldsymbol{\theta}$

utilize *hierarchical* Graph Representation Learning approaches that integrate coarse-grained graph structures, *e.g.*, meaningful clusters, at intermediate steps in the model pipeline (Defferrard *et al.*, 2016; Ying *et al.*, 2018b; Cangea *et al.*, 2018; Gao & Ji, 2019). We will take a closer look at hierarchical graph learning approaches in Section 4.3, where we will also introduce our own domain-specific solution (Fey *et al.*, 2020b) for learning on molecular graphs.

3.2.3.4 Unsupervised Graph Representation Learning. The process of discovering novel or interesting graph structures without any presence of ground-truth labels is commonly referred to as unsupervised Graph Representation Learning (Veličković *et al.*, 2019). Since most graphs in the wild are unlabeled, unsupervised graph learning is essential for many real-world applications, and can be utilized in various different ways. For example, output node embeddings can be directly used for a given downstream task (Grover & Leskovec, 2016; Hamilton *et al.*, 2017; Veličković *et al.*, 2019), or *pre-trained* model parameters are used as weight initializations for further supervised learning (Hu *et al.*, 2020b). In general, unsupervised learning is especially useful in cases where we have access to a large amount of data of which only a small subset of data is actually labeled.

The predominant approaches for unsupervised Graph Representation Learning utilize random walk objectives which aim to learn *individual* node embeddings such that nodes that are close in the input graph are also close in the output embedding space, *cf.* Section 2.2. However, such methods will over-emphasize proximity information at the expense of structural information, and can only be applied in transductive learning tasks (Perozzi *et al.*, 2014; Grover & Leskovec, 2016). These limitations encourage the development of unsupervised learning methods via stronger encoder models based on message passing GNNs (Veličković *et al.*, 2019).

Training GNNs in an unsupervised fashion revolves around the development of suitable objective functions. For example, Hamilton *et al.* (2017) proposed to utilize the link prediction objective of Equation (3.7) for pre-training. Veličković *et al.* (2019) maximize the *mutual information* between node embeddings $\boldsymbol{h}_v^{(L)}$ and graph-level representations $\boldsymbol{h}_{\mathcal{G}} = \textsc{Readout}(\{\!\{\boldsymbol{h}_1^{(L)}, \ldots, \boldsymbol{h}_N^{(L)}\}\!\})$, *i.e.*

$$\mathcal{L} = -\log(\psi_{\boldsymbol{\theta}}(\boldsymbol{h}_v^{(L)}, \boldsymbol{h}_{\mathcal{G}})) - \log(1 - \psi_{\boldsymbol{\theta}}(\tilde{\boldsymbol{h}}_v^{(L)}, \boldsymbol{h}_{\mathcal{G}})), \tag{3.9}$$

where $\psi_{\boldsymbol{\theta}}$ denotes a discriminator network and $\tilde{\boldsymbol{h}}_v^{(L)}$ represents an embedding of node v generated via a *corrupted* version of $\mathcal{G}$, *e.g.*, by shuffling node features or by disturbing edge connectivity. Alternative pre-training strategies involve node-level and graph-level self-supervised tasks, such as context prediction, *i.e.* whether the same node appears in two different subgraphs, attribute masking, *i.e.* the prediction of randomly

masked input features, or based on auxiliary supervised graph-level prediction tasks (Hu *et al.*, 2020b), *e.g.*, by deriving the graph edit distance between two graphs (Bai *et al.*, 2018; Bia *et al.*, 2019).

3.2.4 Design Principles of Graph Neural Networks

So far, we introduced the basic message passing GNN blueprint via three differentiable and parametrized abstract functions, *i.e.* $\textsc{Message}_{\boldsymbol{\theta}}$, $\bigoplus$ and $\textsc{Update}_{\boldsymbol{\theta}}$, *cf.* Equation (3.5). Those functions can be chosen in many different ways, depending on the task at hand: For example, MESSAGE functions can transform incoming features either linearly or non-linearly (Gilmer *et al.*, 2017; Qi *et al.*, 2017a; Wang *et al.*, 2019e), aggregative functions $\bigoplus$ can model static (Morris *et al.*, 2019; Xu *et al.*, 2019c), structure-dependent (Kipf & Welling, 2017) or data-dependent aggregations (Veličković *et al.*, 2018), and UPDATE is typically used to preserve central node information via skip-connections (Hamilton *et al.*, 2017) or residuals (Klicpera *et al.*, 2019a; Chen *et al.*, 2020b). We briefly review how current state-of-the-art GNN operators fit into the given neural message passing scheme (Section 3.2.4.1 — 3.2.4.2), and discuss recently evolving principles in Graph Neural Network design (Section 3.2.4.3 — 3.2.4.6).

3.2.4.1 Concrete Message Passing GNN Instantiations. On of the most popular instance of message passing GNNs (Kipf & Welling, 2017; Hamilton *et al.*, 2017; Veličković *et al.*, 2018) is defined as

$$\boldsymbol{h}_v^{(\ell)} = f_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{h}_v^{(\ell-1)}) = \sigma\Big(\sum_{w \in \mathcal{N}(v) \cup \{v\}} \frac{1}{C_{w,v}} \boldsymbol{W}^{(\ell)} \boldsymbol{h}_w^{(\ell-1)} + \boldsymbol{b}^{(\ell)} \Big), \tag{3.10}$$

where $\boldsymbol{W}^{(\ell)}$ and $\boldsymbol{b}^{(\ell)}$ represent weight and bias parameters in layer $\ell \in \{1, \ldots, L\}$, respectively, $C_{w,v}$ is a normalization coefficient, and σ denotes an element-wise non-linearity, *e.g.*, ReLU (Glorot *et al.*, 2011). $\mathcal{N}(v) \cup \{v\}$ denotes a node's neighborhood with added self-loop. Notably, this formulation shares strong similarities to a traditional fully-connected layer (*cf.* Section 2.3), and is, in fact, equivalent in case $\mathcal{N}(v) = \emptyset$. Central node features $\boldsymbol{h}_v^{(\ell-1)}$ and neighboring node features $\boldsymbol{h}_w^{(\ell-1)}$ are first transformed and combined using linear operations, after which an element-wise non-linearity is applied. However, in contrast to the fully-connected layer formulation, the output node representation both depends on its previous as well as its neighboring node feature representations. Note that each applied transformation will make use of the *same* set of parameters, utilizing the concept of shared weights (LeCun *et al.*, 1998). As such, this Graph Neural Network implementation can be understood as a low-pass filter, where node feature vectors are *smoothed* across local neighborhoods (NT & Maehara, 2019).

We can easily verify that the GNN instantiation of Equation (3.10) fits into the given neural message passing scheme, *i.e.*

$$\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{h}_w^{(\ell-1)}, \boldsymbol{h}_v^{(\ell-1)}) = \frac{1}{C_{w,v}} \boldsymbol{W}^{(\ell)} \boldsymbol{h}_w^{(\ell-1)} + \boldsymbol{b}^{(\ell)}, \ \bigoplus = \sum, \text{ and } \textsc{Update}_{\boldsymbol{\theta}}^{(\ell)} = \sigma.$$

The specific design of $C_{w,v}$ has attracted a lot of research and leads to different GNN capabilities, *cf.* Section 3.4. For example, aggregations can be modeled to be static,

i.e. $C_{w,v} = 1$ (Morris *et al.*, 2019; Xu *et al.*, 2019c) or structure-dependent, *i.e.* $C_{w,v} = |\mathcal{N}(v)|$ or $C_{w,v} = \sqrt{|\mathcal{N}(w)||\mathcal{N}(v)|}$ (Hamilton *et al.*, 2017; Kipf & Welling, 2017). The latter is known as symmetric normalization (motivated by the normalized graph Laplacian matrix (Defferrard *et al.*, 2016)), which weakens the contribution of highly-connected neighbors to the output. Furthermore, $C_{w,v}$ can also be modeled in a data-dependent fashion, *i.e.* guided by attention, in which the contribution of a neighbor's feature vector is directly determined by its transmitted data (Veličković *et al.*, 2018; Vaswani *et al.*, 2017; Shi *et al.*, 2020b; Kim & Oh, 2021).

In contrast to the aforementioned GNN instances, alternative GNN building blocks introduce concepts such as dropping non-linearities completely (Wu *et al.*, 2019a; Klicpera *et al.*, 2019a; Frasca *et al.*, 2020), degree-dependent parameter sharing (Duvenaud *et al.*, 2015; Monti *et al.*, 2017; Fey *et al.*, 2018), or anisotropic and multi-layer transformations (Gilmer *et al.*, 2017; Simonovsky & Komodakis, 2017; Qi *et al.*, 2017b; Wang *et al.*, 2019e).

3.2.4.2 Attentional Aggregation. In attention-guided GNNs, known as *Graph Attention Networks* (*GATs*), the aforementioned normalization coefficient $C_{w,v}$ is determined by the feature representations $\boldsymbol{h}_w^{(\ell-1)}$ and $\boldsymbol{h}_v^{(\ell-1)}$ rather than by the underlying graph structure. Here, the basic idea is to derive an edge-level attention score (Bahdanau *et al.*, 2015) which lets the model directly determine the influence of individual neighbors for the given learning task, either formulated via

(1) additive (Veličković *et al.*, 2018) or (2) multiplicative (Shi *et al.*, 2020b)

$$\tilde{C}_{w,v}^{(\ell)} = \boldsymbol{w}^{(\ell)\top}[\boldsymbol{h}_w^{(\ell-1)}, \boldsymbol{h}_v^{(\ell-1)}] \qquad \tilde{C}_{w,v}^{(\ell)} = \left(\boldsymbol{W}^{(\ell)}\boldsymbol{h}_w^{(\ell-1)}\right)^\top \cdot \boldsymbol{W}^{(\ell)}\boldsymbol{h}_v^{(\ell-1)}$$

attention. Attention scores $\tilde{C}_{w,v}$ are then further normalized *across* neighborhoods via

$$C_{w,v} = \mathrm{softmax}(\tilde{C}_{\cdot,v})_w = \frac{\exp(\tilde{C}_{w,v})}{\sum_{u \in \mathcal{N}(v)} \exp(\tilde{C}_{u,v})}. \tag{3.11}$$

Since one generally wants to attend to more than just a single neighbor, attentional GNNs usually employ a *multi-head* formalism (Vaswani *et al.*, 2017), utilizing K independent attention mechanisms in parallel

$$\boldsymbol{h}_v^{(\ell)} = [f_{\boldsymbol{\theta}_1}(\boldsymbol{h}_v^{(\ell-1)}), f_{\boldsymbol{\theta}_2}(\boldsymbol{h}_v^{(\ell-1)}), \ldots, f_{\boldsymbol{\theta}_K}(\boldsymbol{h}_v^{(\ell-1)})]. \tag{3.12}$$

Notably, attentional GNNs are closely related to the popular *Transformer* architecture (Vaswani *et al.*, 2017) in natural language processing. In fact, attention-guided GNNs are equivalent to Transformers in case they are operating on a fully-connected graph (Joshi, 2020), which let them be seen as a sparse variant of the Transformer model. However, it has been shown that attentional GNNs may fail to attend to meaningful neighbors (Knyazev *et al.*, 2019b), and auxilary supervision of attention scores can drastically boost down-stream performance, *e.g.*, via an additional link prediction objective (Kim & Oh, 2021).

3.2.4.3 Residual Graph Networks. One of the potential disadvantages of the aforementioned GNN instantiation in Equation (3.10) is its low-pass filtering characteristic.

In particular, since self-loops are directly merged into the underlying graph structure, the GNN has difficulties to preserve original central node information. This is especially noticeable in deep GNNs, where node representations become increasingly *washed out* w.r.t. model depth, with node representations becoming gradually more indistinguishable from each other while converging to a stationary point (Xu *et al.*, 2018). We will take a closer look at over-smoothing in GNNs in Section 4.2.

A natural way to alleviate this problem is to utilize $\textsc{Update}_{\boldsymbol{\theta}}$ for preserving the already acquired node information from previous rounds of message passing. The most popular approach in doing so is to employ a simple (learnable) skip-connection (Hamilton *et al.*, 2017)

$$\textsc{Update}^{(\ell)}_{\boldsymbol{\theta}}(\boldsymbol{h}_v^{(\ell-1)}, \boldsymbol{m}_{\mathcal{N}(v)}^{(\ell)}) = \sigma(\boldsymbol{W}^{(\ell)} \boldsymbol{h}_v^{(\ell-1)} + \boldsymbol{m}_{\mathcal{N}(v)}^{(\ell)}). \tag{3.13}$$

Inspiration has also been drawn from the "gating" mechanism (Cho *et al.*, 2014) employed in recurrent neural networks, in which *Gated Recurrent Units* (*GRUs*) (Cho *et al.*, 2014) are used

$$\textsc{Update}^{(\ell)}_{\boldsymbol{\theta}}(\boldsymbol{h}_v^{(\ell-1)}, \boldsymbol{m}_{\mathcal{N}(v)}^{(\ell)}) = \mathrm{GRU}_{\boldsymbol{\theta}}(\boldsymbol{h}_v^{(\ell-1)}, \boldsymbol{m}_{\mathcal{N}(v)}^{(\ell)}) \tag{3.14}$$

to gate newly incoming neighborhood information (Li *et al.*, 2016b). Alternative ideas involve strategies such as "teleporting" back to (or restarting from) earlier representations (Klicpera *et al.*, 2019a)

$$\textsc{Update}^{(\ell)}_{\boldsymbol{\theta}}(\boldsymbol{h}_v^{(0)}, \boldsymbol{m}_{\mathcal{N}(v)}^{(\ell)}) = \alpha\, \boldsymbol{h}_v^{(0)} + (1-\alpha)\, \boldsymbol{m}_{\mathcal{N}(v)}^{(\ell)}, \tag{3.15}$$

which guarantee that initial representations $\boldsymbol{h}_v^{(0)}$ are retained with "teleport probability" $\alpha \in [0,1]$ (Klicpera *et al.*, 2019a; Chen *et al.*, 2020b). Furthermore, Xu *et al.* (2018) proposed a complimentary strategy framed as *Jumping Knowledge* (*JK*) that utilizes layer-wise jump connections and selective aggregations of intermediate representations to allow for node-adaptive neighborhood ranges. Given layer-wise representations $\boldsymbol{h}_v^{(1)}, \boldsymbol{h}_v^{(2)}, \ldots, \boldsymbol{h}_v^{(L)}$ of node v, its final output representation is obtained by either

$$\begin{array}{ccccc} \text{(1) concatenation,} & \text{(2) pooling} & \text{or} & \text{(3) attention} \\ [\boldsymbol{h}_v^{(1)}, \boldsymbol{h}_v^{(2)}, \ldots, \boldsymbol{h}_v^{(L)}] & \max_{\ell=1}^{L} \boldsymbol{h}_v^{(\ell)} & & \sum_{\ell=1}^{L} \alpha_v^{(\ell)} \boldsymbol{h}_v^{(\ell)}, \end{array} \tag{3.16}$$

where attention scores $\alpha_v^{(\ell)}$ are obtained from a bi-directional *Long Short-Term Memory* (*LSTM*) (Hochreiter & Schmidhuber, 1997).

3.2.4.4 Normalization Techniques. Normalization layers shift and scale intermediate representations and are known to help optimization of deep neural networks, leading to more stable and faster convergence. For example, *Batch Normalization* (Ioffe & Szegedy, 2015) can also be applied in the graph domain, where we standardize node representations via

$$\boldsymbol{h}_v^{(\ell)} \leftarrow \boldsymbol{\gamma}^{(\ell)} \frac{\boldsymbol{h}_v^{(\ell)} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} + \boldsymbol{\beta}^{(\ell)}, \tag{3.17}$$

with $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ denoting the mean and standard deviation of $\{\!\{\boldsymbol{h}_1^{(\ell)}, \boldsymbol{h}_2^{(\ell)}, \ldots, \boldsymbol{h}_N^{(\ell)}\}\!\}$, respectively, and $\boldsymbol{\gamma}^{(\ell)}$ and $\boldsymbol{\beta}^{(\ell)}$ are learnable parameters. Lately, there has been interest in developing specific graph normalization layers that take the inter-dependency

characteristics of nodes into account while maintaining (sub-)graph structural properties (Cai *et al.*, 2020). For example, Cai *et al.* (2020) propose to re-scale the mean $\boldsymbol{\mu}$ by trainable parameters in order to avoid the loss of structural information present in the mean statistics. Analogously, Zhou *et al.* (2020) propose a group-wise normalization procedure that utilizes a soft grouping mechanism of nodes into G groups via learnable attention weights $\boldsymbol{\alpha}_v \in \mathbb{R}^G$:

$$\boldsymbol{h}_v^{(\ell)} \leftarrow \sum_{g=1}^{G} \textsc{BatchNorm}_{\boldsymbol{\theta}}^{(g)}(\mathrm{softmax}(\boldsymbol{\alpha}_v)_g \cdot \boldsymbol{h}_v^{(\ell)}). \tag{3.18}$$

These techniques not only ensure that GNNs converge faster to a local optimum, but can also partially negate the effects of over-smoothing (Zhao & Akoglu, 2020; Zhou *et al.*, 2020).

3.2.4.5 Readout Layers. In graph classification tasks, the Readout function is used to aggregate a multiset of features into a single representation in a permutation-invariant fashion. Simple as well as more sophisticated variants for instantiating Readout have been proposed. The most common way is to simply take the sum, the mean or the maximum over all node embeddings in the graph, *cf.* Section 3.2.3, which is often sufficient for graph-level applications involving small graphs. Cangea *et al.* (2018) further propose to combine the outputs of such simple set aggregators in order to facilitate learning. More sophisticated variants integrate the concepts of attention into Readout as well. For example, Li *et al.* (2016b) make use of attention as a gating mechanism

$$\boldsymbol{h}_{\mathcal{G}} = \sum_{v \in \mathcal{V}} \mathrm{softmax}(\boldsymbol{H}^{(L)}\boldsymbol{w})_v \, \boldsymbol{h}_v^{(L)}. \tag{3.19}$$

In a similar fashion, Vinyals *et al.* (2016) leverage attention-based aggregations iteratively for fine-grained refinement of the global output vector. As an alternative to attention, Zhang *et al.* (2018) propose a Readout layer based on sorting. Here, the top-k node feature vectors are stacked in descending order based on their last feature vector channel, which are further processed via a traditional CNN. Notably, this procedure still guarantees permutation-invariance in the overall architecture.

3.2.4.6 Heterogeneous Graph Learning. So far, we assumed deep neural networks operating on simple homogeneous graphs, while in practice, graphs may have much richer information and semantic structure, *e.g.*, in knowledge graphs. In particular, in *heterogeneous graphs* $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T}_V, \mathcal{T}_E)$, nodes and edges belong to *disjunct* node and edge types, and, most importantly, representations of different types are potentially embedded in different feature spaces (Zhu *et al.*, 2019b; Yu *et al.*, 2020), *cf.* Section 2.1. One common approach for allowing GNNs to operate on such heterogeneous graph data is to utilize separate message passing formalisms for each edge type, and combine their individual outputs afterwards (Schlichtkrull *et al.*, 2018)

$$\boldsymbol{h}_v^{(\ell)} = \sum_{r \in \Sigma_E} \bigoplus_{w \in \mathcal{N}_r(v)} \textsc{Message}_{\boldsymbol{\theta}_r}^{(\ell)}(\boldsymbol{h}_w^{(\ell-1)}, \boldsymbol{h}_v^{(\ell-1)}), \tag{3.20}$$

where $\mathcal{N}_r(v)$ denotes the neighborhood set around node v induced by edges of type $r \in \Sigma_E$. Overall, this multi-relational aggregation is analogous to the traditional GNN

formulation, except that it now aggregates information across different types separately. As such, instantiations of this scheme can as well make use of recent advancements in Graph Representation Learning. For example, Hu *et al.* (2020c) implement $\textsc{Message}^{(\ell)}_{\boldsymbol{\theta}_r}$ via graph attention. However, one major issue of Equation (3.20) is the rapid growth in the number of parameters w.r.t. the number of edge types $|\Sigma_E|$, leading to over-fitting on rare relations. Therefore, Schlichtkrull *et al.* (2018) leverage a regularization scheme via *basis-decomposition*

$$\boldsymbol{h}_v^{(\ell)} = \sum_{r \in \Sigma_E} \sum_{b=1}^{B} \bigoplus_{w \in \mathcal{N}_r(v)} \alpha_{r,b} \cdot \textsc{Message}^{(\ell)}_{\boldsymbol{\theta}_b}(\boldsymbol{h}_w^{(\ell-1)}, \boldsymbol{h}_v^{(\ell-1)}) \tag{3.21}$$

with learnable scalars $\alpha_{r,b} \in \mathbb{R}$ for each edge type $r \in \Sigma_E$ and basis $b \in \{1, \ldots, B\}$. Notably, $\alpha_{r,b}$ are the only relation-specific parameters in this approach.

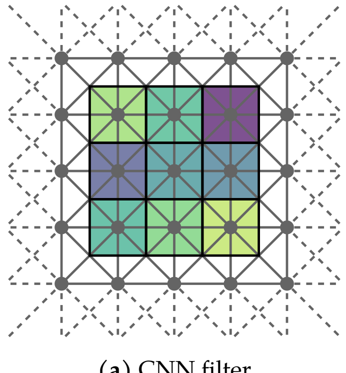
3.3 Edge-Conditioned Message Passing via B-Spline Kernels

Most achievements obtained by deep learning methods over the last years heavily rely on properties of the convolution operation in Convolutional Neural Networks (CNNs) (LeCun *et al.*, 1998), *i.e.* local connectivity, weight sharing and shift invariance. Since those layers are defined on inputs with a grid-like structure, they are not trivially portable to non-Euclidean domains like graphs or discrete manifolds. However, learning on grid-like structures can be understood as a special case of graph-based machine learning, in which the graph is restricted to present sequences, 2D squares or 3D cube lattices with regular connectivity between neighboring data points. As a result, transferring and generalizing the high performance of traditional CNNs to a more general class of data holds the potential for large improvements in several relevant tasks.

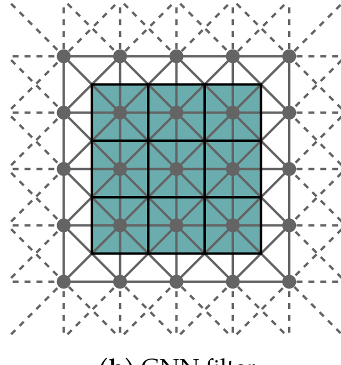
Notably, message passing GNNs as introduced in Section 3.2 already utilize highly similar concepts of convolution on regular domains. In particular, they share the principles of locality — traditional CNNs perform a learnable aggregative transformation of local patches, while GNNs perform a learnable aggregative transformation of local neighborhoods (Shuman *et al.*, 2013). Additionally, both make use of the weight sharing principle across different patches and neighborhoods, leading to translation and permutation equivariance in the image and graph domain, respectively.

However, in comparison to CNNs, commonly utilized GNN operators are also more restrictive and limited, which hurts the potential generality of models we can build (Huszár, 2016). To illustrate this, consider the GNN operator (Kipf & Welling, 2017) of Equation (3.10), *i.e.* $\boldsymbol{h}_v^{(\ell)} = \sum_{w \in \mathcal{N}(v) \cup \{v\}} \frac{1}{C_{w,v}} \boldsymbol{W}^{(\ell)} \boldsymbol{h}_w^{(\ell-1)}$, that processes an (infinitely large) 2D lattice, *cf.* Figure 3.4. Such a GNN does not know about the directions of edges (such as bottom right or top left), which amounts to a CNN containing only *center-surround patterns*, *e.g.*,

$$\boldsymbol{K}_{i,j}^{(\ell)} = \begin{bmatrix} W_{i,j}^{(\ell)} & W_{i,j}^{(\ell)} & W_{i,j}^{(\ell)} \\ W_{i,j}^{(\ell)} & W_{i,j}^{(\ell)} & W_{i,j}^{(\ell)} \\ W_{i,j}^{(\ell)} & W_{i,j}^{(\ell)} & W_{i,j}^{(\ell)} \end{bmatrix} \tag{3.22}$$



(**a**) CNN filter



(**b**) GNN filter

Figure 3.4: Comparison of CNN and GNN filters on regular grids. In a (**a**) CNN each neighbor is transformed via individual weights, while in a (**b**) GNN each neighbor is transformed equally.

with $\boldsymbol{K}_{i,j}^{(\ell)}$ denoting a 3×3 filter matrix for input and output feature map i and j, respectively (Huszár, 2016).[3] This is to be expected, as graphs are arbitrary abstract structures that naturally do not come with directional descriptors. However, in this application, a GNN inherently fails to learn about any patterns present in the input.

This observation gives rise to the question on how we can model a general class of Graph Neural Networks that can effectively incorporate anisotropy in the form of directional descriptors. One solution towards this goal is presented in our *Spline-Based Convolutional Neural Network* (*SplineCNN*) architecture (Fey *et al.*, 2018). SplineCNNs denote a variant of deep Graph Neural Networks for irregularly structured and geometric input, *e.g.*, graphs or discrete manifolds, which generalize the traditional CNN convolution operator via a continuous kernel formulation (Section 3.3.1) and fully resemble CNNs on discrete input (Section 3.3.2).

3.3.1 Graph Convolution via Continuous B-Spline Kernels

While aforementioned GNN operators (Section 3.2) assume that information is solely encoded in the connectivity, edge weights and node features of the input, our spline-based convolution operator leverages multi-dimensional edge features to facilitate the learning of anisotropic kernel functions. For example, for images or meshes, additional information such as the relative positions of nodes is present in the input data, which we consider with our method. Therefore, similar to the work of Monti *et al.* (2017), we expect the input of our convolution operator to be a *directed* graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \boldsymbol{E})$, where $\boldsymbol{E} \in [-1, 1]^{|\mathcal{E}| \times D}$ contains D-dimensional *edge features* or *pseudo-coordinates* for each directed edge $(v, w) \in \mathcal{E}$. Our convolution operator aggregates node features in local neighborhoods weighted by a trainable, continuous kernel function.

The input node features $\boldsymbol{H}^{(\ell-1)} \in \mathbb{R}^{|\mathcal{V}| \times F}$ represent features on an irregular geometric structure, whose spatial relations are locally defined by the pseudo-coordinates $\boldsymbol{E}$. Therefore, when locally aggregating feature values in a node's neighborhood, the

[3] Normalization coefficients $C_{w,v}$ are omitted as they are constant in regular graphs.

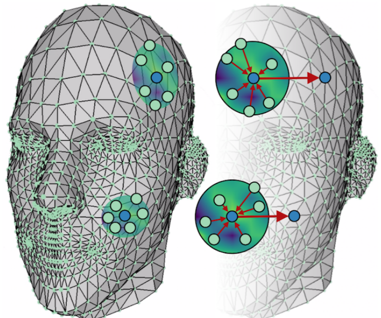


Figure 3.5: An example of deep mesh data processing (Fey *et al.*, 2018)**.**

content of $\boldsymbol{E}$ is used to determine *how* the features are aggregated and the content of $\boldsymbol{H}^{(\ell-1)}$ defines *what* is aggregated. We argue that common irregular inputs can be mapped to this model while preserving relevant information:

- For **graphs**, $\mathcal{V}$ and $\mathcal{E}$ are given and $\boldsymbol{E}$ can contain user-defined edge features or degree-related information such as $\boldsymbol{e}_{w,v} = 1/\sqrt{|\mathcal{N}(w)||\mathcal{N}(v)|}$.

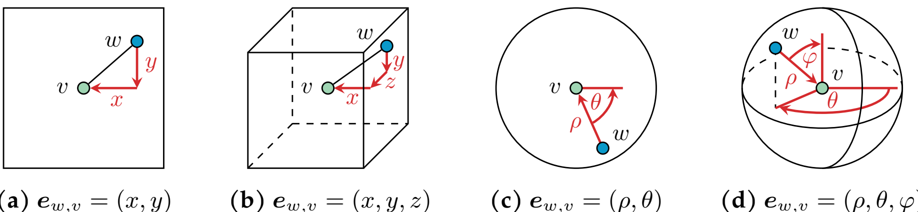
- For **meshes** or **discrete manifolds**, $\mathcal{V}$ contains points of the discrete manifold, $\mathcal{E}$ represents connectivity in local Euclidean neighborhoods and $\boldsymbol{E}$ can contain local relational information like polar, spherical or Cartesian coordinates of the source node in respect to the target point for each edge, *cf.* Figure 3.5.

- For **point clouds**, $\mathcal{V}$ is given such that each node v is associated with a point $\boldsymbol{p}_v \in \mathbb{R}^3$, and $\mathcal{E}$ can be obtained synthetically, *e.g.*, via *k-nearest neighbor search* $\mathcal{E} = \{(w, v) : w \in \mathcal{N}_{\leq k}(\boldsymbol{p}_v)\}$ with $\mathcal{N}_{\leq k}(\boldsymbol{p}_v)$ containing the k nearest points of $\boldsymbol{p}_v$, or via *ball query search* $\mathcal{E} = \{(w, v) : \|\boldsymbol{p}_w - \boldsymbol{p}_v\| < r\}$ based on radius r. Here, $\boldsymbol{E}$ holds geometric relations, *e.g.*, via Cartesian coordinates $\boldsymbol{e}_{w,v} = \boldsymbol{p}_w - \boldsymbol{p}_v$.

- Other forms of geometric graphs are **scene graphs**, in which nodes in $\mathcal{V}$ represent objects or entities in a given scene, and $\mathcal{E}$ describes their geometric as well as semantic relationships as part of their edge features $\boldsymbol{E}$.

In general, SplineCNN state no restriction on the values of $\boldsymbol{E}$, except being element of a fixed interval range. Therefore, meshes, for example, can be either interpreted as embedded three-dimensional graphs or as two-dimensional manifolds, using local Euclidean neighborhood representations like obtained by the work of Boscaini *et al.* (2016). Also, either polar/spherical coordinates or Cartesian coordinates can be used, as shown in Figure 3.6. Independent from the type of coordinates stored in $\boldsymbol{E}$, our trainable, continuous kernel function, which we define in the following, maps each $\boldsymbol{e}_{w,v}$ to a matrix used for an edge-conditioned transformation of node representations.

3.3.1.1 Spline-based Convolution Operator. Our spline-based convolution operator utilizes a *continuous* kernel formulation $g_{\boldsymbol{\theta}}^{(\ell)} : \mathbb{R}^D \to \mathbb{R}^{F \times F'}$ conditioned on the pseudo-coordinates $\boldsymbol{e}_{w,v} \in \mathbb{R}^D$ that defines the specific and anisotropic transforma-



(a) $\boldsymbol{e}_{w,v} = (x, y)$ (b) $\boldsymbol{e}_{w,v} = (x, y, z)$ (c) $\boldsymbol{e}_{w,v} = (\rho, \theta)$ (d) $\boldsymbol{e}_{w,v} = (\rho, \theta, \varphi)$

Figure 3.6: Possibilities for pseudo-coordinates $\boldsymbol{e}_{w,v}$: (**a**) Two- and (**b**) three-dimensional Cartesian, (**c**) polar and (**d**) spherical coordinates. Values for scaling and translating $\boldsymbol{e}_{w,v}$ to $[-1, 1]^D$ are omitted (Fey *et al.*, 2018).

tion applied to each individual neighbor $\boldsymbol{h}_w^{(\ell-1)} \in \mathbb{R}^F$:

$$\boldsymbol{h}_v^{(\ell)} = C_{w,v}^{-1} \sum_{w \in \mathcal{N}(v) \cup \{v\}} \textsc{Message}^{(\ell)}(\boldsymbol{h}_w^{(\ell-1)}, \boldsymbol{e}_{w,v}) = C_{w,v}^{-1} \sum_{w \in \mathcal{N}(v) \cup \{v\}} g_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{e}_{w,v}) \cdot \boldsymbol{h}_w^{(\ell-1)}. \tag{3.23}$$

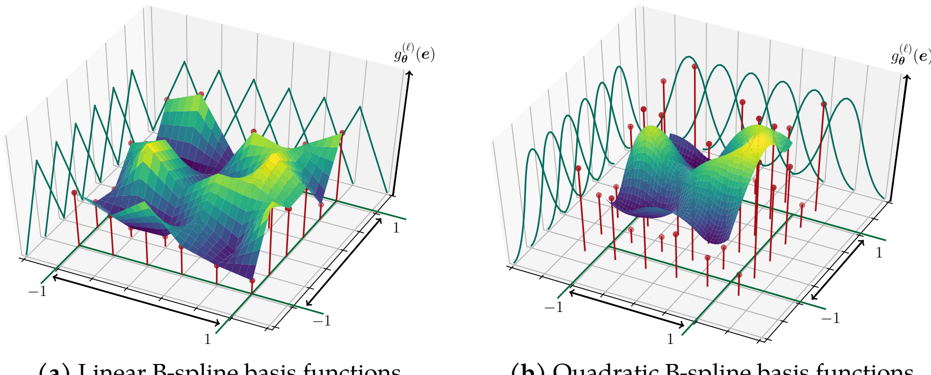
with $C_{w,v} = |\mathcal{N}(v)|$ denoting the coefficients for averaging. As in traditional CNNs, we utilize the pseudo-coordinates $\boldsymbol{e}_{w,v}$, *i.e.* the relative positions of nodes, to determine *how* input features $\boldsymbol{h}_w^{(\ell-1)}$ are aggregated. In particular, this scheme generalizes traditional CNNs to graph-structured data, as we can sample a transformation matrix via $g_{\boldsymbol{\theta}}^{(\ell)}$ for *any* given value $\boldsymbol{e}_{w,v} \in [-1, 1]^D$, not just at discrete stationary points. Furthermore, averaging messages across neighborhoods allows the model to better generalize to varying sampling densities that may be present in the underlying graph-structured data.

The definition of our continuous kernel $g_{\boldsymbol{\theta}}^{(\ell)}$ leverages the concept of B-spline bases (Piegl & Tiller, 1997), parametrized by a *constant* number of trainable control values. In particular, the local support property of B-spline basis function, which states that basis functions evaluate to zero for all inputs outside of a known interval, proves to be advantageous for efficient computation and scalability. In the following, we denote the set of *open* B-spline basis functions of degree m as $\{N_k^m : k \leq i \leq K\}$ based on K uniform, *i.e.* equidistant control points with knot vector $\boldsymbol{\tau}$, *cf.* Piegl & Tiller (1997).

Figure 3.7 visualizes the spline-based kernel formulation for differing B-spline basis degree. Here, we introduce a trainable weight matrix $\boldsymbol{W}_{\boldsymbol{p}}^{(\ell)} \in \mathbb{R}^{F \times F'}$ for each element $\boldsymbol{p}$ from the Cartesian product $\mathcal{P} = \{1, 2, \ldots, K\}^D$, resulting in $K^D \cdot F \cdot F'$ trainable parameters. In particular, K^D can be interpreted as the *kernel size* and is dependent on the dimensionality of pseudo-coordinates. With this, we define our continuous convolution kernel

$$g_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{e}) = \sum_{\boldsymbol{p} \in \mathcal{P}} \boldsymbol{W}_{\boldsymbol{p}}^{(\ell)} \prod_{d=1}^{D} N_{p_d}^m(e_d) \tag{3.24}$$

as a linear combination of weight matrices weighted by the product of basis functions along pseudo-coordinate dimensions. One way to interpret this kernel is to see the trainable parameters $W_{\boldsymbol{p},i,j}$ as control values for the height of a $(D+1)$-dimensional B-spline surface, from which a weight is sampled for each neighbor $w \in \mathcal{N}(v)$, dependent on $\boldsymbol{e}_{w,v}$. However, in contrast to traditional $(D+1)$-dimensional B-spline approximation, we only have one-dimensional control points and approximate functions instead of curves. To ensure that the unity property of the B-spline bases holds (Piegl & Tiller, 1997), we scale their knot vector $\boldsymbol{\tau}$ to exactly match the interval $[-1, 1]$.



(**a**) Linear B-spline basis functions (**b**) Quadratic B-spline basis functions

Figure 3.7: Examples of spline-based continuous kernels for B-spline basis degree (a) $m = 1$ and (b) $m = 2$ with pseudo-coordinate dimensionality $D = 2$. The heights of the red dots are the trainable parameters $\boldsymbol{W}^{(\ell)}_{:,i,j}$ for a single input and output feature map i and j, respectively. They are multiplied by the elements of the B-spline tensor product basis before influencing the kernel value (Fey *et al.*, 2018).

Depending on the type of pseudo-coordinate vectors $\boldsymbol{e}_{w,v}$, we can use *closed* B-spline approximation in some dimensions. One frequently occurring example of such a situation is when $\boldsymbol{e}_{w,v}$ contains angle attributes of polar coordinates. Using closed B-spline approximation in the angle dimension naturally enforces the angle 0 to be evaluated to the same weight as the angle 2π or, for higher m, the kernel function to be continuously differentiable at those points. The proposed kernels can easily be modified so that they use closed approximation in an arbitrary subset of the D dimensions by mapping different $\boldsymbol{p} \in \mathcal{P}$ to the same trainable parameters, leading to a reduction of trainable parameters and B-spline basis functions. Referring to Figure 3.7, this approach can be interpreted as a periodic repetition of the function surface along the corresponding axis.

Due to the local support property of B-splines, $N^m_k(e_d) \neq 0$ holds true for exactly $m + 1$ of the K different basis functions. Therefore, $g^{(\ell)}_{\boldsymbol{\theta}}$ only depends on $(m + 1)^D$ of the K^D B-spline bases for each neighbor w, where D and m are constant and usually small. In addition, for each pair of nodes $(w, v) \in \mathcal{E}$, the vectors $\boldsymbol{p} \in \mathcal{P}$ with non-zero contribution can be found in constant time, given constant D and m. This tremendously helps in reducing the time complexity of the given operation due to being independent of the actual kernel size K^D. In practice, we achieve independence from the number of trainable weights by computing matrices $\boldsymbol{P} \in \mathbb{N}^{|\mathcal{E}| \times (m+1)^D}$ and $\boldsymbol{B} \in \mathbb{R}^{|\mathcal{E}| \times (m+1)^D}$, *cf.* Algorithm 4. $\boldsymbol{P}$ contains the indices of non-zero product bases (line 3), while $\boldsymbol{B}$ contains their real values (line 4) according to $B_{e,i} = \prod_{d=1}^{D} N^m_{P_{e,i}}(e_d)$, *cf.* Equation (3.24). We can then use entries in $\boldsymbol{P}$ and $\boldsymbol{B}$ directly to query the corresponding weight matrix $\boldsymbol{W}$, to re-weight it, and to finally multiply it with the node feature matrix (line 8). For the evaluation of the basis functions required for $\boldsymbol{B}$, we use explicit low-degree polynomial formulations (Piegl & Tiller, 1997). In total, the forward operation of our convolution operator (as outlined in Algorithm 4) has a time complexity of $\mathcal{O}(|\mathcal{E}| \cdot (m + 1)^D \cdot F \cdot F')$. We will see in Chapter 6 how to efficiently parallelize this scheme on GPUs.

Algorithm 4 Spline-Based Convolution

Require: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \boldsymbol{E})$, Node features $\boldsymbol{H}^{(\ell-1)} \in \mathbb{R}^{|\mathcal{V}| \times F}$,
1: **for** each $(w, v), e \in \mathcal{E}$ **do**
2: **for** each $i \in \{1, \ldots, (m+1)^D\}$ **do**
3: $p \leftarrow \boldsymbol{P}[e, i]$
4: $b \leftarrow \boldsymbol{B}[e, i]$
5: **for** each $f' \in \{1, \ldots, F'\}$ **do**
6: $r \leftarrow 0$
7: **for** each $f \in \{1, \ldots, F\}$ **do**
8: $r \leftarrow r + b \cdot \boldsymbol{W}[p, f, f'] \cdot \boldsymbol{H}^{(\ell-1)}[w, f]$
9: **end for**
10: $\boldsymbol{H}^{(\ell)}[v, f'] \leftarrow r$
11: **end for**
12: **end for**
13: **end for**

3.3.1.2 Applications. Similar to traditional CNNs, our convolution operator can be used as a module in a deep neural network architecture, which we call SplineCNN. SplineCNNs can be applied on all kinds of tasks involving irregular structured data, *e.g.*, arbitrary (embedded) graphs and meshes, and allows for end-to-end training without using hand-crafted feature descriptors. For example, for learning on discrete manifolds, we can directly interpret mesh connectivity as graph-structured data and perform spatial aggregation around local mesh vertices. Notably, SplineCNN can directly learn geometric features from raw mesh data, while similar approaches (Masci *et al.*, 2015; Boscaini *et al.*, 2016; Litany *et al.*, 2017; Monti *et al.*, 2017) rely on hand-crafted feature descriptors as input node features, like the local histogram of normal vectors known as SHOT descriptors (Tombari *et al.*, 2010). In SplineCNN, input node features of meshes are trivially given by $\mathbf{1} \in \mathbb{R}^{|\mathcal{V} \times 1}$, and the network solely learns local geometric pattern from the spatial relations encoded in the pseudo-coordinates $\boldsymbol{E}$. Due to missing pre-processing, this allows for even faster processing of data. Superpixel images denote an alternative domain for learning on embedded graphs (Monti *et al.*, 2017; Fey *et al.*, 2018; Knyazev *et al.*, 2019a), in which both spatial relations and node features in the form of RGB values are important for the given down-stream task.

Notably, SplineCNN can be applied for learning on arbitrary graphs as well, and allows, in contrast to related approaches (Kipf & Welling, 2017), for an *anisotropic* transformation of neighboring node features, conditioned on edge features. Here, neighbors are transformed differently depending on graph characteristics, *e.g.*, node degree. For example, in citation graphs (Sen *et al.*, 2008; Yang *et al.*, 2016), it might be desirable to discriminate between highly cited works (such as general books) and less cited (but more specialized) works to determine the category of a paper more accurately.

3.3.1.3 Alternative Edge-Conditioned GNNs. An alternative strategy for edge-conditioned Graph Representation Learning was proposed in Gilmer *et al.* (2017); Simonovsky & Komodakis (2017), in which the continuous kernel function $g_{\boldsymbol{\theta}}^{(\ell)} : \mathbb{R}^D \to \mathbb{R}^{F \times F'}$ is implemented as an MLP, *i.e.*

$$\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{h}_w^{(\ell-1)}, \boldsymbol{e}_{w,v}) = g_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{e}_{w,v}) \cdot \boldsymbol{h}_w^{(\ell-1)} = \mathrm{MLP}_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{e}_{w,v}) \cdot \boldsymbol{h}_w^{(\ell-1)}, \tag{3.25}$$

rather than via B-spline approximation. In comparison, our scheme provides the following advantages:

- **SplineCNN is computationally efficient:** A kernel MLP needs to explicitly materialize an individual weight matrix for each edge, which leads to an overall memory-consumption of $\mathcal{O}(|\mathcal{E}| \cdot F \cdot F')$. SplineCNN naturally enforces a low memory footprint by only interpolating between a fixed number of weight matrices. Furthermore, the computation complexity of SplineCNN is independent of the utilized kernel size, which is not the case for MLPs.
- **SplineCNN output channels are inter-independent:** Following the principles of traditional CNNs and in contrast to kernel MLPs, SplineCNN defines individual filters that do not share parameters between different input and output channels.
- **SplineCNN utilizes a low amount of parameters:** The number of parameters in a kernel MLP drastically increases with MLP width and depth, leading to over-parametrization. In contrast, the parameters of SplineCNN solely depend on the chosen kernel size, which usually tends to be small.

However, implementing $g_{\boldsymbol{\theta}}^{(\ell)}$ as an MLP is advantageous in case of high-dimensional edge features, as our continuous B-spline kernels scale exponentially w.r.t. the dimensionality of pseudo-coordinates D. As a result, the best choice of implementing $g_{\boldsymbol{\theta}}^{(\ell)}$ clearly depends on the given task, with SplineCNNs being very efficient to train on low-dimensional relational input such as Cartesian relations in meshes or point clouds.

3.3.2 Relation to Traditional CNNs

Our spline-based convolution operator (*cf.* Section 3.3.1) can be seen as a direct generalization of the traditional convolutional layer in CNNs for *odd* filter sizes in each dimension. In particular, if we assume to have a D-dimensional grid-graph with diagonal, horizontal and vertical edges to be the input (*cf.* Figure 3.4), B-spline degree $m = 1$, kernel size 3×3, and the vectors $\boldsymbol{e}_{w,v}$ to contain Cartesian relations between adjacent nodes, then our convolution operator is equivalent to a discrete convolution of an image with a kernel of size 3×3. Here, each Cartesian relation $\boldsymbol{e} \in \{-1, 0, 1\}^D$ maps to exactly to one distinct weight matrix in the parameter space of SplineCNN, *i.e.* $\prod_{d=1}^{D} N_{p_d}^1(e_{w,v,d}) = 1$ for exactly one $\boldsymbol{p} \in \mathcal{P}$ and 0 otherwise. We provide further proof for two-dimensional grid-graphs in the following:

Let elements of $\boldsymbol{H}^{(\ell)}$ and $\boldsymbol{W}^{(\ell)}$ be both accessible via one or two-dimensional indices. Then,

$$\boldsymbol{h}_v^{(\ell)} = \sum_{w \in \mathcal{N}(v) \cup \{v\}} \left(\sum_{\boldsymbol{p} \in \{1,2,3\}^2} \boldsymbol{W}_{\boldsymbol{p}}^{(\ell)} \prod_{d=1}^{2} N_{p_d}^1(e_{w,v,d}) \right) \cdot \boldsymbol{h}_w^{(\ell-1)} \tag{3.26}$$

$$= \sum_{\boldsymbol{e}=(i,j) \in \{-1,0,1\}^2} \left(\sum_{\boldsymbol{p} \in \{1,2,3\}^2} \boldsymbol{W}_{\boldsymbol{p}}^{(\ell)} \prod_{d=1}^{2} N_{p_d}^1(e_d) \right) \cdot \boldsymbol{h}_{v+i,v+j}^{(\ell-1)}. \tag{3.27}$$

With $N^1_{e_d+2}(e_d) = 1$ for $e \in \{-1, 0, 1\}$, it immediately follows that

$$\boldsymbol{h}_v^{(\ell)} = \sum_{(i,j) \in \{-1,0,1\}^2} \boldsymbol{W}^{(\ell)}_{i+2,j+2} \cdot \boldsymbol{h}^{(\ell-1)}_{v+i,v+j}, \tag{3.28}$$

resembling the definition of traditional CNNs (LeCun *et al.*, 1998). Notably, this relation holds for any D-dimensional grid-graph, and we utilized $D = 2$ simply due to ease of notation. Interestingly, this relation even holds true on grid-graph borders, assuming that the CNN is applied with appropriately sized zero-padding. Furthermore, it is important to see that we dropped the normalization coefficients $C_{w,v}$ of SplineCNN in Equation (3.26). Since we operate on fixed-sized graphs, $C_{w,v}$ is constant and therefore only amounts to a negligible global scaling factor.

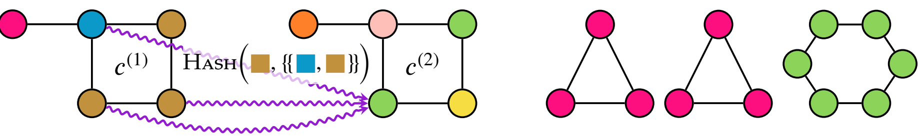
The generalization capabilities of SplineCNN also hold for larger discrete kernels beyond sizes of 3×3. However, for this, the neighborhoods of the grid-graph need to be modified accordingly, *e.g.*, in a kernel size of 5×5, a node needs to be connected to its "original" 2-hop neighborhood.

The equivalence of CNNs and SplineCNNs on discrete input data can also be verified experimentally, leading to *exactly the same* outputs across consecutive layers and identical model performance, given the same parameter initialization. Notably, the continuous kernel formulation has the potential advantage (1) to account for larger receptive field sizes in a more parameter efficient manner, (2) to handle partially-observed data in an elegant fashion, and (3) to be deployed at resolutions other than those observed during training (Romero *et al.*, 2021).

3.4 Maximally Expressive Graph Neural Networks

The design of novel GNN operators has been largely tackled based on intuition and empirical justification, as there is little theoretical understanding of the representational properties and limitations of GNNs. In particular, a neural network model for graph-structured data should be able to learn representations of nodes in a graph, taking both the graph structure and feature description of nodes into account. While it is known that GNNs are a powerful tool for feature aggregation across irregularly structured data (Kipf & Welling, 2017; Veličković *et al.*, 2018), it is unclear how well GNNs are actually doing in reasoning about and encoding structural properties of the underlying graph. Ideally, a maximally powerful GNN is able to map isomorphic graphs to the same representation in the embedding space, while it maps non-isomorphic ones to different representations. Such an ability, however, implies solving the challenging graph isomorphism problem, *cf.* Section 2.1. Here, we offer a theoretical exploration of a GNN's expressive power in distinguishing non-isomorphic graph structures by relating their power to the *WL algorithm* (Weisfeiler & Lehman, 1968), a powerful heuristic that successfully solves the graph isomorphism test for a broad class of graphs (Section 3.4.1)[4]. As a result, we show that there exist GNN architectures and parameter initializations that provably inherit the expressivity of the WL algorithm.

[4]In this thesis, we use the more wide-spread "Lehman" spelling, although the less-known "Leman" is the preferred spelling, *cf.* `https://www.iti.zcu.cz/wl2018/pdf/leman.pdf` (last access: August 25, 2022).



(a) WL color refinement iteration

(b) Indistinguishable graphs by WL

Figure 3.8: (a) Illustration of the update rule of the WL algorithm for a single iteration: The HASH function computes, for each node, an injective new coloring based on the current colors of the node itself and its direct neighbors (Morris *et al.*, 2021a). **(b) Two graphs (■ and ■) that cannot be distinguished by the WL:** Each node will obtain the same coloring since the node degree is shared across all nodes in both graphs (Morris *et al.*, 2021a).

Since the power of the WL has been completely characterized (Arvind *et al.*, 2015; Kiefer *et al.*, 2015), we can transfer this results to the case of GNNs, showing that both approaches share the same benefits and shortcomings. Going further, we leverage this theoretical relationship to propose a generalization of GNNs, called *k-dimensional GNNs* (*k-GNNs*) (Morris *et al.*, 2019), which describe neural architectures based on the *k-dimensional Weisfeiler-Lehman* (*k-WL*) algorithm (Cai *et al.*, 1992), that are strictly more powerful than GNNs in distinguishing graph structures (Section 3.4.2). The key insight in these *higher-dimensional* variants is that they perform message passing directly between subgraph structures rather than between individual nodes. As a result, this higher-order form of message passing can capture structural information that is not visible at the node-level.

3.4.1 Relation to the Weisfeiler-Lehman Isomorphism Test

The graph isomorphism problem (Section 2.1) asks whether two graphs are topologically identical, which is a challenging problem as no polynomial-time algorithm is known for it yet (Garey, 1979; Garey & Johnson, 2002; Babai, 2016). The *Weisfeiler-Lehman* or *color refinement* algorithm (Weisfeiler & Lehman, 1968) is a well-known heuristic for deciding whether two graphs are isomorphic: Given an initial coloring or labeling of the nodes of both graphs, two nodes with the same label get different labels if the number of identically labeled neighbors is not equal. More formally, for a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and iteration $\ell \geq 1$, the WL algorithm recursively refines a node coloring $c^{(\ell)} \colon \mathcal{V} \to \Sigma$ based on the intermediate coloring of its neighbors

$$c_v^{(\ell)} = \textsc{Hash}\left(c_v^{(\ell-1)}, \{\!\{c_w^{(\ell-1)} : w \in \mathcal{N}(v)\}\!\}\right), \tag{3.29}$$

where HASH injectively maps the above pair to a unique value in Σ which has not been used in previous iterations. That is, in each iteration, the algorithm computes a new color for a node based on the colors or features of its direct neighbors, *cf.* Figure 3.8a for an illustration. We initialize $c_v^{(0)}$ by either assigning the same color to all nodes (in case of unlabeled graphs) or based on some initially given discrete node labeling (in case of labeled graphs). After L iterations, the color of a node reflects the structure of its L-hop neighborhood. To test if two graphs $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ and $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$ are non-isomorphic, we run the above algorithm in "parallel" on both graphs until convergence, *i.e.* until the number of colors between two iterations does not change.

Termination is guaranteed after at most $\max\{|\mathcal{V}_1|, |\mathcal{V}_2|\}$ iterations (Grohe, 2017). If the two graphs have a different histogram of node colorings, the WL concludes that the graphs are not isomorphic.

The Weisfeiler-Lehman algorithm constitutes one of the earliest approaches to graph isomorphism testing (Weisfeiler, 1976; Weisfeiler & Lehman, 1968; Immerman & Lander, 1990), and has been heavily investigated by the graph theory community over the last few decades (Grohe *et al.*, 2014). Notably, its power has been completely characterized (Arvind *et al.*, 2019; Kiefer *et al.*, 2015). Apart from some corner cases, it has been shown that this simple algorithm is already an effective and computationally efficient strategy to decide whether two graphs are non-isomorphic (Babai *et al.*, 1980), and has been therefore applied in many different areas (Grohe *et al.*, 2014; Kersting *et al.*, 2014; Li *et al.*, 2016a; Yao & Holder, 2015; Zhang & Chen, 2017). On the other hand, it is easy to see that the algorithm cannot distinguish *all* non-isomorphic graphs (Cai *et al.*, 1992). For example, the WL fails to distinguish all k-regular graphs of same node size, *cf.* Figure 3.8b.

Shervashidze *et al.* (2011) were the first that used the WL algorithm as a graph kernel, the so-called *Weisfeiler-Lehman subtree kernel*. The kernel's idea is to compute the WL coloring for a fixed number of steps, resulting in a color histogram or feature vector for each graph. The kernel is then computed by taking the pairwise inner product between these vectors. Hence, the kernel measures the similarity between two graphs by counting the number of common colors in all refinement steps, inputted into a linear SVM. Numerous other graph kernel variants utilize the concepts of WL colors (Kriege *et al.*, 2016; Nikolentzos *et al.*, 2017; Togninalli *et al.*, 2019). For example, Kriege *et al.* (2016) proposes the *Weisfeiler-Lehman optimal assignment kernel*, where the WL colors are used as a similarity measure between pairs of nodes. Furthermore, extensions of the WL subtree kernel have been proposed that can deal with continuous node information (Orsini *et al.*, 2015; Morris *et al.*, 2016). Although already quite powerful, these kernels are limited by the fact that they cannot effectively adapt their feature representations to a given data distribution since they rely on a fixed set of pre-computed features.

Interestingly, a single iteration of the WL algorithm shares high similarities with the message passing scheme employed in GNNs, *cf.* Equation (3.5), where the $\textsc{Hash}$ function is replaced by differentiable and trainable $\textsc{Message}_{\boldsymbol{\theta}}$ and $\textsc{Update}_{\boldsymbol{\theta}}$ functions utilized for neighborhood aggregation (Kipf & Welling, 2017). As the WL is a powerful tool for discriminating non-isomorphic graph structures, the tight connection between WL and GNNs lets us study to what extend a GNN is actually able to encode graph structure information into its vectorial node representations from a theoretical point of view. As such, we will see that Graph Neural Networks can also be well-motivated as graph isomorphism approximators, making them powerful tools to reason about structural graph properties as well. While prior evaluation and analysis of GNNs has been largely empirical, our theoretical study in Morris *et al.* (2019, 2021a) (and in the concurrent work of Xu *et al.* (2019c)) on the expressive power of GNNs, *i.e.* their ability to distinguish non-isomorphic graphs, leads to major implications of our understanding of GNNs and gives, as well, rise to solutions for overcoming some of their shortcomings.

In the following, we study the expressive power of the specific GNN instantiation

$$\boldsymbol{h}_v^{(\ell)} = \sigma\Big(\boldsymbol{W}_1^{(\ell)}\boldsymbol{h}_v^{(\ell-1)} + \sum_{w \in \mathcal{N}(v)} \boldsymbol{W}_2^{(\ell)}\boldsymbol{h}_w^{(\ell-1)}\Big), \tag{3.30}$$

which utilizes a simple learnable skip-connection to discriminate between central and neighboring node features (Hamilton *et al.*, 2017). Input node features can be either given as $\boldsymbol{h}_v^{(0)}$ by the one-hot encoding of initial labels or node degrees. Notably, we restrict our theoretical analysis on discrete rather than continuous input features.

Our first theoretical result shows that a GNN architecture does not have more power in terms of distinguishing between non-isomorphic (sub-)graphs than the WL algorithm. More formally, for every feature encoding of input labels to vectors $\boldsymbol{h}_v^{(0)}$, and for every choice of parametrization via $\boldsymbol{W}_1^{(\ell)}$ and $\boldsymbol{W}_2^{(\ell)}$, we have that the coloring of $c_v^{(\ell)}$ of the WL always refines the coloring $\boldsymbol{h}_v^{(\ell)}$ induced by a GNN parametrized by $\boldsymbol{W}_1^{(\ell)}$, $\boldsymbol{W}_2^{(\ell)}$:

Theorem 1. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph. Then, for all $\ell \geq 1$, and for all weights $\boldsymbol{W}_1^{(\ell)}$ and $\boldsymbol{W}_2^{(\ell)}$, it holds that*

$$c_u^{(\ell)} = c_v^{(\ell)} \Rightarrow \boldsymbol{h}_u^{(\ell)} = \boldsymbol{h}_v^{(\ell)}$$

for all $u, v \in \mathcal{V}$, i.e. *a GNN is* at most as powerful as *the WL algorithm in distinguishing non-isomorphic graph structures.*

Proof. We proof by induction, *cf.* (Morris *et al.*, 2019). The induction hypothesis and $c_u^{(\ell)} = c_v^{(\ell)}$ imply that the multisets of neighboring node features $\{\!\{\boldsymbol{h}_w^{(\ell-1)} : w \in \mathcal{N}(u)\}\!\}$ and $\{\!\{\boldsymbol{h}_w^{(\ell-1)} : w \in \mathcal{N}(v)\}\!\}$ are identical as well, leading to $\boldsymbol{h}_u^{(\ell)} = \boldsymbol{h}_v^{(\ell)}$ by design. □

Importantly, this result holds for a broad class of GNN architectures and all possible choices of parameters for them, not just for the given instantiation of Equation (3.30). On the positive side, we can further show that given the right parameter initialization, the given GNN instantiation has the same expressive power as the WL algorithm, completing the equivalence:

Theorem 2. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph. Then, for all $\ell \geq 1$, there exists a sequence of weights $((\boldsymbol{W}_1^{(1)}, \boldsymbol{W}_2^{(1)}), \ldots (\boldsymbol{W}_1^{(\ell)}, \boldsymbol{W}_2^{(\ell)}))$ such that*

$$\boldsymbol{h}_u^{(\ell)} = \boldsymbol{h}_v^{(\ell)} \iff c_u^{(\ell)} = c_v^{(\ell)}$$

for all $u, v \in \mathcal{V}$, i.e. *a GNN* can be as powerful as *the WL algorithm in distinguishing non-isomorphic graph structures.*

Proof. The proof in Morris *et al.* (2019) starts by giving the proof for graphs where all nodes have the same initial color, and is then extended to the case of labeled graphs. It involves creating a weight matrix $\boldsymbol{W}_2^{(\cdot)} \in \mathbb{R}^{F \times |\mathcal{V}|}$ that guarantees an injective mapping from the multiset of neighboring node features $\{\!\{\boldsymbol{h}_w^{(\ell-1)} : w \in \mathcal{N}(v)\}\!\}$ to its aggregative representation $\boldsymbol{m}_{\mathcal{N}(v)}$. □

Xu *et al.* (2019c) further showed that *any* GNN can be as expressive as the WL algorithm as long as it is injective, *i.e.* it will never map two different multisets to the same underlying representation. In the light of the above results, GNNs can be viewed as an extension of the WL algorithm which in principle have the same power but are more flexible in their ability to adapt to the learning task at hand and are naturally able to handle continuous node features. Since the node colorings produced by the WL algorithm are essentially one-hot encodings, they cannot really capture the similarity between different subgraphs. In contrast, a GNN as expressive as the WL algorithm learns to embed nodes into a continuous space. This enables the GNN to not only discriminate different structures, but also to learn to map similar graph structures to similar embeddings and to capture dependencies between graph structures (Xu *et al.*, 2019c), improving generalization and strengthen robustness w.r.t. nosiy edges and node features (Yanardag & Vishwanathan, 2015).

3.4.2 Provably Powerful Graph Neural Networks

One disadvantage of the aforementioned provably powerful GNN instantiation given in Equation (3.30) is that its parameter matrix needs to scale linearly with the number of nodes $|\mathcal{V}|$ in order to guarantee sufficient expressive power. A different way of proving Theorem 2 is to rely on the *universal approximation theorem* of MLPs (Hornik, 1991; Hornik *et al.*, 1989) in order to show that $\sum_{w \in \mathcal{N}(v)} \mathrm{MLP}_{\boldsymbol{\theta}}(\boldsymbol{h}_w^{(\ell-1)})$ defines an injective aggregation on multisets, as described in Zaheer *et al.* (2017); Xu *et al.* (2019c) and in Section 3.2.1. As a result, Xu *et al.* (2019c) derive the *Graph Isomorphism Network* (*GIN*) layer

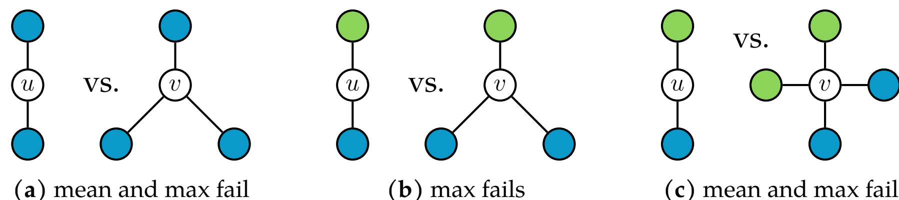
$$\boldsymbol{h}_v^{(\ell)} = \mathrm{MLP}_{\boldsymbol{\theta}_1}^{(\ell)}\Big(\big(1+\epsilon^{(\ell)}\big) \cdot \boldsymbol{h}_v^{(\ell-1)} + \sum_{w \in \mathcal{N}(v)} \mathrm{MLP}_{\boldsymbol{\theta}_2}\big(\boldsymbol{h}_w^{(\ell-1)}\big)\Big), \tag{3.31}$$

where $\epsilon^{(\ell)} \in \mathbb{R}$ denotes a trainable scalar that learns to distinguish central node information from neighboring node information. In case input features are given as one-hot encodings, the second $\mathrm{MLP}_{\boldsymbol{\theta}_2}$ can be dropped as their summation alone is injective and MLPs can represent the composition of functions, *i.e.* $\mathrm{MLP}_{\boldsymbol{\theta}_1} \circ \mathrm{MLP}_{\boldsymbol{\theta}_2}$.

One important takeaway is that the applied sum aggregation is necessary to ensure sufficient model expressivity, opposed to other well defined multiset functions such as mean or max-pooling aggregators (Xu *et al.*, 2019c). Such aggregators get confused by surprisingly simple graphs and are therefore less powerful than the WL algorithm, *cf.* Figure 3.9. Nonetheless, aggregative functions such as mean or max can still inject an effective bias into the learning model, *e.g.*, in cases where distributional information is more important than exact structure (mean aggregation) or where learning about distinct elements of a multiset is crucial (max aggregation) (Xu *et al.*, 2019c).

Inspired by this observation, Corso *et al.* (2020) propose to leverage multiple aggregative functions within a single GNN layer. In particular, they show that one needs at least m aggregators in order to discriminate between multisets of size m whose underlying set is continuous rather than discrete. As a result, their *Principal Neighborhood Aggregation* (*PNA*) network combines multiple aggregative functions with degree-scalars to better capture graph structural properties

$$\boldsymbol{h}_v^{(\ell)} = \boldsymbol{W}_2^{(\ell)} \boldsymbol{h}_v^{(\ell-1)} + \bigoplus_{w \in \mathcal{N}(v)} \boldsymbol{W}_1^{(\ell)} \boldsymbol{h}_w^{(\ell-1)}, \tag{3.32}$$



(**a**) mean and max fail (**b**) max fails (**c**) mean and max fail

Figure 3.9: Examples of insufficient expressive power regarding mean and max aggregation. Here, node colors represent different node features. Nodes u and v will aggregate the same representation even though their surrounding graph structures clearly differ. (**a**) Since all nodes have the same features, both mean and max aggregation cannot discriminate differently sized-neighborhoods. (**b**) Max aggregation collapses to the same representation although the number of same-colored neighbors is different. (**c**) In case identical features across neighborhoods are evenly distributed, the same holds true for mean aggregation. In all cases, sum aggregation can distinguish the different graph structures (Xu *et al.*, 2019c).

where

$$\bigoplus = \underbrace{\begin{bmatrix} 1 \\ s(\deg(v), 1) \\ s(\deg(v), -1) \end{bmatrix}}_{\text{Scalers}} \otimes \underbrace{\begin{bmatrix} \text{mean} \\ \text{max} \\ \text{min} \end{bmatrix}}_{\text{Aggregators}} \tag{3.33}$$

with $\bigotimes$ being the tensor product and

$$s(d, \alpha) = \left(\frac{\log(d+1)}{\frac{1}{|\mathcal{V}|} \sum_{v \in \mathcal{V}} \log(\deg(v) + 1)} \right)^{\alpha} \tag{3.34}$$

denoting scalers based on node degree $\deg(v)$. Here, degree-scalers are used to perform either a degree-dependent amplification or an attenuation of incoming messages. With this, sum aggregation can be well expressed as the composition of mean aggregation and a degree amplifying scaler. The choice of logarithmic amplification has the same expressive power, but may lead to better generalization since small changes in node degrees will not cause aggregations to be amplified exponentially w.r.t. model depth.

3.4.2.1 Higher-Order Graph Neural Networks. Since the power of the WL algorithm has been completely characterized (Arvind *et al.*, 2015; Kiefer *et al.*, 2015), GNNs obey the same shortcomings. For example, both methods will give the same color to every node in two regular graphs of same node degree, although there exists regular graphs that are clearly non-isomorphic, *cf.* Figure 3.8b. Moreover, they are not capable of capturing simple graph theoretic properties, *e.g.*, triangle counts, which are an important measure in social network analysis (Milo *et al.*, 2002; Newman, 2003). As a result, a natural question to ask is whether we can design provably more powerful GNNs that are able to overcome such shortcomings induced by the WL algorithm.

Here, the k-WL algorithm proposed by László Babai (Cai *et al.*, 1992) generalizes the WL algorithm to higher-order sub-structures. To make the WL algorithm more powerful, it colors k-sized subsets of nodes from $\mathcal{V}$ of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $k \geq 2$, instead of

nodes.[5] By defining a neighborhood between these subsets, we can leverage an update rule that is similar to the one utilized in the WL algorithm. Formally, the k-WL algorithm computes a coloring $c_{\boldsymbol{v}}^{(\ell)} \in \Sigma$ of a k-sized subset of nodes $\boldsymbol{v} = \{v_1, \ldots, v_k\} \subseteq \mathcal{V}$ where the neighborhood of $\boldsymbol{v}$ is defined as

$$\mathcal{N}(\boldsymbol{v}) = \{\boldsymbol{w} : \boldsymbol{w} \subseteq \mathcal{V}, |\boldsymbol{w}| = k, |\boldsymbol{v}| \cap |\boldsymbol{w}| = k - 1\}. \tag{3.35}$$

That is, the neighborhood set $\mathcal{N}(\boldsymbol{v})$ of $\boldsymbol{v}$ is described by all other k-sized subsets of $\mathcal{V}$ that only differ in a single node. In iteration 0, the algorithm labels each k-sized subset of $\mathcal{V}$ by its *atomic type*, *i.e.* two subsets $\boldsymbol{v}, \boldsymbol{w} \subseteq \mathcal{V}$ get the same color if the map $\boldsymbol{v} \mapsto \boldsymbol{w}$ induces an isomorphism between their induced subgraphs $\mathcal{G}[\boldsymbol{v}]$ and $\mathcal{G}[\boldsymbol{w}]$, respectively. For iteration $\ell \geq 1$, we define

$$c_{\boldsymbol{v}}^{(\ell)} = \textsc{Hash}\left(c_{\boldsymbol{v}}^{(\ell-1)}, \{\!\{ c_{\boldsymbol{w}}^{(\ell-1)} : \boldsymbol{w} \in \mathcal{N}(\boldsymbol{v}) \}\!\}\right). \tag{3.36}$$

Hence, two subsets $\boldsymbol{v}$ and $\boldsymbol{w}$ with $c_{\boldsymbol{v}}^{(\ell-1)} = c_{\boldsymbol{w}}^{(\ell-1)}$ will receive different colors in iteration ℓ only if there exist a neighboring subset with a different coloring. The k-WL is probably more powerful than the originally proposed WL algorithm. For example, for $k = 3$, the k-WL can easily reason about triangle counts in the underlying graph structure. Furthermore, by increasing k, the algorithm gets more powerful in terms of distinguishing non-isomorphic graphs. In particular, for each $k \geq 2$, there exists non-isomorphic graphs which can be distinguished by the $(k+1)$-WL but not by the k-WL (Cai *et al.*, 1992).

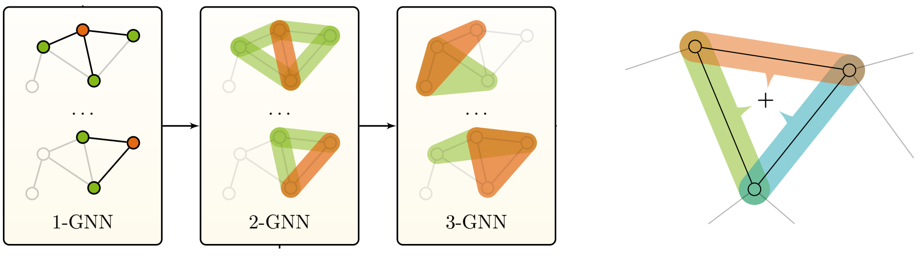
While powerful, the k-WL has the disadvantage of being inherently *global*, *i.e.* it labels a k-sized subset of nodes by looking at *all* other k-sized subsets that only differ in a certain node, and therefore does not take the sparsity of the underlying graph into account. In order to capture the local properties of the graph better, we propose to leverage a *local* k-WL variant, in which the neighborhood set $\mathcal{N}(\boldsymbol{v})$ of a k-sized subset $\boldsymbol{v} \subseteq \mathcal{V}$ is split into a *local neighborhood* $\mathcal{N}^{(L)}(\boldsymbol{v})$ and a *global neighborhood* $\mathcal{N}^{(G)}(\boldsymbol{v}) = \mathcal{N}(\boldsymbol{v}) \setminus \mathcal{N}^{(L)}(\boldsymbol{v})$, where the local neighborhood $\mathcal{N}^{(L)}(\boldsymbol{v})$ consists of all $\boldsymbol{w} \in \mathcal{N}(\boldsymbol{v})$ such that $(v, w) \in \mathcal{E}$ for the unique $v \in \boldsymbol{v} \setminus \boldsymbol{w}$ and the unique $w \in \boldsymbol{w} \setminus \boldsymbol{v}$.

Given the aforementioned k-WL formulation, we can design a k-GNN operator as in Equation (3.30) that exchanges messages between k-sized node subsets rather than between single nodes, *i.e.*

$$\boldsymbol{h}_{\boldsymbol{v}}^{(\ell)} = \sigma\Big(\boldsymbol{W}_1^{(\ell)} \boldsymbol{h}_{\boldsymbol{v}}^{(\ell-1)} + \sum_{\boldsymbol{w} \in \mathcal{N}^{(L)}(\boldsymbol{v}) \cup \mathcal{N}^{(G)}(\boldsymbol{v})} \boldsymbol{W}_2^{(\ell)} \boldsymbol{h}_{\boldsymbol{w}}^{(\ell-1)}\Big). \tag{3.37}$$

Similar to the k-WL, we can initialize $\boldsymbol{h}_{\boldsymbol{v}}^{(0)}$ to denote the atomic type of $\boldsymbol{v} \subseteq \mathcal{V}$ as a one-hot encoding. Moreover, one could split the sum of Equation (3.37) into two sums ranging over $\mathcal{N}^{(L)}(\boldsymbol{v})$ and $\mathcal{N}^{(G)}(\boldsymbol{v})$, using distinct parameters that enable the model to learn the importance of local and global neighborhoods, respectively. Alternatively, we can restrict Equation (3.37) to solely operate on the local neighborhood $\mathcal{N}^{(L)}(\boldsymbol{v})$ in order to scale k-GNNs to larger datasets and to prevent over-fitting. The running time of k-GNN depends on $|\mathcal{V}|$, k, and the sparsity of the graph, where each iteration is bounded by the number of subsets of size k times the maximum degree. In order to

[5] Following Morris *et al.* (2017), we define a variant of the k-WL operating on k-sets instead of k-tuples due to scalability and ease of notation.



(**a**) Hierarchical 1-2-3-GNN architecture (**b**) Pooling from 2- to 3-GNN

Figure 3.10: Illustration of the hierarchical variant of the k-GNN architecture. For each k-sized node subset $\boldsymbol{v} \subseteq \mathcal{V}$, a representation $\boldsymbol{h}_{\boldsymbol{v}}^{(L)}$ is learned that is initialized with the learned representations of all $(k-1)$-sized subsets $\boldsymbol{w}$ of $\boldsymbol{v}$, $\boldsymbol{w} \subseteq \boldsymbol{v}$, $|\boldsymbol{v}| - |\boldsymbol{w}| = 1$ (Morris *et al.*, 2019).

scale our method to larger datasets, we can also utilize scalability methods of GNNs, which we introduce in detail in Chapter 5.

Importantly, Theorem 2 also applies to the relation of k-WL and k-GNN, *i.e.* there exists parameter configurations for k-GNN that are as powerful as the set-based k-WL variant in distinguishing non-isomorphic graph structures. This result trivially lifts to the k-dimensional case due to the equivalence of Equation (3.29) and Equation (3.36), which we prove in detail in Morris *et al.* (2019).

One key benefit of the end-to-end trainable k-GNN framework compared to the discrete k-WL algorithm is that we can hierarchically combine representations learned at different granularities, *cf.* Figure 3.10a. Concretely, rather than simply using one-hot encoded feature inputs in a k-GNN, we propose to utilize a *hierarchical* variant of k-GNN that uses the features learned by a $(k-1)$-dimensional GNN (in addition the its atomic type), *i.e.*

$$\boldsymbol{h}_{\boldsymbol{v}}^{(0)} \leftarrow \sigma\Big(\big[\boldsymbol{h}_{\boldsymbol{v}}^{(0)}, \sum_{\substack{\boldsymbol{w} \subseteq \boldsymbol{v} \\ |\boldsymbol{v}|-|\boldsymbol{w}|=1}} \boldsymbol{h}_{\boldsymbol{w}}^{(L)}\big] \cdot \boldsymbol{W}\Big) \tag{3.38}$$

where $\boldsymbol{w}$ and $\boldsymbol{v}$ represent $(k-1)$-sized and k-sized node subsets of $\mathcal{V}$, respectively, *cf.* Figure 3.10b. Here, all final representations $\boldsymbol{h}_{\boldsymbol{w}}^{(L)}$, $\boldsymbol{w} \subseteq \boldsymbol{v}$, obtained by a L-layer $(k-1)$-GNN are *pooled* into an initial representation for $\boldsymbol{h}_{\boldsymbol{v}}^{(0)}$ for further processing. We further propose to *learn* such pooling mechanism via the trainable parameter matrix $\boldsymbol{W}$ of appropriate size. Hence, the features are recursively learned from dimensions 1 to k in an end-to-end fashion. This hierarchical model also satisfies Theorem 2, so its representational capacity is theoretically equivalent to a standard k-GNN (in terms of its relationship to the k-WL) (Morris *et al.*, 2019). Nonetheless, hierarchy is a natural inductive bias for graph modeling, since many real-world graphs incorporate hierarchical structure, so we expect this hierarchical formulation to offer empirical utility.

3.4.2.2 Alternative Provably More Powerful GNN Variants. The relationship between Graph Neural Networks and the WL algorithm has attracted a lot of research regarding the design of provably more powerful GNN variants over the past few years. For example, Maron *et al.* (2019b); Morris *et al.* (2020b) propose alternative neural

architectures with the same power as the k-WL. Murphy *et al.* (2019b); Vignac *et al.* (2020) propose to utilize node identifiers as additional initial node features to maintain information about which node has contributed which information to the aggregated information, leading to provably more powerful architectures. To ensure invariance w.r.t. permutation, aggregation is either performed across all possible permutations $\boldsymbol{\pi} \in S_{|\mathcal{V}|}$ (Murphy *et al.*, 2019a,b) (and made tractable via canonical orientations or sampling), or by passing node feature matrices instead of node feature vectors as messages (Vignac *et al.*, 2020). A similar idea was utilized in Sato *et al.* (2020), which includes the addition of random node features instead of unique node identifiers to the message passing phase. The latter was refined by Dasoulas *et al.* (2020); Abboud *et al.* (2020), which investigate the connection between random coloring and universality. Furthermore, Bouritsas *et al.* (2020) calculate higher-order topology features in a pre-processing stage, and You *et al.* (2019); Li *et al.* (2020) incorporate distance information encodings to the message passing phase. Finally, Zeng *et al.* (2020a) propose to utilize deep L-layer GNNs on shallow k-hop subgraphs around each node ($L > k$) in combination with subgraph pooling, leading to provably more power in capturing graph structural properties compared to the WL.

3.4.3 Trading Expressivity and Generalization

The relation of GNNs to the WL expressivity gave rise to the development of various provably powerful GNN architectures, fostering our understanding on theoretical graph-based model capacity. However, it is important to acknowledge that although model expressivity is an important property, it might not be the deciding factor for graph machine learning after all. In the end, we do not need to solve the graph isomorphism problem, but are instead interested in representations that are able to capture structural and feature-based local information in a meaningful and data-driven way. Strong discriminative power might even backfire due to over-parametrization and over-fitting, and provably more powerful GNN architectures are known to scale less-well to bigger datasets and deeper models (Dwivedi *et al.*, 2020).

To confirm this intuition, we have computed the *completeness ratio*, *i.e.* the fraction of graphs that can be perfectly distinguished by the WL algorithm from all other graphs (Morris *et al.*, 2021a), across a wide-spread set of graph classification benchmark datasets (Morris *et al.*, 2020a; Hu *et al.*, 2020a), *cf.* Figure 3.11. The results reveal that the WL is sufficiently expressive to distinguish most non-isomorphic graphs in Morris *et al.* (2021a) after only two iterations. This effect even holds true on our proposed large-scale OGB datasets (Hu *et al.*, 2020a), *cf.* Table 3.1, which we introduce in Section 5.4. Thus, although devising provably powerful graph learning architectures is a meaningful theoretical endeavor, the key to improving real-world tasks lies in improving GNN's generalization abilities, for which so far only a few notable contributions have been made (Kriege *et al.*, 2018; Garg *et al.*, 2020; Xu *et al.*, 2021). Alternatively, the generalization capabilities of a GNN can be improved by developing more task-specific architectures that directly inject further domain-knowledge into its design, which we do in Chapter 4.

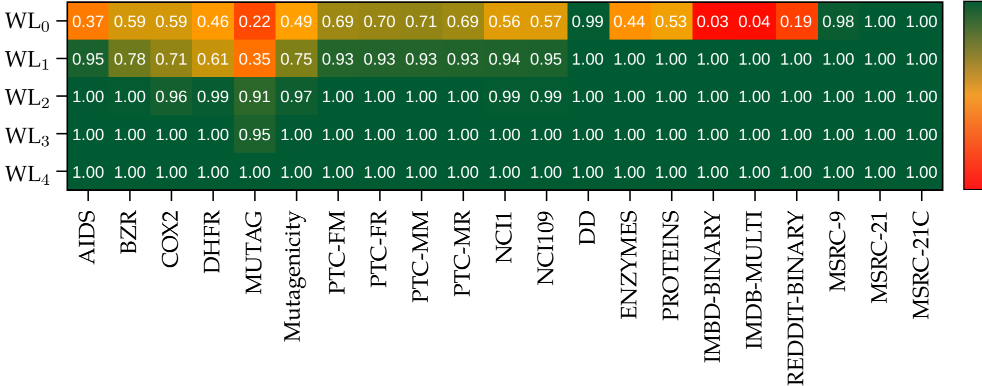


Figure 3.11: The ability of the WL algorithm to distinguish non-isomorphic graphs (Morris *et al.*, 2021a) over the benchmark datasets in Morris *et al.* (2020a). WL$_i$ refers to running the WL algorithm for exactly i iterations. It can be seen that the WL is already sufficiently expressive to distinguish *most* non-isomorphic graphs after only two iterations, and is able to distinguish *all* non-isomorphic graphs after exactly four iterations.

Dataset	Ratio of WL distinguishable non-isomorphic graph pairs
`ogbg-moltox21`	> 0.99
`ogbg-moltoxcast`	> 0.99
`ogbg-molfreesolv`	1.00
`ogbg-molesol`	1.00
`ogbg-mollipo`	1.00
`ogbg-molhiv`	> 0.99

Table 3.1: The ratio of distinguished non-isomorphic graph pairs by the WL algorithm (Morris *et al.*, 2021a) over the datasets in Hu *et al.* (2020a).

3.5 Evaluation

In this section, we showcase the application and power of GNNs across a wide range of diverse tasks and domains. In detail, we will look into how GNNs enable state-of-the-art results on tasks such as graph classification and regression (Section 3.5.1), node classification (Section 3.5.2), superpixel image classification (Section 3.5.3) and geometric learning (Section 3.5.4). We will, in particular, see how our own proposed solutions, namely SplineCNN (*cf.* Section 3.3) and k-GNN (*cf.* Section 3.4) help to improve performance in these diverse learning tasks.

3.5.1 Graph Classification and Regression

We perform an extensive set of experiments to evaluate the performance of GNNs and its higher-order k-GNN variants on the task of graph classification and regression.

3.5.1.1 Datasets. To compare GNN architectures to graph kernel approaches, we use well-established small-scale benchmark datasets from the kernel literature (Morris *et al.*, 2020a). In particular, we make use of bioinformatic datasets (PROTEINS, PTC-FM, MUTAG) and social network datasets (IMDB-BINARY, IMDB-MULTI). IMDB-BINARY and IMDB-MULTI are movie collaboration datasets, where each graph corresponds to an ego-network of actors/actresses. An edge is drawn between two actors/actresses if they appear in the same movie, and the task is to infer the genre of the given movie. PROTEINS contains graphs representing proteins according to the graph model of Borgwardt *et al.* (2005). Nodes represent secondary structure elements (SSEs) which are connected whenever there are neighbors either in the amino acid sequence or in 3D space. The task is to perform protein function prediction. PTC-FM is a dataset from the Predicte Toxicology Challenge (PTC) (Helma *et al.*, 2001), containing chemical compounds labeled according to carcinogenicity on female mice (FM). MUTAG is a dataset consisting of mutagenetic aromatic and heteroaromatic nitro compounds, divided into two classes according to their mutagenic effect on a bacterium (Debnath *et al.*, 1991; Kriege & Mutzel, 2012). The nodes of each graph in these dataset are annotated with (discrete) labels or no labels. For datasets without pre-defined node features, we use one-hot encodings of node degrees as inputs.

To demonstrate that GNNs, especially k-GNNs, are able scale to larger datasets and offer benefits on real-world applications, we further conduct experiments on the challenging and publicly available molecular QM9 dataset (Ramakrishnan *et al.*, 2014). Molecules in the dataset consist of Hydrogen (H), Carbon (C), Oxygen (O), Nitrogen (N), and Flourine (F) atoms and contain up to 9 heavy (non-Hydrogen) atoms. This results in about 134k drug-like organic molecules that span a wide range of chemistry. Additionally, a wide range of interesting and fundamental chemical properties are pre-computed via expensive density functional theory. The aim here is to perform regression on twelve different targets, representing energetic (U_0, U, H, G, C_{v}), electronic ($\varepsilon_{\mathrm{HOMO}}$, $\varepsilon_{\mathrm{LUMO}}$, $\Delta\varepsilon$), geometric ($\langle R^2 \rangle$, μ, α), and thermodynamic (ZPVE) properties. Sufficiently successful models on this dataset could help to automate challenging chemical search problems in drug discovery (Gilmer *et al.*, 2017). For this, we represent molecules as graphs, in which nodes represent atoms and edges represent chemical bonds. This raw graph-based representation naturally avoids the need for designing hand-crafted molecular fingerprints.

3.5.1.2 Experimental Setup. We evaluate the performance of a simple GNN (Hamilton *et al.*, 2017) (1-GNN) as defined in Equation (3.30), and compare against its higher-order model, namely 1-2-3-GNN, *cf.* Section 3.4.2. On the QM9 dataset, we conduct further studies on other higher-order variants, *i.e.* 1-2-GNN and 1-3-GNN. We always use three layers for 1-GNN, and two layers for 2-GNN and 3-GNN, all with a hidden feature dimensionality of 64. For the final classification and regression steps, we use a three-layer MLP with binary cross entropy and mean squared error as loss criterion, respectively. For classification, we use a dropout layer after the first layer of the MLP with dropout probability $p = 0.5$ (Srivastava *et al.*, 2014). We apply global mean pooling to generate a vector representation of the graph from the computed node features. For the higher-order variants, we utilize global pooling for each k, and concatenate the resulting graph-wise vectors before feeding them into the MLP. Moreover, we use the Adam optimizer (Kingma & Ba, 2015) with an initial learning rate of 10^{-2} and leverage an adaptive learning rate decay based on validation results to a minimum of 10^{-5}. We train the classification networks for 100 epochs and the regression networks for 200

Method	**Dataset**				
	PROTEINS	IMDB-BINARY	IMDB-MULTI	PTC-FM	MUTAG
#graphs	1,113	1,000	1,500	349	188
Avg. #Nodes	39.06	19.77	13.00	14.11	17.93
Avg. #Edges	72.82	96.53	65.94	14.48	19.79
1-WL	73.8	72.5	**51.5**	62.9	78.3
2-WL	75.2	72.6	50.6	**64.7**	77.0
3-WL	74.7	73.5	49.7	61.5	83.2
WL-OA	75.3	73.1	50.4	62.7	84.5
1-GNN	72.2	71.2	47.7	59.3	82.2
1-2-3-GNN	**75.5**	**74.2**	49.5	62.8	**86.1**

Table 3.2: Mean classification accuracies [%] using 10-fold cross validation on various graph benchmark datasets from Morris *et al.* (2020a).

epochs. Notably, we train a separate model for predicting each of the twelve chemical properties of the QM9 dataset.

For the graph classification experiments, we additionally compare against related graph kernel approaches. In detail, we utilize the Weisfeiler-Lehman subtree kernel (1-WL) (Shervashidze *et al.*, 2011), the global-local k-WL kernel ($k \in \{2, 3\}$) (Morris *et al.*, 2017), and the Weisfeiler-Lehman Optimal Assignment kernel (WL-OA) (Kriege *et al.*, 2016). For each kernel, we first compute the normalized Gram matrix and obtain classification accuracies using the C-SVM implementation of LibSVM (Chang & Lin, 2011) afterwards. We select the C parameter from $\{10^{-3}, 10^{-2}, \ldots, 10^{2}, 10^{3}\}$ and the number of WL iterations from $\{0, 1, \ldots, 5\}$.

For the smaller graph classification datasets (Morris *et al.*, 2020a), which we use for comparison against the graph kernel methods, we perform 10-fold cross validation and randomly sample 10% of each training fold to act as a validation set. For the QM9 dataset, we follow the dataset splits decribed in (Wu *et al.*, 2018): We randomly sample 10% of the examples for validation, another 10% for testing, and use the remaining examples for training. We use the same initial node features as described in Gilmer *et al.* (2017). The code for reproducing all results utilizes the PyTorch Geometric library (Fey & Lenssen, 2019) and is available on GitHub.[6]

3.5.1.3 Results and Discussion. Table 3.2 shows the results for comparison with the graph kernel methods on the graph classification benchmark datasets. Interestingly, our hierarchical k-GNN model performs on par or slightly better than the kernel methods despite the small dataset sizes. We also find that the 1-2-3-GNN significantly outperforms the standard GNN model on all five datasets, with the GNN being the overall weakest method across all tasks. We contribute this to the simplicity of the utilized GNN, as recent works have shown superior results over kernels when using more advanced pooling techniques (Ying *et al.*, 2018b). We take a closer look into advanced pooling techniques in Section 4.3. Here, we opted to use a standard global pooling approach in order to compare simple GNN and k-GNN implementations with standard off-the-shelf kernels.

[6] Code for k-GNN: `https://github.com/chrsmrrs/k-gnn` (last access: August 25, 2022)

Target	Method				Gain
	1-GNN	1-2-GNN	1-3-GNN	1-2-3-GNN	
μ	0.493	0.493	**0.473**	0.476	4.0%
α	0.78	**0.27**	0.46	**0.27**	65.3%
$\varepsilon_{\text{HOMO}}$	**0.00321**	0.00331	0.00328	0.00337	—
$\varepsilon_{\text{LUMO}}$	0.00355	**0.00350**	0.00354	0.00351	1.4%
$\Delta\varepsilon$	0.0049	0.0047	**0.0046**	0.0048	6.1%
$\langle R^2 \rangle$	34.1	**21.5**	25.8	22.9	37.0%
ZPVE	0.00124	**0.00018**	0.00064	0.00019	85.5%
U_0	2.32	**0.0357**	0.6855	0.0427	98.5%
U	2.08	**0.107**	0.686	0.111	94.9%
H	2.23	0.070	0.794	**0.0419**	98.1%
G	1.94	0.140	0.587	**0.0469**	97.6%
C_{v}	0.27	0.0989	0.158	**0.0944**	65.0%

Table 3.3: Mean absolute errors on the QM9 dataset (Ramakrishnan *et al.*, 2014) of k-GNN variants (Morris *et al.*, 2019). The far-right column shows the improvement of the best k-GNN model in comparison to the 1-GNN baseline.

Table 3.3 shows the results for the QM9 dataset. On eleven out of twelve targets, all our hierarchical variants beat the classical node-wise GNN model. For example, on the target H, we achieve a 98.1% improvement in mean absolute error compared to 1-GNN. However, the additional structural information extracted by the k-GNN layers does not serve all tasks equally, leading to huge differences in gains across the targets. It should be further noted that our k-GNN models have more parameters than the 1-GNN model, since we stack two additional GNN layers for each k. However, extending the 1-GNN model by additional layers to match the number of parameters of the k-GNN did not lead to better results in any experiment.

3.5.2 Node Classification

Next, we address the problem of (semi-supervised) node classification, and see how GNNs and SplineCNNs in particular enable state-of-the-art performance.

3.5.2.1 Datasets. We utilize the three citation graph datasets Cora, CiteSeer and PubMed (Sen *et al.*, 2008; Yang *et al.*, 2016). Here, nodes are scientific publications and edges represent citation links between these documents. The datasets contain sparse bag-of-words feature vectors for each document (Kipf & Welling, 2017). Citation links are treated as undirected edges from which we obain a binary, symmetric adjacency matrix. The task is to predict the categories of documents based on known categories from a subet of nodes $\mathcal{V}_{\text{train}} \subseteq \mathcal{V}$.

3.5.2.2 Experimental Setup. Following the experimental setup of Levie *et al.* (2017), we randomly split the dataset into 500 nodes used for validation and testing each, and utilize the remaining nodes as training nodes. We repeat experiments for 100 times with different parameter initialization and varying random splits. We evaluate the performance of three different GNNs models: GCN (Kipf & Welling, 2017), GAT

Method	Dataset		
	Cora	CiteSeer	PubMed
#Nodes	2,708	3,327	19,717
#Edges	5,278	4,552	44,324
MLP	74.92±1.19	73.39±1.69	87.06±1.38
Label Propagation	83.72±1.49	69.80±1.95	83.76±1.71
GCN	89.15±1.17	77.37±1.45	87.73±1.24
GAT	89.11±1.19	77.28±1.79	88.13±1.29
SplineCNN	**89.51**±1.21	**78.34**±1.60	**88.58**±1.39

Table 3.4: Mean classification accuracies with standard deviation [%] on the three citation graphs from Sen *et al.* (2008); Yang *et al.* (2016), computed across 100 different runs.

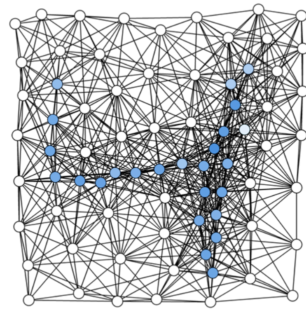
(Veličković *et al.*, 2018) and SplineCNN (Fey *et al.*, 2018) as described in Section 3.2.4 and Section 3.3. We utilize two-layer GNNs with a hidden feature dimensionality of 16 (64 across 8 heads for GAT), and use ReLU as our non-linear activation function (Glorot *et al.*, 2011). While GCN and GAT will transform neighboring node features via the *same* weight matrix, SplineCNN learns an anisotropic aggregation based on pre-defined pseudo-coordinate representations. For pseudo-coordinates, we choose the globally normalized degree of source nodes, *i.e.* $e_{w,v} = \deg(w)/\max_{v \in \mathcal{V}} \deg(v)$, mapped into the fixed interval $[-1, 1]$, which allows the network to discrimate between highly cited works and less cited works to determine the category of a document more accurately. We use a B-spline basis degree of $m = 1$ and a kernel size of 2 to train SplineCNN.

We further compare against a *graph-agnostic* MLP as well as a (non-trainable) Label Propagation (Zhu *et al.*, 2003) baseline. The MLP will only make use of feature information for prediction, while the Label Propagation algorithm solely relies on graph-structure and ground-truth label information. In contrast, GNNs are able to incorporate both feature and structural information into their final prediction.

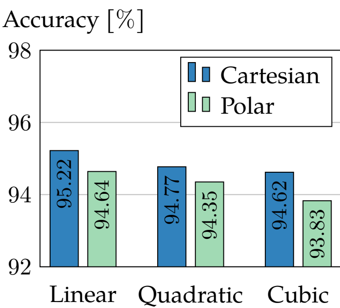
Training was done using the Adam optimization method (Kingma & Ba, 2015) for 200 epochs with learning rate of 10^{-2}, dropout probability $p = 0.5$ (Srivastava *et al.*, 2014) and L_2 regularization of $5 \cdot 10^{-3}$. As loss function, the cross entropy between the network's softmax output and a one-hot target distribution was used. The code for reproducing all results is directly incorporated into the PyTorch Geometric library (Fey & Lenssen, 2019).[7]

3.5.2.3 Results and Discussion. Results of our and related methods are given in Table 3.4, which reports the mean classification accuracy with standard deviation over 100 different runs across all three citation networks. Notably, all GNNs variants outperform the MLP and Label Propagation baselines by a significant margin, highlighting the benefits in fusing both feature and structural information into a unified model. Furthermore, it can be seen that SplineCNN consistently improves the state-of-the-art over all datasets. We contribute this improvement to the filtering based on pseudo-

[7] `https://github.com/rusty1s/pytorch_geometric` (last access: August 25, 2022)



(a) Superpixels graph example



(b) Classification accuracy

Figure 3.12: MNIST superpixels (a) example and (b) classification accuracy of SplineCNN (Fey *et al.*, 2018), using different pseudo-coordinate variants (Cartesian, polar) and B-spline basis degrees (linear, quadratic, cubic).

coordinates, which contains node degrees as additional information to learn more complex kernel functions. This indicates that SplineCNNs can be successfully applied to irregular but non-geometric data as well by being able to improve previous results in this domain.

3.5.3 Superpixels Graph Classification

In image recognition, input data fed to models tend to be very high-dimensional, requiring a lot of labelled data (Knyazev *et al.*, 2019a). Here, we consider an alternative approach that reduces the input dimensionality by converting images into superpixels graph representations (Radhakrishna *et al.*, 2012), which group redundant pixel-wise information into clusters that represent nodes. Despite a certain fine-grained loss of information, framing image recognition as a graph-based image classification problem has the advantage to reduce the model size by a significant amount of parameters, as far fewer GNN iterations are necessary to capture long-range information.

3.5.3.1 Dataset. Here, we make use of the MNIST dataset (LeCun *et al.*, 1998) which consists of 70k greyscale 28×28-sized images of handwritten digits (60k for training and 10k for testing). Following Monti *et al.* (2017), we represent each image as an embedded graph of 75 nodes that define the centroids of superpixels, *cf.* Figure 3.12a for an example. This amounts to a reduction in input dimensionality of around 90%. Superpixels are obtained using the SLIC algorithm (Radhakrishna *et al.*, 2012), after which we obtain the edge connectivity via k-nearest neighbor search. Notably, each graph utilizes *different* node positions and connectivities, so that traditional CNNs cannot be applied anymore.

3.5.3.2 Experimental Setup. We utilize SplineCNN (Section 3.3) for learning on the given two-dimensional embedded graphs, and evaluate all configurations of B-spline basis degree m (linear, quadratic, cubic) and possible pseudo-coordinate representations (Cartesian, polar). We make use of a two-layer SplineCNN architecture with intermediate hidden feature dimensionality of 32 and 64, after which we perform a

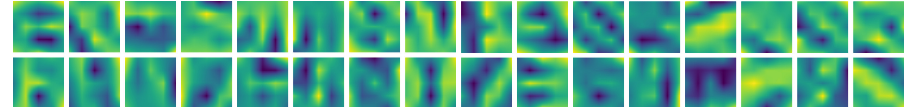


Figure 3.13: Visualization of the 32 kernels from the first SplineCNN layer (Fey *et al.*, 2018) with kernel size 5×5 and B-spline basis degree $m = 1$, trained on the MNIST superpixels dataset (Monti *et al.*, 2017).

global mean pooling and a two-layer MLP. We use the *Exponential Linear Unit* (*ELU*) as non-linearity after each spline-based convolutional layer. For Cartesian coordinates, we choose the kernel size to be $(4 + m) \times (4 + m)$ and for polar coordinates $(1 + m) \times 8$. Notably, our SplineCNN architecture use intermediate pooling operators based on the GRACLUS method (Dhillon *et al.*, 2007) after each convolution, as proposed by Defferrard *et al.* (2016). The pooling operation is able to obtain a coarsened graph by deriving a clustering on the graph nodes, aggregating nodes in one cluster and computing new pseudo-coordinates and new edge-connectivity for each of the new nodes, *cf.* Section 4.3. Training was done for 20 epochs with a batch size of 64, initial learning rate 10^{-2} and dropout probability $p = 0.5$ (Srivastava *et al.*, 2014). All networks were trained for 30 epochs using the ADAM optimizer (Kingma & Ba, 2015).

3.5.3.3 Results and Discussion. All results of the MNIST superpixels experiment are shown in Figure 3.12b. The best model is able to reach a final accuracy of 95.22%, which improves upon previous results from MoNet by 4.11 percentage points (Monti *et al.*, 2017). While we are using a similar architecture and the same input data as MoNet, the better results are an indication that our operator is able to capture more relevant information in the structure of the input. This can be explained by the fact that, in contrast to the MoNet kernels, our SplineCNN kernel function leverages individual trainable weights for each combination of input and output feature maps, just like the filters in traditional CNNs. In contrast, MoNet leverages pseudo-coordinates to compute attention coefficients to guide aggregation. Furthermore, we only notice small differences in accuracy for varying B-spline basis degree m and pseudo-coordinate representations. However, lower m and using Cartesian coordinates performs slightly better than the other configurations.

In addition, we visualize the 32 learned input kernels of the first spline-based convolutional layer, *cf.* Figure 3.13. It can be observed that edge detecting patterns are still visible despite being training on irregularly structured and compressed data. However, it is worth noting that there is still room for improvement, as a traditional CNN has no problem with reaching 99% accuracy on the image-based MNIST dataset.

3.5.4 Shape Correspondence

As our last and largest experiment, we validate SplineCNN on a collection of three-dimensional meshes solving the task of shape correspondence similar to Masci *et al.* (2015); Boscaini *et al.* (2016); Monti *et al.* (2017); Litany *et al.* (2017). Shape correspondence refers to the task of labeling each node of a given shape to the corresponding node of a reference shape (Masci *et al.*, 2015).

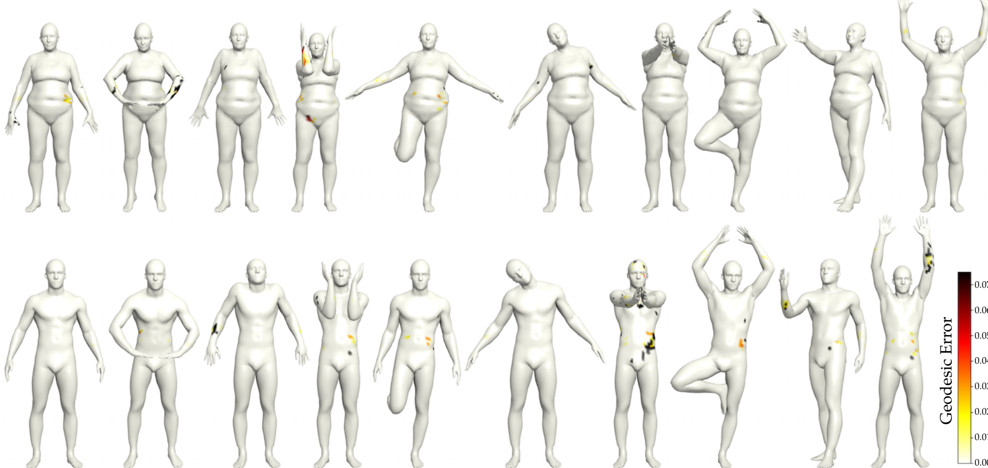
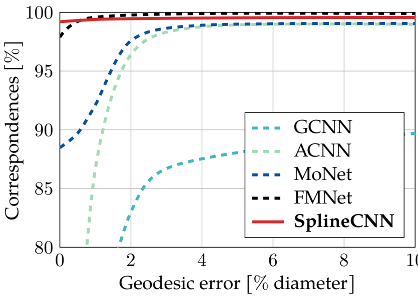


Figure 3.14: Node-wise geodesic errors of SplineCNN predictions (Fey *et al.*, 2018) on all 20 meshes of the FAUST test dataset (Bogo *et al.*, 2014).

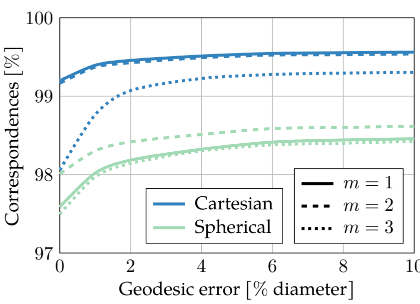
3.5.4.1 Dataset. We use the FAUST dataset (Bogo *et al.*, 2014), containing 10 scanned human shapes in 10 different poses, resulting in a total of 100 non-watertight meshes with 6,890 nodes each, *cf.* Figure 3.14 for mesh examples. The first 80 subjects in FAUST were used for training and the remaining 20 subjects for testing, following the datasets splits introduced in Monti *et al.* (2017). Ground-truth correspondence of FAUST meshes are given implicitly, since nodes are sorted in the exact same order for every example. Correspondence quality is then measured according to the Princeton benchmark protocol (Kim *et al.*, 2011), counting the percentage of derived correspondences that lie within a geodesic radius r around the correct node.

3.5.4.2 Experimental Setup. We apply a SplineCNN architecture with 6 convolutional layers, using a hidden feature dimensionality of 64 for each. As non-linear activation function, ELU is used after each layer. We further cast the shape correspondence experiment to a node classification task, in which a final linear layer maps to exactly 6,890 different outputs, denoting the probability that a node corresponds to a specific node in the reference shape. However, other loss criterion formulations are applicable as well, *cf.* Section 4.4. For Cartesian coordinates, we choose the kernel size to be $(4+m) \times (4+m) \times (4+m)$ and for spherical coordinates $(4+m) \times 8 \times (4+m)$. We evaluate our method on multiple choices of B-spline basis degree $m \in \{1, 2, 3\}$. Training was done for 100 epochs with a batch size of 1, initial learning rate 10^{-2} and dropout probability $p = 0.5$ (Srivastava *et al.*, 2014), using the Adam optimizer (Kingma & Ba, 2015) and cross entropy loss.

In contrast to related approaches (Masci *et al.*, 2015; Boscaini *et al.*, 2016; Monti *et al.*, 2017; Litany *et al.*, 2017), we go without hand-crafted feature descriptors as inputs, *e.g.*, SHOT descriptors (Tombari *et al.*, 2010), and force the network to learn from the geometry, *i.e.* the spatial relations, itself. Therefore, input features are trivially given by $\mathbf{1} \in \mathbb{R}^{|\mathcal{V}| \times 1}$. These simplifications reduce the computation time and memory consumption that are required to pre-process the data by a wide margin, making training and inference completely end-to-end and very efficient.



(**a**) SplineCNN and other approaches



(**b**) Different SplineCNN configurations

Figure 3.15: Geodesic error plots of the shape correspondence experiments with (a) SplineCNN and related approaches and (b) different SplineCNN configurations (Fey *et al.*, 2018)**.** The x-axis displays the geodesic distance in % of diameter and the y-axis displays the percentage of correspondences that lie within a given geodesic radius around the correct node. Our SplineCNN achieves high accuracy for low geodesic error compared to FMNet (Litany *et al.*, 2017) and significantly outperforms other approaches like MoNet (Monti *et al.*, 2017), GCNN (Masci *et al.*, 2015) and ACNN (Boscaini *et al.*, 2016).

3.5.4.3 Results and Discussion. Obtained accuracies for different geodesic errors are plotted in Figure 3.15. The results for different SplineCNN configurations match the observations from the MNIST superpixels experiments, where only small differences could be seen but using Cartesian coordinates and small B-spline degrees perform slightly better. Our SplineCNN outperforms all other approaches with 99.20% of predictions on the test set having *zero* geodesic error. However, the global behavior over larger geodesic error bounds is slightly worse in comparison to FMNet (Litany *et al.*, 2017). In Figure 3.14, it can be seen that most nodes are classified correctly but that the few false classifications have a high geodesic error. We contribute this differences to the varying loss formulations. While we train against a one-hot binary vector using the cross entropy loss, FMNet trains using a specialized soft error loss, which is a more geometrically meaningful criterion that punishes geodesically far-away predictions stronger than predictions near the correct node (Litany *et al.*, 2017). However, it is worth highlighting that we do not use SHOT descriptors as input features, like all other approaches do we compare against. Instead, we train only on the geometric structure of the meshes.

We conduct runtime experiments to highlight this benefit even further. In particular, we report an average forward step runtime of 0.043 seconds for a single FAUST example processed by the best performing SplineCNN architecture on a single NVIDIA GTX 1080 Ti. We train this network in approximately 40 minutes. Regarding scalability, we are able to stack up to 160 spline-based convolutional layers before running out-of-memory, while the runtime only scales linearly with the number of layers. However, for this task, we do not observe significant improvements in accuracy when using deeper networks.

4

Task-Specific Design of Graph Neural Networks

Although Graph Neural Networks are a powerful class of neural networks for operating on graph-structured data, they are also subject to various weaknesses that require task-specific solutions in order to reach desirable performance. One such weakness is their inability to propagate local information between distant nodes. In order to overcome this restriction, we propose the Dynamic Neighborhood Aggregation, which allows for a selective and node-adaptive aggregation of neighbors of potentially differing locality. Furthermore, Graph Neural Networks are limited by their incapability to detect certain sub-structures, e.g., circles, which is an important property in molecular learning. To circumvent, we propose the Hierarchical Inter-Message Passing architecture that leverages message-passing in higher-order substructures while still being very efficient. Furthermore, Graph Neural Networks are inherently local, which limits their applicability in tasks in which node-level predictions require global information, e.g., in the problem of graph matching. Here, we present a neural architecture named Deep Graph Matching Consensus that resolves ambiguities induced by locality via distributing global encodings in local neighborhoods.

4.1 Introduction 55
4.2 Locality-Adaptive Neighborhood Aggregation 57
4.3 Hierarchical Learning in Molecular Graphs 63
4.4 Graph Matching via Differentiable Neighborhood Consensus 72

4.1 Introduction

Graph Neural Networks (GNNs) are able to learn from graph-structured data in a flexible manner and end-to-end fashion, and can be applied to a wide range of ap-

plications, domains and learning tasks, *cf.* Chapter 3. In particular, they are able to reason about graph structure and feature information simultaneously without the necessity of manual feature engineering, which quickly led to major breakthroughs on a variety of benchmark datasets (Kipf & Welling, 2017; Gilmer *et al.*, 2017; Veličković *et al.*, 2018).

Despite their flexibility and expressivity, they are nonetheless subject to inherent weaknesses that prevent them from solving certain tasks with high precision, induced by the purely local and iteratively applied message passing formulation. In this chapter, we will present those weaknesses in detail, and propose solutions to overcome them in an efficient manner.

Specifically, in Section 4.2, we introduce the *over-squashing* or *over-smoothing* problem induced by the exponentially growing information pushed into fixed-sized vectors. In particular, such over-smoothing of node representations over layers limits a GNN's capability to propagate local information between distant nodes in the graph. Here, we propose the *Dynamic Neighborhood Aggregation* (*DNA*) approach, a procedure to allow a GNN to perform a selective and node-adaptive aggregation of neighbors of potentially differing locality. DNA exploits a highly adaptive neighborhood aggregation procedure based on *attention*, which is able to aggregate local information from neighbors far away, avoiding the over-smoothing problem in return.

Although GNNs are powerful in capturing a broad class of (sub-)graph structures, they are nonetheless unable to maintain information about *which* node in a receptive field has contributed *what* to the aggregated output, limiting their capabilities in, *e.g.*, detecting cycles. Such loss of information can have crucial impact on the final model performance in tasks in which cycles denote special meaning, *e.g.*, in molecular graphs. In Section 4.3, we present the *Hierarchical Inter-Message Passing* (*HIMP*) network, which can naturally overcome the aforementioned restriction of GNNs by jointly operating on two complementary graph representations: the original graph representation and its coarsened higher-order variant obtained via tree decomposition. Such higher-order information is then exchanged via inter-message passing in an efficient manner, which lets a GNN reason about hierarchies and cluster assignments, and strengthens its expressive power in return.

Since GNNs operate in a purely local fashion, they are unable to capture locations of nodes within the broader context of the graph structure. For example, if two nodes reside in very different parts of the graph but share topologically identical neighborhoods, they will be embedded to the same point in the embedding space. This creates ambiguities in learning tasks in which both topological and positional information is important, *e.g.*, in link prediction or graph matching. In Section 4.4, we present the *Deep Graph Matching Consensus* (*DGMC*) architecture to tackle the task of graph matching by incorporating both positional and topological information in a two-stage pipeline. In particular, ambiguities in matchings induced by locality are resolved by sparsely distributing global encodings in local neighborhoods around each node in order to reach a data-driven neighborhood consensus.

4.2 Locality-Adaptive Neighborhood Aggregation

Many different Graph Neural Network variants have been proposed and significantly advanced the state-of-the-art in Graph Representation Learning (Defferrard *et al.*, 2016; Kipf & Welling, 2017; Monti *et al.*, 2017; Gilmer *et al.*, 2017; Hamilton *et al.*, 2017; Veličković *et al.*, 2018; Fey *et al.*, 2018). However, while most of these approaches focus on novel instantiations of the neural message passing framework, deeply stacking those layers usually result in gradually decreasing performance despite having, in principal, access to a wider range of information (Kipf & Welling, 2017; Fey, 2019). This is, in particular, in high contrast to the application of deep learning in other domains, *e.g.*, image data, where deeply stacked neural networks allow to learn highly non-linear function approximators based on large receptive field sizes (Krizhevsky *et al.*, 2012; Simonyan & Zisserman, 2014; He *et al.*, 2016).

Such decreased performance is usually framed as the *over-smoothing* or *over-squashing* phenomenon of GNNs (Xu *et al.*, 2018; NT & Maehara, 2019; Zhao & Akoglu, 2020; Yan *et al.*, 2021; Zhu *et al.*, 2021a; Alon & Yahav, 2021), and explained by the exponentially growing neighborhood information pushed into fixed-sized vectors. In particular, it has been shown that there even exist some GNNs that produce node representations that become increasingly *washed out* and gradually more indistinguishable from each other w.r.t. model depth, while converging to a stationary point in case the number of layers goes to infinity (Li *et al.*, 2018a). Hence, most GNN formulations are implicitly designed under *homophily* assumptions, in which the required information for a given down-stream task is accessible from the direct local neighborhood (Zhu *et al.*, 2021a). As a result, GNNs based on homophily assumptions have difficulties to propagate local information between distant nodes in the graph, and are therefore limited to at most two or three layers. We give a detailed overview of the over-smoothing problem in GNNs and review related methods in Section 4.2.1.

In order to overcome the weakness of over-smoothing induced by deep Graph Neural Networks, we present the *Dynamic Neighborhood Aggregation* (*DNA*) in Section 4.2.2, a procedure that allows for a selective and node-adaptive aggregation of neighbors of potentially differing locality. Based on the so-called Jumping Knowledge networks (Xu *et al.*, 2018), DNA exploits a highly adaptive neighborhood aggregation procedure based on scaled dot-product attention (Vaswani *et al.*, 2017), which is able to aggregate local information from neighbors far away, avoiding the over-smoothing problem in return. In our experimental evaluation (Section 4.2.3), we show that our DNA approach is able to outperform traditional stacking of GNN layers, and does not result in decreasing performance the deeper our GNN gets.

4.2.1 State-of-the-Art

The over-smoothing phenomenon of GNNs has been analyzed by connecting the influence distribution of neighborhoods (Xu *et al.*, 2018), *i.e.* the range of nodes whose features affect a given node's representation, to the spread of random walks. Overall, they observe that the over-smoothing problem is well related to the given graph structure. For example, random walk distributions inside expander-like graph structures, *e.g.*, social networks, collapse rapidly in $\mathcal{O}(\log |\mathcal{V}|)$ steps to an almost-uniform distribution (Hoory *et al.*, 2006), thus leading to node representations that are almost equally

influenced by all other nodes in the graph and carry limited information about individual nodes. In contrast, random walks starting in trees with bounded width converge more slowly, letting GNNs be able to retain more local information (Xu *et al.*, 2018).

By analyzing GNNs from the graph signal processing perspective (Shuman *et al.*, 2013), it has been further shown that GNN instantiations that apply Laplacian-style propagation are subject to low-pass filter characteristics (Li *et al.*, 2018a; NT & Maehara, 2019). Thus, the graph structure only provides a mean to *denoise* initially given node representations, leading to more and more uninformative node representations w.r.t. model depth. Repeatedly applying smoothing too many times drives node representations to a stationary point, washing away all the information from these features (NT & Maehara, 2019). Notably, not all GNN instantiations are necessarily limited to perform low-pass filtering (Xu *et al.*, 2019c; Wang *et al.*, 2019e; Fey *et al.*, 2018).

Alon & Yahav (2021) explain the phenomenon of over-smoothing by relating the GNN computation graph to a *bottleneck*, in which exponentially growing information over layers is squashed into fixed-length vectors. This bottleneck hinders GNNs from fitting long-range signals in the training data. Notably, they find that GNNs that absorb incoming edges equally, such as GCN and Graph Isomorphism Network (GIN), are more susceptible to over-smoothing than attention-based variants such as Graph Attention Network (GAT) (Alon & Yahav, 2021).

Zhao & Akoglu (2020) propose two measures to quantify over-smoothing in GNNs: a row-diff measure that computes the average of all pair-wise distances between node features (*node-wise over-smoothing*), and a col-diff measure that computes the average of all pair-wise distances between the individual feature values of all nodes (*feature-wise over-smoothing*). As a result, they propose a graph-specific normalization technique that aims to maximize both measures.

Recent efforts in GNN model design also propose the usage of skip and residual connections or jumping knowledge to limit the effects of over-smoothing, *cf.* Section 3.2.4. For example, Hamilton *et al.* (2017) and Li *et al.* (2016b) propose to employ learnable skip-connections and Gated Recurrent Units (GRUs) to be able to preserve localized central node information, respectively. Alternative ideas utilize the concepts of residual connections to be able to *teleport back* to earlier representations (Klicpera *et al.*, 2019a; Chen *et al.*, 2020b). Jumping Knowledge networks (Xu *et al.*, 2018) aggregate layer-wise representations to obtain the final output representation, *e.g.*, via concatenation, pooling or attention.

Furthermore, Yan *et al.* (2021) relate over-smoothing to the homophily assumption in common GNNs via a unified theoretical framework. Formally, the homophily of a graph can be determined by the fraction of edges whose nodes share the same class label (*intra-class edges*). In particular, Yan *et al.* (2021) analyze the capabilities of GNNs to linearly separate node representations that belong to different classes w.r.t. the number of layers. As a result, allowing GNNs to effectively operate in heterophily graphs (which requires the aggregation of informative representations from distant nodes) weakens the effects of over-smoothing and vice versa.

This observation has motiviated a tremendous amount of research regarding the effective application of GNNs in a heterophily graph setting. Zhu *et al.* (2020) identify a set of key design principles to boost graph learning in heterophily, *e.g.*, ego- and neighbor-embedding separation, higher-order neighborhoods, and intermediate rep-

resentation re-use. Our DNA approach extends these insights by using intermediate representations to guide propagation in a novel fashion. Zhu *et al.* (2021a) propose a two-stage architecture, which first estimates the class of a node in isolation, and uses the obtained estimation to guide the propagation process in a second stage. Yan *et al.* (2021) propose *signed* message passing, *e.g.*, via negative attention weights. Similarly, DNA extends the attention procedure to be able to discard uninformative information.

4.2.2 Attentional Jumping Knowledge Neighborhood Aggregation

Closely related to the Jumping Knowledge (JK) networks (Xu *et al.*, 2018), we are seeking for a way to let a GNN node-adaptively craft receptive-fields for a specific task at hand, *cf.* Section 3.2.4. JK nets achieve this by dynamically jumping to the most representative layer-wise embedding *after* a fixed range of node representations were obtained. As a result, Jumping Knowledge can not guarantee that higher-order features will not become washed out in later layers, but instead will fall back to more localized information preserved from earlier representations (Fey, 2019). Furthermore, fine-grained details may still get lost very early on in expander-like subgraph structures despite the use of Jumping Knowledge (Hoory *et al.*, 2006; Xu *et al.*, 2018).

In contrast, we propose to allow *jumps* to earlier knowledge immediately *while* aggregating information from neigbors. This results in a highly-dynamic receptive-field in which neighborhood information is potentially gathered from representations of differing locality. Each node's representation controls its own spread-out, possibly aggregating more global information in one branch, and falling back to more local information in others (Fey, 2019).

Formally, we allow each node-neighborhood pair $(w, v) \in \mathcal{E}$ to attend to *all* its former representations $\boldsymbol{h}_w^{(1)}, \ldots, \boldsymbol{h}_w^{(\ell-1)}$, and use its most informative representation for neighborhood aggregation (Fey, 2019):

$$\boldsymbol{h}_v^{(\ell)} = f_{\boldsymbol{\theta}}^{(\ell)} \left(\boldsymbol{h}_{v \leftarrow v}^{(\ell)}, \left\{\!\!\left\{ \boldsymbol{h}_{v \leftarrow w}^{(\ell)} : w \in \mathcal{N}(v) \right\}\!\!\right\} \right) \tag{4.1}$$

where

$$\boldsymbol{h}_{v \leftarrow w}^{(\ell)} = \textsc{Attention}_{\boldsymbol{\theta}}^{(\ell)} \left(\boldsymbol{h}_v^{(\ell-1)}, \left[\boldsymbol{h}_w^{(1)}, \ldots, \boldsymbol{h}_w^{(\ell-1)} \right] \right), \tag{4.2}$$

in which $\textsc{Attention}_{\boldsymbol{\theta}}^{(\ell)}(\cdot, \cdot)$ computes a neighboring node embedding for w based on all its previous representations $[\boldsymbol{h}_w^{(1)}, \ldots, \boldsymbol{h}_w^{(\ell-1)}]$, dependent on v. In practice, we implement $\textsc{Attention}_{\boldsymbol{\theta}}^{(\ell)}(\cdot, \cdot)$ via scaled dot-product attention (Vaswani *et al.*, 2017). That is,

$$\textsc{Attention}_{\boldsymbol{\theta}}(\boldsymbol{q}, \boldsymbol{K}) = \mathrm{softmax} \left(\frac{\boldsymbol{q}^\top \boldsymbol{W}_1 \cdot (\boldsymbol{K} \boldsymbol{W}_2)^\top}{\sqrt{D}} \right) \boldsymbol{K} \tag{4.3}$$

with $\boldsymbol{W}_1$ and $\boldsymbol{W}_2 \in \mathbb{R}^{D \times D}$ denoting trainable symmetric projection matrices (Vaswani *et al.*, 2017; Fey, 2019). Dot-product attention weights a set of keys $\boldsymbol{K} \in \mathbb{R}^{|\mathcal{V}| \times D}$ according to a given query $\boldsymbol{q} \in \mathbb{R}^D$ by using the softmax-normalized results as weighting coefficients. As such, the usage of $\textsc{Attention}_{\boldsymbol{\theta}}^{(\ell)}$ as part of a GNN pipeline will compute the most informative representation of node w for aggregation to node v, *e.g.*, more global information from later layers or more local information from earlier ones. Importantly, attended information is both dependent on both node v and w,

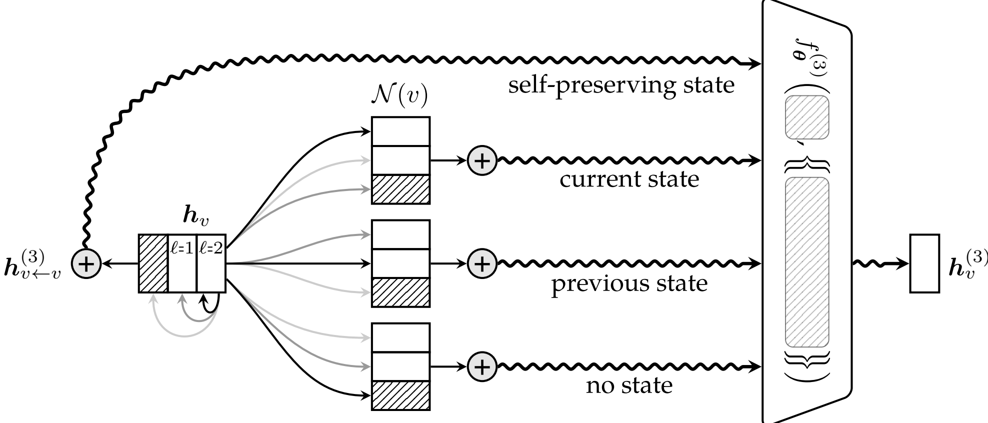


Figure 4.1: Overview of the computation flow of our DNA procedure. Given current node representation $\boldsymbol{h}_v^{(2)}$ as query, a node-adaptive embedding $\boldsymbol{h}_{v \leftarrow w}^{(3)}$ gets computed for all neighbors $w \in \mathcal{N}(v)$ based on their former representations $\boldsymbol{h}_w^{(1)}$ and $\boldsymbol{h}_w^{(2)}$, either preserving current state, previous state, or no state at all. In addition, self-attention is applied to retain central node information as well, *e.g.*, when applied within a GCN (Fey *et al.*, 2020b).

leading to a node-adaptive aggregation that is different for every node in the graph (while still being able to generalize to unseen nodes). By ensuring that earlier information can be preserved, our operator can be stacked deep by design, in particular without the need of JK nets (Fey, 2019).

In practice, we replace the single attention module by *multi-head* attention with a user-defined number of heads while maintaining the same number of parameters. We implemented $f_{\boldsymbol{\theta}}^{(\ell)}$ as the Graph Convolutional Network (GCN) operator from Kipf & Welling (2017), although any other GNN layer is applicable as well. For example, using the GAT operator (Veličković *et al.*, 2018) allows us to perform both attention-based message passing across the number of layers and the number of neighbors.

Importantly, we also incorporate an additional parameter to the softmax distribution of the $\textsc{Attention}_{\boldsymbol{\theta}}^{(\ell)}$ module to allow the model to *refuse* the aggregation from individual neighbors, as illustrated in Figure 4.1. Instead of actually over-parametrizing the resulting distribution, we restrict this parameter to be fixed (Goodfellow *et al.*, 2016), leading to a softmax function of the form (Fey, 2019)

$$\text{softmax}(\boldsymbol{x})_i = \frac{\exp(x_i)}{1 + \sum_j \exp(x_j)}. \tag{4.4}$$

In case all attention scores of previous layers are pushed to large negative values, *no* information will be received from this particular neighbor. As such, the model can learn to avoid aggregation from neighbors that might be irrelevant for a given downstream task.

Our proposed operator does scale linearly in the number of previously seen node representations for each edge, *i.e.* $\mathcal{O}(L \cdot |\mathcal{E}|)$. Importantly, the slight increase in runtime complexity does not effect memory consumption of GNN training at all, as node rep-

resentations for all layers need to be kept in memory anyways in order to apply back propagation. Notably, in order to leverage the attention module, input and output feature dimensionalities are forced to remain equal across *all* layers. We found this to be only a weak constraint as it is common practice in most GNN architectures (Gilmer *et al.*, 2017; Xu *et al.*, 2018).

Lastly, our attention module introduces additional parameters to a GNN pipeline. In order to avoid over-fitting on small-scale graphs, we apply dropout (Srivastava *et al.*, 2014) to the softmax-normalized attention weights and make use of *grouped linear projections* (Krizhevsky *et al.*, 2012) for weights $\boldsymbol{W}_1^{(\ell)}$ and $\boldsymbol{W}_2^{(\ell)}$. Grouped linear projections control the channel-wise connections between an input $\boldsymbol{x} \in \mathbb{R}^D$ and an output $\boldsymbol{y} \in \mathbb{R}^D$ by reducing the number of parameters by G, the number of groups. G must be chosen so that D is divisible by G. If $G = D$, the operation is performed independently over every channel in D (Chollet, 2017). The grouped projections regulate the attention heads by forcing them to only have a local influence on other attention heads (or even restricting them to have no influence at all). We observed that these adjustments greatly help the model to avoid over-fitting while still maintaining large hidden feature dimensionalities (Fey, 2019).

4.2.3 Evaluation

We evaluate our DNA approach on eight transductive benchmark datasets: the tasks of classifying academic papers (Cora, CiteSeer, PubMed, Cora Full) (Sen *et al.*, 2008; Bojchevski & Günnemann, 2018), active research fields of authors in co-author graphs (Coauthor CS, Coauthor Physics) (Shchur *et al.*, 2018) and classifying product categories in co-purchase graphs (Amazon Computers, Amazon Photo) (Shchur *et al.*, 2018). We randomly split nodes into 20% nodes for training, 20% for validation, and 60% for testing. The code of DNA and all its training scripts is integrated into the PyTorch Geometric[1] (Fey & Lenssen, 2019) library.

We compare our DNA approach to GCN (Kipf & Welling, 2017) and GAT (Veličković *et al.*, 2018) with and without Jumping Knowledge, closely following the network architectures as described in Xu *et al.* (2018): We first project node features separately into a lower-dimensional space, apply a number of GNN layers $L \in \{1, 2, 3, 4, 5\}$ with hidden feature dimensionality $F \in \{16, 32, 64, 128\}$ and ReLU non-linearity, and perform the final prediction via a fully-connected layer. All models were implemented using grouped linear projections and evaluated with the number of groups $G \in \{1, 8, 16\}$.

We use the Adam optimizer (Kingma & Ba, 2015) with a learning rate of 0.005 and stop training early with a patience value of 10. We apply a fixed dropout rate of 0.5 before and after GNN layer execution and apply L_2 regularization of $5 \cdot 10^{-4}$ to all model parameters. For our proposed model and GAT, we additionally tune the number of heads $H \in \{8, 16\}$ and set the dropout rate of attention weights to 0.8. Hyperparameter configurations of the best performing models with respect to the validation set are reported in Fey (2019).

Table 4.1 shows the average classification accuracy over 10 random data splits and initializations. Our DNA approach outperforms traditional stacking of GNN layers

[1] Code for DNA: `https://github.com/rusty1s/pytorch_geometric` (last access: August 25, 2022)

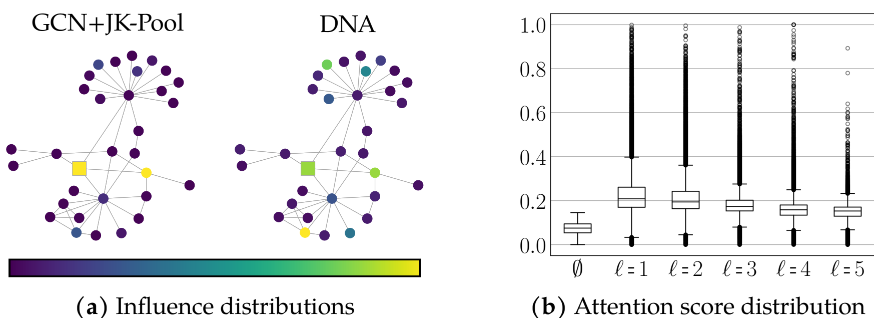
	Model	Cora	CiteSeer	PubMed	Cora Full
GCN	JK-None	83.20±0.98	73.87±0.81	86.93±0.25	62.55±0.60
	JK-Concat	83.99±0.72	73.77±0.89	87.52±0.25	65.62±0.49
	JK-Pool	84.36±0.62	73.86±0.97	87.61±0.27	65.14±0.81
	JK-LSTM	80.46±0.88	72.92±0.69	87.38±0.29	55.39±0.40
GAT	JK-None	**86.35**±0.74	73.70±0.53	86.76±0.25	65.70±0.32
	JK-Concat	84.70±0.57	73.97±0.46	**88.73**±0.30	66.18±0.47
	JK-Pool	83.91±0.87	73.42±0.71	88.44±0.33	61.52±1.17
	JK-LSTM	78.08±1.53	71.84±1.20	87.85±0.26	55.41±0.35
DNA	$G = 1$	83.88±0.50	73.37±0.83	87.80±0.25	63.72±0.44
	$G = 8$	85.86±0.45	74.19±0.66	88.04±0.17	66.50±0.42
	$G = 16$	86.15±0.57	**74.50**±0.62	88.04±0.22	**66.64**±0.47
	Model	**Coauthor CS**	**Coauthor Physics**	**Amazon Computers**	**Amazon Photo**
GCN	JK-None	92.90±0.14	95.90±0.16	89.32±0.20	93.11±0.27
	JK-Concat	95.44±0.32	96.71±0.15	90.27±0.28	94.74±0.29
	JK-Pool	**95.47**±0.21	**96.74**±0.17	90.30±0.37	94.64±0.24
	JK-LSTM	94.40±0.28	96.55±0.08	90.06±0.23	94.54±0.30
GAT	JK-None	93.54±0.17	96.21±0.08	88.02±1.39	93.00±0.42
	JK-Concat	95.12±0.18	96.66±0.09	89.67±0.59	94.93±0.31
	JK-Pool	94.84±0.16	96.62±0.06	89.42±0.47	94.80±0.24
	JK-LSTM	94.09±0.23	96.45±0.05	87.26±1.82	94.47±0.33
DNA	$G = 1$	94.02±0.17	96.49±0.10	90.52±0.40	94.89±0.26
	$G = 8$	94.46±0.15	96.58±0.09	**90.99**±0.40	94.96±0.24
	$G = 16$	94.64±0.15	96.53±0.10	90.81±0.38	**95.00**±0.19

Table 4.1: Results of our DNA approach in comparison to GCN and GAT with and without Jumping Knowledge. Mean accuracy and standard deviations are computed across 10 random data splits and initializations (Fey, 2019).

(JK-None) and even exceeds the performance of using Jumping Knowledge in most cases. Noticeably, the use of grouped linear projections greatly improves attention-based approaches, especially when combined with large hidden feature dimensionalities. We noticed gains in accuracy up to 3 percentage points when comparing the best results of $G = 1$ to $G > 1$, both for GAT and DNA. Best hyperparameter configurations in Fey (2019) show advantages in using increased feature dimensionalities across all datasets. However, for vanilla GCN, we found those gains to be negligible. Similar to JK nets, our approach benefits from an increased amount of stacked layers, and most importantly, the performance does not decrease when increasing the number of layers (Fey, 2019).

Furthermore, we perform a qualitative analysis of DNA on the Cora dataset by analyzing the *influence score*. The influence score $I_v(w)$ measures the sensitivity of a node w to a node v, *i.e.* the influence a node w has on the prediction for node v (Xu *et al.*, 2018). Formally, it is given by

$$I_v(w) = \mathbf{1}^\top \left[\frac{\partial \boldsymbol{h}_v^{(L)}}{\partial \boldsymbol{h}_w^{(0)}} \right] \mathbf{1}, \tag{4.5}$$



(**a**) Influence distributions (**b**) Attention score distribution

Figure 4.2: Qualitative analysis of DNA. (**a**) The influence distribution of a node which is correctly classified by DNA, but is incorrectly classified by GCN+JK-Pool. While the node in GCN+JK-Pool is purely influenced from a node nearby, DNA is able to aggregate local information from distant nodes. (**b**) The attention score distribution of a 5-layer DNA-GNN model. DNA attends to both nearby as well as distant nodes (Fey, 2019).

where $\mathbf{1}$ denotes the all-ones vector, *i.e.* it denotes the sum of the absolute values of the entries of the Jacobian matrix $\left[\frac{\partial \boldsymbol{h}_v^{(L)}}{\partial \boldsymbol{h}_w^{(0)}}\right]$ (Xu *et al.*, 2018). We visualize the differences in aggregation by looking at the influence scores for a node which is correctly classified by DNA, but is incorrectly classified by GCN+JK-Pool, *cf.* Figure 4.2a. While the node embedding produced by GCN+ JK-Pool is nearly exclusively influenced by its central node and a node nearby, DNA is able to aggregate *localized* information even from nodes far away (Fey, 2019). Figure 4.2b visualizes the attention score distribution of a 5-layer DNA-GNN. It signals that aggregations typically attend to earlier representations on the Cora dataset, which verifies that nearby information is indeed often sufficient to classify most nodes in homophily graphs. However, there are some nodes that do make heavy usage of information retrieved from latter representations, indicating the merits of a Dynamic Neighborhood Aggregation procedure (Fey, 2019).

4.3 Hierarchical Learning in Molecular Graphs

Next, we look into the incapability of GNNs to detect certain sub-structures (*cf.* Section 3.4) which play an important role in certain documents, *e.g.*, detecting circles in molecular graphs.

Machine learning algorithms offer great potentials in reducing the computation time required for predicting molecular properties from several hours to just a few milliseconds (Wu *et al.*, 2018), leading to many fruitful opportunities in chemistry, drug discovery, and materials science (Gilmer *et al.*, 2017). To date, most applied machine learning in chemistry tasks resolves around manual feature engineering (Rupp *et al.*, 2012; Rogers & Hahn, 2010; Montavon *et al.*, 2012; Behler & Parrinello, 2007; Schoenholz *et al.*, 2016; Gilmer *et al.*, 2017). However, with the rise of large-scale quantum chemistry calculations and molecular dynamic simulations that generate data at an unprecedented rate, these approaches generally struggle to make effective use of the large amount of data available nowadays (Gilmer *et al.*, 2017).

With the advent of Graph Neural Networks, machine learning in chemistry has experienced a small revolution, exceeding the previously predominated approach of manual feature engineering by a large margin (Gilmer *et al.*, 2017; Schütt *et al.*, 2017; Klicpera *et al.*, 2020b). Here, molecules are represented as graphs $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, in which nodes $\mathcal{V}$ denote atoms and edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ denote connections between them, *e.g.*, given by a pre-defined structure such as in the form of atomic bonds, or by connecting atoms that lie within a certain cutoff distance. Such natural and raw molecular graph representations allow GNNs to *learn* high-dimensional embeddings of atoms that are able to represent their complex interactions by exchanging and aggregating messages between them (Klicpera *et al.*, 2020b; Fey *et al.*, 2020b).

However, it has been shown that GNNs are unable to distinguish certain molecular graphs, *e.g.*, Cyclohexane and two Cyclopropane molecules (Xu *et al.*, 2019c; Klicpera *et al.*, 2020b). These restrictions mostly stem from the fact that GNNs are not capable of detecting cycles (Loukas, 2020) since they are unable to maintain information about *which* node in a receptive field has contributed *what* to the aggregated information (Hy *et al.*, 2018). These limitations of GNNs have encouraged lots of research to allow for more expressive GNNs, either in the form of GNN design (Murphy *et al.*, 2019b; Klicpera *et al.*, 2020b) or by injecting such information as part of an additional hand-crafted input feature set, *e.g.*, whether an atom is part of a ring (Hu *et al.*, 2020a,b). However, the former leads to overly complex models with high inference runtime requirements (Murphy *et al.*, 2019b; Hy *et al.*, 2018; Albooyeh *et al.*, 2019) or to models that require information about 3D equilibrium structures (Schütt *et al.*, 2017; Gilmer *et al.*, 2017; Jørgensen *et al.*, 2018; Unke & Meuwly, 2019; Chen *et al.*, 2019a; Klicpera *et al.*, 2020b), which is in itself expensive to compute via Density Functional Theory (Helgaker *et al.*, 2014) (may take up to several hours per small molecule (Hu *et al.*, 2021b)). The latter contradicts with the desire of hand-crafted feature independence. Adding such additional structural features to the input set again limits the model's capabilities to the choice of input features.

Here, we present the *Hierarchical Inter-Message Passing* (*HIMP*) network, a new approach for learning on molecular graph structures. HIMP is both efficient to train and can naturally overcome the restrictions of traditional GNNs, strengthening their performance with minimal computational overhead in return. Our model utilizes two separate Graph Neural Networks that operate on two complementary graph representations: the raw molecular graph representation and its associated junction tree (Section 4.3.2), in which nodes represent meaningful clusters in the original graphs, *e.g.*, rings or bridged compounds. We then proceed to learn a molecule's representation by passing messages inside each graph, and exchange messages between the two representations using a coarse-to-fine and fine-to-coarse information flow (Section 4.3.3). This allows the network to reason about hierarchy in molecules in a natural fashion. We show that this simple scheme can drastically increase the performance of a GNN, reaching state-of-the-art performance on a variety of different molecular graph datasets (Section 4.3.4). Despite its higher-order nature, our HIMP network architecture is still very efficient to train and causes only marginal additional costs in terms of memory and execution time (Fey *et al.*, 2020b).

4.3.1 State-of-the-Art

We briefly review state-of-the-art learning on molecular graphs, and discuss the related work and their relation to our proposed HIMP approach.

4.3.1.1 Learning on Molecular Graphs. Instead of using hand-crafted representations (Bartók *et al.*, 2013), recent advancements in deep graph learning rely on an end-to-end learning of representations which has quickly led to major breakthroughs in machine learning on molecular graphs (Duvenaud *et al.*, 2015; Gilmer *et al.*, 2017; Schütt *et al.*, 2017; Jørgensen *et al.*, 2018; Unke & Meuwly, 2019; Chen *et al.*, 2019a). Most of these works are especially designed for learning on the 3D equilibrium molecular structure obtained from Density Functional Theory (Helgaker *et al.*, 2014). Here, earlier models (Schütt *et al.*, 2017; Gilmer *et al.*, 2017; Jørgensen *et al.*, 2018; Unke & Meuwly, 2019; Chen *et al.*, 2019a) fulfill rotational invariance constraints by relying on inter-atomic distances, while recent models employ more expressive variants. For example, DimeNet (Klicpera *et al.*, 2020b) deploys directional message passing between node triplets to also model angular potentials. It was further extended in Klicpera *et al.* (2020a) to a faster variant that can also handle non-equilibrium molecules. Another line of work breaks symmetries by taking permutations of nodes into account (Murphy *et al.*, 2019b; Hy *et al.*, 2018; Albooyeh *et al.*, 2019). Recently, it has been shown that strategies for pre-training models on molecular graphs can effectively increase their performance for certain down-stream tasks (Hu *et al.*, 2020b). Our approach fits nicely into these lines of work since it also increases the expressiveness of GNNs while being orthogonal to further advancements in this field (Fey *et al.*, 2020b).

4.3.1.2 Junction Trees. So far, junction trees have solely been used for molecule generation based on a coarse-to-fine generation procedure (Jin *et al.*, 2018, 2019). In contrast to the generation of SMILES strings (Gómez-Bombarelli *et al.*, 2018), this allows the model to enforce chemical validity while generating molecules significantly faster than the node-per-node generation procedure applied in autoregressive methods (You *et al.*, 2018; Fey *et al.*, 2020b).

4.3.1.3 Inter-Message Passing. The idea of inter-message passing between graphs has been already heavily investigated in practice, mostly in the fields of deep graph matching (Wang *et al.*, 2018a; Li *et al.*, 2019a; Fey *et al.*, 2020a) and graph pooling (Ying *et al.*, 2018b; Cangea *et al.*, 2018; Gao & Ji, 2019). For graph pooling, most works focus on *learning* a coarsened version of the input graph. However, due to being learned, the coarsened graphs are unable to strengthen the expressiveness of GNNs by design, *cf.* Section 4.3.2. For example, DiffPool (Ying *et al.*, 2018b) always maps the atoms of two disconnected rings to the *same* cluster, while attention-based pooling approaches (Cangea *et al.*, 2018; Gao & Ji, 2019) either keep or remove *all* atoms inside those rings (since their attention scores are shared). The approach that comes closest to ours involves inter-message passing to a "virtual" node that is connected to *all* atoms (Gilmer *et al.*, 2017; Hu *et al.*, 2020a). Our approach can be seen as a simple yet effective extension to this procedure (Fey *et al.*, 2020b). Furthermore, inter-message passing is also closely related to learning in *hypergraphs*, in which messages from nodes are passed to intermediate arbitrary node-clusters (rather than pairs of nodes) and vice versa (Feng *et al.*, 2019; Zhang *et al.*, 2020b; Bai *et al.*, 2021).

4.3.2 Higher-Order Graph Coarsening via Tree Decompositions

It has been shown that GNNs are unable to distinguish certain molecular graphs (Xu *et al.*, 2019c; Klicpera *et al.*, 2020b). These restrictions mostly stem from the fact that GNNs are not capable of detecting cycles (Loukas, 2020) since they are unable to maintain information about *which* node in its receptive field has contributed *what* to the aggregated information (Hy *et al.*, 2018). However, this ultimately limits the capabilities of GNNs for learning on molecular graphs, since information about rings in molecular graphs provide crucial information.

As a result, we propose a simple yet effective method named HIMP to overcome the shortcomings of existing GNNs (Fey *et al.*, 2020b). In particular, our method involves learning on two molecular graph representations simultaneously in an end-to-end fashion: the original graph representation and its associated junction tree. The junction tree representation encodes the tree structure of molecules and defines how clusters (singletons, bonds, rings, bridged compounds) are mutually connected, while the original graph structure of the molecular graph captures its more fine-grained connectivity (Jin *et al.*, 2018). The addition of such coarse-grained information of molecular graphs provides two major benefits: (1) It provides task-specific information about meaningful clusters in the graph that a GNN cannot detect on its own, (2) the usage of a tree structure helps to avoid the GNN's shortcomings on the coarse-grained representation (Fey *et al.*, 2020b).

A *junction tree* of a graph $\mathcal{G}$ can be obtained via *tree decomposition* such that certain nodes are contracted into single nodes so that $\mathcal{G}$ becomes cycle-free (Fey *et al.*, 2020b). Formally, given a graph $\mathcal{G}$, a tree decomposition maps $\mathcal{G}$ into a junction tree $\mathcal{T} = (\mathcal{C}, \mathcal{R})$ with node set $\mathcal{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$, $\mathcal{C}_i \subseteq \mathcal{V}$ for all $i \in \{1, \ldots, m\}$, and edge set $\mathcal{R} \subseteq \mathcal{C} \times \mathcal{C}$ so that (Jin *et al.*, 2018):

- All nodes and edges are included in the coarsened junction tree, *i.e.* it holds that $\bigcup_i \mathcal{C}_i = \mathcal{V}$ and $\bigcup_i \mathcal{E}[\mathcal{C}_i] = \mathcal{E}$ with $\mathcal{E}[\mathcal{C}_i] \subseteq \mathcal{C}_i \times \mathcal{C}_i$ denoting the edge set of the induced subgraph $\mathcal{G}[\mathcal{C}_i]$.
- $\mathcal{T}$ is cycle-free, *i.e.* it holds that $\mathcal{C}_i \cap \mathcal{C}_j \subseteq \mathcal{C}_k$ for all clusters $\mathcal{C}_i$, $\mathcal{C}_k$, $\mathcal{C}_j$ with connections $(\mathcal{C}_i, \mathcal{C}_k) \in \mathcal{R}$ and $(\mathcal{C}_k, \mathcal{C}_j) \in \mathcal{R}$ (*running intersection*).

Notably, there exists a dedicated tree decomposition algorithm tailored for molecules, which finds its root in chemistry (Rarey & Dixon, 1998), and which was recently used for generating chemically valid molecules in a deep learning pipeline (Jin *et al.*, 2018). We closely follow this tree decomposition algorithm in our work as well (Fey *et al.*, 2020b):

Given a molecular graph $\mathcal{G}$, we first group all its simple cycles and all edges that do not belong to any cycle into distinct clusters in $\mathcal{C}$. Two rings are merged together if they share more than two overlapping atoms, since they constitute a specific structure called bridged compounds (Clayden *et al.*, 2001; Jin *et al.*, 2018). Each of those cycles and edges is considered as a cluster. As a result of ring merging, any two clusters have at most two atoms in common (Jin *et al.*, 2018). For atoms lying inside more than three clusters, we add the intersecting atom as a singleton cluster. A cluster graph is constructed by adding edges between all intersecting clusters, and the final junction tree $\mathcal{T}$ is then given as one its spanning trees. Figure 4.3 visualizes how clusters are formed on an exemplary molecule. For each cluster, we additionally hold its

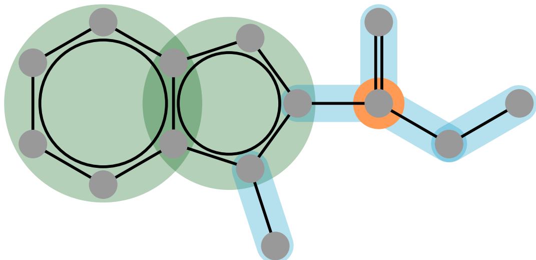


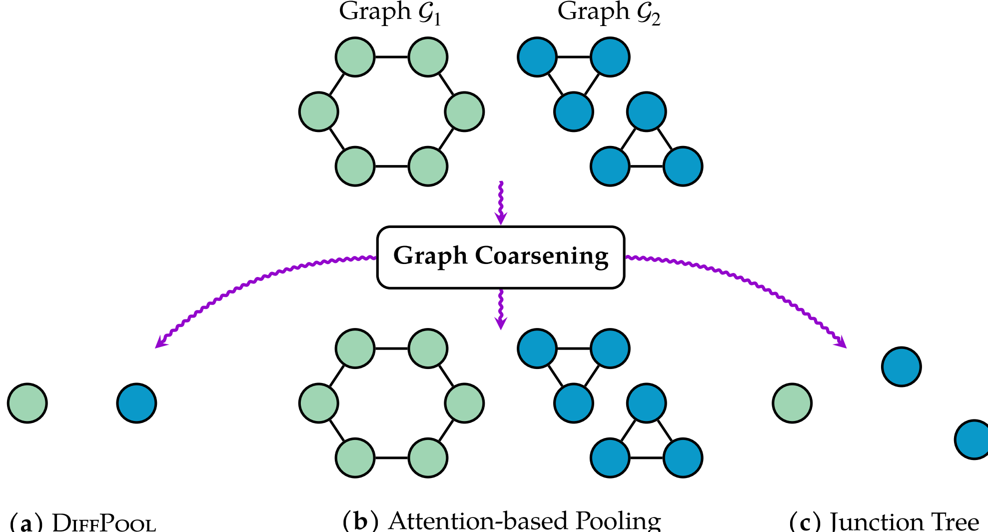
Figure 4.3: A molecular graph and the cluster assignment of its junction tree. Cluster colors refer to ■ singletons, ■ bonds and ■ rings (Fey *et al.*, 2020b).

respective category (singleton, bond, ring, bridged compound) as a one-hot encoding feature vector $\boldsymbol{z}_i$, *i.e.* $\boldsymbol{z}_i \in \{0,1\}^4$, with $i \in \{1, \ldots, |\mathcal{C}|\}$ (Fey *et al.*, 2020b).

It is important to note that the junction tree representation of a molecule is complementary to its original molecular graph representation. As such, when used in a deep learning pipeline, it provides additional and important higher-order information that cannot be captured by a GNN on the molecular graph alone. In contrast, other related graph coarsening approaches rely on learning to find meaningful clusters in a graph (Ying *et al.*, 2018b; Cangea *et al.*, 2018; Gao & Ji, 2019). This is contradictory in case a GNN does not have the power to find such clusters in the first place, *e.g.*, a GNN will never be able to group rings into a single cluster. Figure 4.4 visualizes how different differentiable pooling algorithms cluster certain graph structures in comparison to using a pre-processed higher-order coarsening scheme in the form of a junction tree. Here, we compare DiffPool (Ying *et al.*, 2018b) and the attention-based pooling approaches (Cangea *et al.*, 2018; Gao & Ji, 2019) with the clustering procedure based on junction tree decomposition. Since the node representations obtained by a GNN will be identical in these two graphs, DiffPool will assign all nodes to the same cluster. An attention-based pooling approach computes an attention score to determine which nodes will be kept for coarsening. As scores will be identical for all nodes, all nodes and their original graph structure will be kept. Notably, for both pooling approaches it holds that both graphs are still indistinguishable after graph coarsening is performed. Hence, learned graph coarsening approaches are limited by the expressive power of the underlying GNN. In contrast, the higher-order information induced by the junction tree decomposition approach is able to distinguish both graphs after coarsening takes place, as the two cycles in th egraph $\mathcal{G}_2$ in Figure 4.4 will be assigned to two different clusters instead of one.

4.3.3 Inter-Message Passing with Junction Trees

We now present our HIMP architecture (Fey *et al.*, 2020b) for learning on molecular graphs, based on the previous definition of junction trees in Section 4.3.2. A high-level overview of our method is visualized in Figure 4.5. Our method is able to extend any GNN model for molecular property prediction by making use of intra-message passing *in* and inter-message passing *to* and *from* a complementary junction tree representation. Here, instead of using a single GNN operating on the molecular graph, we make use of *two* GNN models: one operating on the original graph $\mathcal{G}$ and one operat-



(**a**) DiffPool (**b**) Attention-based Pooling (**c**) Junction Tree

Figure 4.4: The output graphs after coarsening two graphs (■ and ■) using different graph coarsening procedures, as indicated by the ■ arrows. Each graph coarsening technique will coarsen the two input graphs differently. (**a**) Since node representations obtained by a GNN will be identical, DiffPool assigns all nodes to the same cluster. (**b**) Attention-based pooling approaches keep all original nodes, as attention scores are shared across all nodes. (**c**) The junction tree decomposition procedure injects higher-order information into the cluster assignment decision, and is therefore able to coarsen the two graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ differently.

ing on its associated junction tree $\mathcal{T}$, each passing intra-messages to their respective neighbors, *i.e.*

$$\boldsymbol{h}_v^{(\ell)} = f_{\boldsymbol{\theta}}^{(\ell)}\big(\boldsymbol{h}_v^{(\ell-1)}, \{\!\{(\boldsymbol{h}_w^{(\ell-1)}, \boldsymbol{e}_{w,v}) : (w,v) \in \mathcal{E}\}\!\}\big) \tag{4.6}$$

and

$$\tilde{\boldsymbol{h}}_i^{(\ell)} = g_{\boldsymbol{\theta}}^{(\ell)}\big(\tilde{\boldsymbol{h}}_i^{(\ell-1)}, \{\!\{\tilde{\boldsymbol{h}}_j^{(\ell-1)} : (\mathcal{C}_j, \mathcal{C}_i) \in \mathcal{R}\}\!\}\big), \tag{4.7}$$

where $\boldsymbol{h}_v^{(\ell)}$ and $\tilde{\boldsymbol{h}}_i^{(\ell)}$ represent node and cluster representations obtained from $\mathcal{G}$ and $\mathcal{T}$ using distinct GNN operators $f_{\boldsymbol{\theta}}^{(\ell)}$ and $g_{\boldsymbol{\theta}}^{(\ell)}$, respectively (Fey *et al.*, 2020b). We further enhance this scheme by making use of an *inter-message passing* flow. That is, after each round of message passing, we enhance $\tilde{\boldsymbol{h}}_i^{(\ell)}$ by an additional fine-to-coarse information flow from $\mathcal{G}$ to $\mathcal{T}$

$$\tilde{\boldsymbol{h}}_i^{(\ell)} \leftarrow \tilde{\boldsymbol{h}}_i^{(\ell)} + \sigma\left(\boldsymbol{W}_1^{(\ell)} \sum_{v \in \mathcal{C}_i} \boldsymbol{h}_v^{(\ell)}\right), \tag{4.8}$$

and $\boldsymbol{h}_v^{(\ell)}$ by an additional coarse-to-fine information flow from $\mathcal{T}$ to $\mathcal{G}$

$$\boldsymbol{h}_v^{(\ell)} \leftarrow \boldsymbol{h}_v^{(\ell)} + \sigma\left(\boldsymbol{W}_2^{(\ell)} \sum_{\substack{\mathcal{C}_i \in \mathcal{C}, \\ v \in \mathcal{C}_i}} \tilde{\boldsymbol{h}}_i^{(\ell)}\right), \tag{4.9}$$

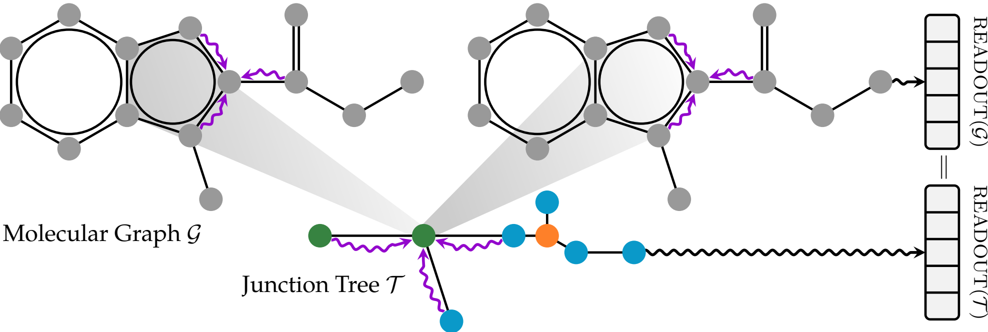


Figure 4.5: Overview of the computation flow of our HIMP network architecture. Two separate GNNs are operating on the distinct graph representations $\mathcal{G}$ and $\mathcal{T}$, and receive coarse-to-fine and fine-to-coarse information before another round of message passing starts. Finally, both node-level and cluster-level representations are aggregated into a unified graph-level representation, used as input to a given down-stream task (Fey *et al.*, 2020b).

with $\boldsymbol{W}_1^{(\ell)}, \boldsymbol{W}_2^{(\ell)}$ denoting trainable weight matrices, and σ being a non-linearity (Fey *et al.*, 2020b). After L rounds of message passing, the *readout* function (Section 3.2.3) of the model aggregates both node-level and cluster-level representations into a unified graph-level representation, *i.e.*

$$\boldsymbol{h}_{\mathcal{G}} = \left[\sum_{v \in \mathcal{V}} \boldsymbol{h}_v^{(L)}, \sum_{\mathcal{C}_i \in \mathcal{C}} \tilde{\boldsymbol{h}}_i^{(L)} \right]. \tag{4.10}$$

The final graph-level representation can then be used as input to a given down-stream task.

Overall, our HIMP architecture leads to a *hierarchical-variant* of message passing for learning on molecular graphs, similar to the ones applied in computer vision (Ronneberger *et al.*, 2015; Newell *et al.*, 2016; Lin *et al.*, 2017). Furthermore, each atom is able to know about its cluster assignment, and, more importantly, which other nodes are part of the same cluster (Fey *et al.*, 2020b). Specifically, this leads to an increased expressivity of the underlying GNN. For example, the popular example of a Cyclohexane molecule and two Cyclopropane molecules (a single ring and two disconnected rings) (Klicpera *et al.*, 2020b) are now distinguishable by our scheme since its junction tree representations are distinguishable, *cf.* Figure 4.4.

While our HIMP architecture is provably more expressive by incorporating higher-order information into its message passing scheme, it is still very efficient to train. In particular, the computational complexity of HIMP is given by $\mathcal{O}(|\mathcal{E}| + |\mathcal{R}|) = \mathcal{O}(|\mathcal{E}|)$ with $|\mathcal{R}| \ll |\mathcal{E}|$, and is therefore equivalent to performing message passing solely on the molecular graph. This is in high contrast to alternative higher-order variants such as k-dimensional GNNs (k-GNNs) (Morris *et al.*, 2019) with computational complexities of $\mathcal{O}(|\mathcal{V}|^k)$, *cf.* Section 3.4.2. Obtaining junction trees from molecular graphs is done in a pre-processing step, which only leads to a minor and negligible overhead in the training stage.

4.3.4 Evaluation

We evaluate our proposed HIMP architecture on the ZINC dataset (Kusner *et al.*, 2017; Dwivedi *et al.*, 2020), and a subset of datasets stemming from the MoleculeNet and OGB benchmark dataset collections (Wu *et al.*, 2018; Hu *et al.*, 2020a). For all experiments, we make use of the GIN-E operator for learning on the molecular graph (Hu *et al.*, 2020b) and the GIN operator (Xu *et al.*, 2019c) for learning on the associated junction tree (since no edge features are available there). GIN-E is a variant of GIN (Section 3.4.2) that includes edge features (*e.g.*, bond type, bond stereochemistry) by simply adding them to the incoming node features

$$\boldsymbol{h}_v^{(\ell)} = \mathrm{MLP}_{\boldsymbol{\theta}}^{(\ell)}\Big(\big(1+\epsilon^{(\ell)}\big)\cdot \boldsymbol{h}_v^{(\ell-1)} + \sum_{w\in\mathcal{N}(v)} \boldsymbol{h}_w^{(\ell-1)} + \boldsymbol{W}^{(\ell)}\boldsymbol{e}_{w,v}\Big). \tag{4.11}$$

All models were trained with the Adam optimizer (Kingma & Ba, 2015) using a learning rate of 10^{-4}, while other hyperparameters (#epochs, #layers, hidden size, batch size, dropout ratio) are tuned via an additional validation set. Our method is implemented in PyTorch (Paszke *et al.*, 2019) and utilizes the PyTorch Geometric (Fey & Lenssen, 2019) library. The code for reproducing all results is available on GitHub.[2]

4.3.4.1 ZINC Dataset. The ZINC dataset (Kusner *et al.*, 2017) contains about $250,000$ molecular graphs and was introduced in Dwivedi *et al.* (2020) as a benchmark for evaluating GNN performances (using a subset of $10,000$ training graphs). Here, the task is to regress the constrained solubility of a molecule. While this is a fairly simple task that can be exactly computed in a short amount of time, it can nonetheless reveal the capabilities across different neural architectures (Fey *et al.*, 2020b). We compare ourselves to all the baselines presented in Dwivedi *et al.* (2020) and Morris *et al.* (2020b), and additionally report results of a GIN-E baseline that does not make use of any additional junction tree information. Following upon Morris *et al.* (2020b), we also perform experiments on the full dataset.

As shown in Table 4.2, our method is able to significantly outperform all competing methods. In comparison to GIN-E, its best performing competitor, the additional junction tree extension is able to reduce the error rate by about 40–60% (Fey *et al.*, 2020b).

4.3.4.2 MoleculeNet Datasets. Following upon Murphy *et al.* (2019b), we evaluate our model on the HIV, MUV and Tox21 datasets from the MoleculeNet benchmark collection (Wu *et al.*, 2018), using a 80%/10%/10% random split. Here, the task is to predict certain molecular properties (cast as binary labels), *e.g.*, whether a molecule inhibits HIV virus replication or not. We use the ROC-AUC metric to compare ourselves to the neural graph fingerprint (NGF) operator (Duvenaud *et al.*, 2015), and its relational pooling variant RP-NGF (Murphy *et al.*, 2019b), as well as our own GIN-E baseline.

As the results in Table 4.3 indicate, our method advances performance compared to NGF and GIN-E significantly. Although RP-NGF is able to distinguish any graph

[2]Code for HIMP: `https://github.com/rusty1s/himp-gnn` (last access: August 25, 2022)

Method	**Mean Absolute Error (MAE)**	
	ZINC (10k)	ZINC (Full)
GCN (Kipf & Welling, 2017)	0.367±0.011	—
GraphSAGE (Hamilton *et al.*, 2017)	0.398±0.002	—
GIN (Xu *et al.*, 2019c)	0.408±0.008	—
GAT (Veličković *et al.*, 2018)	0.384±0.007	—
MoNet (Monti *et al.*, 2017)	0.292±0.006	—
GatedGCN (Bresson & Laurent, 2017)	0.435±0.011	—
GatedGCN-E (Dwivedi *et al.*, 2020)	0.282±0.015	—
GIN-E (Hu *et al.*, 2020b)	0.252±0.014	0.088±0.002
δ-2-GNN (Morris *et al.*, 2020b)	—	0.042±0.003
δ-2-LGNN (Morris *et al.*, 2020b)	—	0.045±0.006
HIMP	**0.151**±0.006	**0.036**±0.002

Table 4.2: Performance of HIMP on the ZINC datasets (Kusner *et al.*, 2017; Dwivedi *et al.*, 2020)**.** HIMP performs favourable in comparison to traditional GNNs, GNNs that incorporate edge/bond information, as well as alternative higher-order GNN variants (Fey *et al.*, 2020b).

Method	**ROC-AUC** (%)		
	HIV	MUV	Tox21
NGF (Duvenaud *et al.*, 2015)	81.20±1.40	79.80±2.50	79.4±1.00
RP-NGF (Murphy *et al.*, 2019b)	83.20±1.30	79.40±0.50	79.9±0.60
GIN-E (Hu *et al.*, 2020b)	83.83±0.67	79.57±1.14	86.68±0.77
HIMP	**84.81**±0.42	**81.80**±2.02	**87.36**±0.50

Table 4.3: Performance of HIMP on a subset of MoleculeNet datasets (Wu *et al.*, 2018; Fey *et al.*, 2020b)**.**

structure by considering all permutations of nodes (Murphy *et al.*, 2019b), our approach leads to overall better generalization despite its simplicity, and is also orders of magnitude faster to train (Fey *et al.*, 2020b).

4.3.4.3 OGB Datasets. We also test the performance of our model on the datasets `molhiv` and `molpcba` from the OGB benchmark dataset suite (*cf.* Section 5.4), which are adopted from MoleculeNet and enhanced by a more challenging and standardized scaffold splitting procedure (Hu *et al.*, 2020a). We closely follow the experimental protocol of Hu *et al.* (2020b) and report ROC-AUC and PRC-AUC for `molhiv` and `molpcba`, respectively. We compare ourselves to three variants that do not make use of additional junction tree information, namely GCN-E, GatedGCN-E and GIN-E (Kipf & Welling, 2017; Bresson & Laurent, 2017; Dwivedi *et al.*, 2020; Hu *et al.*, 2020a,b).

Results are presented in Table 4.4. As one can see, our approach is able to outperform all its competitors. Interestingly, our model achieves its best results in combination with a small amount of layers (2 or 3), making its runtime and memory requirements on par with the other baselines (which make use of 5 layers). This can be explained

Method	ROC-AUC (%) molhiv	PRC-AUC (%) molpcba
GCN-E (Hu *et al.*, 2020a)	76.07±0.97	19.83±0.16
GatedGCN-E (Dwivedi *et al.*, 2020)	77.65±0.50	20.77±0.27
GIN-E (Hu *et al.*, 2020b)	75.58±1.40	22.17±0.23
HIMP	**78.80**±0.82	**27.39**±0.17

Table 4.4: Performance of HIMP on the `molhiv` and `molpcba` datasets of OGB (Hu *et al.*, 2020a; Fey *et al.*, 2020b).

by the fact that the additional coarse-to-fine information flow enhances the receptive field size of a GNN, and therefore omits the need to stack a multitude of layers (Fey *et al.*, 2020b).

4.4 Graph Matching via Differentiable Neighborhood Consensus

Finally, we identify inherent limitations of the local message passing scheme of GNNs in task that require global information, *e.g.*, in the problem of graph matching, and propose efficient solutions to overcome these shortcomings.

The problem of *graph matching* refers to the task of establishing meaningful *structural correspondences* of nodes between two or more graphs by taking both node similarities and pairwise edge similarities into account (Wang *et al.*, 2019c). Since graphs are natural representations for encoding relational data, the problem of graph matching lies at the heart of many real-world applications. For example, comparing molecules in cheminformatics (Kriege *et al.*, 2019b), matching protein networks in bioinformatics (Sharan & Ideker, 2006; Singh *et al.*, 2008), linking user accounts in social network analysis (Zhang & Philip, 2015), and tracking objects, matching 2D/3D shapes or recognizing actions in computer vision (Vento & Foggia, 2012) can all be formulated as a graph matching problem (Fey *et al.*, 2020a).

Formally, we are given two graphs, a *source graph* $\mathcal{G}_s = (\mathcal{V}_s, \mathcal{E}_s)$ and a *target graph* $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t)$, w.l.o.g. $|\mathcal{V}_s| \leq |\mathcal{V}_t|$, and are interested in finding a *correspondence matrix* $\boldsymbol{S} \in \{0,1\}^{|\mathcal{V}_s| \times |\mathcal{V}_t|}$ which minimizes a given objective while being subject to the *one-to-one mapping constraints* $\sum_{j \in \mathcal{V}_t} S_{i,j} = 1$ for all $i \in \mathcal{V}_s$ and $\sum_{i \in \mathcal{V}_s} S_{i,j} \leq 1$ for all $j \in \mathcal{V}_t$. In particular, we require $\boldsymbol{S}$ to infer an injective mapping $\pi \colon \mathcal{V}_s \to \mathcal{V}_t$ that maps each node in $\mathcal{G}_s$ to its corresponding node in $\mathcal{G}_t$ (Fey *et al.*, 2020a), *cf.* Figure 4.6.

While graph matching has been traditionally tackled via combinatorial optimization, machine learning models, in particular Graph Neural Networks, are a promising approach to solve the graph matching problem in a data-dependent fashion, *cf.* Section 4.4.1. Specifically, real-world graphs such as social networks are often noisy, and therefore exact solutions are often not needed or even desirable. Furthermore, machine learning models are able to adapt to the given data distribution, leading to fast inference time on unseen graphs once the model is fully trained. Lastly, combinatorial approaches often do not consider *continuous* node or edge embeddings that naturally

Figure 4.6: An example solution of establishing meaningful structural correspondences between two graphs. Every node in the source graph $\mathcal{G}_s$ is mapped to exactly one node in the target graph $\mathcal{G}_t$, as denoted by the ■ connections.

arise in real-world applications, while GNN have no problem in incorporating them (Fey *et al.*, 2020a).

However, GNNs are subject to inherent weaknesses solving this task with high precision, as their local message passing formulation limits their applicability to resolve ambiguities in node embeddings, *cf.* Section 4.4.2. Here, we propose a fully-differentiable two-stage neural architecture named *Deep Graph Matching Consensus* (*DGMC*) that circumvents this weakness by sparsely distributing global positional encodings in the two graphs, *cf.* Section 4.4.3. Intuitively, our graph matching procedure is trained in an end-to-end fashion to reach a data-driven *neighborhood consensus* between matched node pairs without the need to solve any optimization problem during inference. In addition, our approach is *purely local*, *i.e.* it operates on fixed-sized neighborhoods around nodes, and is *sparsity-aware*, *i.e.* it takes the sparsity of the underlying structures into account. Hence, our approach scales well to large, real-world input domains while still being able to recover global correspondences consistently (Fey *et al.*, 2020a). In our evaluation (Section 4.4.4), we demonstrate the practical effectiveness of DGMC on the real-world tasks of keypoint matching in computer vision and entity alignment between knowledge graphs.

4.4.1 State-of-the-Art

Identifying correspondences between the nodes in different graphs has been studied in various domains. Closely related problems are summarized under the terms *maximum common subgraph* (Kriege *et al.*, 2019b), *network alignment* (Zhang, 2016), *graph edit distance* (Chen *et al.*, 2019b) and *graph matching* (Yan *et al.*, 2016). Recently, Graph Neural Networks have been used to tackle the task of graph matching in a learnable fashion (Wang *et al.*, 2019c; Zhang & Lee, 2019; Xu *et al.*, 2019d; Derr *et al.*, 2019).

4.4.1.1 Graph Matching via Graph Theory. In graph theory, the combinatorial *maximum common subgraph isomorphism* problem is studied, which asks for the largest graph that is contained as a subgraph in two given graphs. The problem is NP-hard in general and remains so even in trees (Garey & Johnson, 1979) unless the common subgraph is required to be connected (Matula, 1978). Moreover, most variants of the problem are difficult to approximate with theoretical guarantees (Kann, 1992).

Fundamentally different techniques have been developed in bioinformatics and computer vision, where the problem is commonly referred to as *graph matching*. In graph matching, for two graphs with adjacency matrices $\boldsymbol{A}_s$ and $\boldsymbol{A}_t$ of order $|\mathcal{V}|$, the term

$$\|\boldsymbol{A}_s - \boldsymbol{S}^\top \boldsymbol{A}_t \boldsymbol{S}\|_F^2 = \|\boldsymbol{A}_s\|_F^2 + \|\boldsymbol{A}_t\|_F^2 - 2 \sum_{\substack{i,i' \in \mathcal{V}_s \\ j,j' \in \mathcal{V}_t}} A^{(s)}_{i,i'} A^{(t)}_{j,j'} S_{i,j} S_{i',j'} \tag{4.12}$$

is to be minimized, where $\boldsymbol{S} \in \mathcal{P}$ with $\mathcal{P}$ denoting the set of $|\mathcal{V}| \times |\mathcal{V}|$ permutation matrices, and $\|\boldsymbol{A}\|_F^2 = \sum_{i,i' \in \mathcal{V}} A^2_{i,i'}$ denotes the squared Frobenius norm. There is a long line of research trying to minimize Equation (4.12) for $\boldsymbol{S} \in [0,1]^{|\mathcal{V}| \times |\mathcal{V}|}$ by a Frank-Wolfe type algorithm (Jaggi, 2013) and finally projecting the fractional solution to $\mathcal{P}$ (Gold & Rangarajan, 1996; Zaslavskiy *et al.*, 2009; Leordeanu *et al.*, 2009; Egozi *et al.*, 2013; Zhou & De la Torre, 2016). However, the applicability of relaxation and projection is still poorly understood and only few theoretical results exist (Aflalo *et al.*, 2015; Lyzinski *et al.*, 2016). A classical result by Tinhofer (1991) states that the Weisfeiler-Lehman (WL) heuristic distinguishes two graphs $\mathcal{G}_s$ and $\mathcal{G}_t$ if and only if there is no fractional $\boldsymbol{S}$ such that the objective function in Equation (4.12) becomes zero. Kersting *et al.* (2014) showed how the Frank-Wolfe algorithm can be modified to obtain the WL partition. Aflalo *et al.* (2015) proved that the standard relaxation yields a correct solution for a particular class of asymmetric graphs, which can be characterized by the spectral properties of their adjacency matrix. Finally, Bento & Ioannidis (2018) studied various relaxations, their complexity and properties. Other approaches to graph matching exist, *e.g.*, based on spectral relaxations (Umeyama, 1988; Leordeanu & Hebert, 2005) or random walks (Gori *et al.*, 2005a; Fey *et al.*, 2020a).

Furthermore, the problem of graph matching is closely related to the *Quadratic Assignment Problem* (*QAP*) (Zhou & De la Torre, 2016), *e.g.*, Equation (4.12) can be directly interpreted as *Koopmans-Beckmann's QAP*. The more recent literature on graph matching typically considers a weighted version, where node and edge similarities are taken into account. This leads to the formulation as *Lawler's QAP*, which involves an affinity matrix of size $|\mathcal{V}|^2 \times |\mathcal{V}|^2$ and is computational demanding. Zhou & De la Torre (2016) propose to factorize the affinity matrix into smaller matrices and incorporate global geometric constraints. Zhang *et al.* (2019d) studied *kernelized graph matching*, where the node and edge similarities are kernels, which allows to express the graph matching problem again as Koopmans-Beckmann's QAP in the associated Hilbert space. Inspired by established methods for *Maximum-A-Posteriori* (*MAP*) inference in conditional random fields, Swoboda *et al.* (2017) studied several Lagrangean decompositions of the graph matching problem, which are solved by dual ascent algorithms. Recently, *functional representations* for graph matching have been proposed as a generalizing concept with the additional goal to avoid the construction of the affinity matrix (Wang *et al.*, 2019a; Fey *et al.*, 2020a).

The problem of *network alignment* is typically defined analogously to Equation (4.12), where a similarity function between pairs of nodes is given in addition. Most algorithms follow a two step approach: First, an $|\mathcal{V}| \times |\mathcal{V}|$ node-to-node similarity matrix $\boldsymbol{M}$ is computed from the given similarity function and the topology of the two graphs. Then, in the second step, an alignment is computed by solving the assignment problem for $\boldsymbol{M}$. Singh *et al.* (2008) proposed IsoRank, which is based on the adjacency matrix of the product graph $\boldsymbol{K} = \boldsymbol{A}_s \otimes \boldsymbol{A}_t$ of $\mathcal{G}_s$ and $\mathcal{G}_t$, where $\otimes$ denotes the Kronecker product. The matrix $\boldsymbol{M}$ is then obtained by applying PageRank (Page *et al.*,

1999), using a normalized version of $\boldsymbol{K}$ as the Google matrix and the node similarities as the personalization vector. Kollias *et al.* (2012) proposed an efficient approximation of IsoRank by decomposition techniques to avoid generating the product graph of quadratic size. Zhang (2016) present an extension supporting node and edge similarities and propose its computation using non-exact techniques. Klau (2009) proposed to solve network alignment by linearizing the quadratic optimization problem to obtain an integer linear program, which is then approached via Lagrangian relaxation. Bayati *et al.* (2013) developed a message passing algorithm for sparse network alignment, where only a small number of matches between the nodes of the two graphs are allowed (Fey *et al.*, 2020a).

A related concept is the *graph edit distance*, which measures the minimum cost required to transform a graph into another graph by adding, deleting and substituting nodes and edges (Sanfeliu & Fu, 1983). However, its computation is NP-hard, since it generalizes the maximum common subgraph problem (Bunke, 1997). Moreover, it is also closely related to the Quadratic Assignment Problem (Bougleux *et al.*, 2017). Recently, several elaborated exact algorithms for computing the graph edit distance have been proposed (Gouda & Hassaan, 2016; Lerouge *et al.*, 2017; Chen *et al.*, 2019b), but are still limited to small graphs. Therefore, heuristics based on the assignment problem have been proposed (Riesen & Bunke, 2009) and are widely used in practice (Stauffer *et al.*, 2017). The original approach requires cubic running time, which can be reduced to quadratic time using greedy strategies (Riesen *et al.*, 2015a,b), and even linear time for restricted cost functions (Kriege *et al.*, 2019a; Fey *et al.*, 2020a).

The techniques briefly summarized above aim to find an optimal correspondence according to a clearly defined objective function. However, it is often difficult to specify node and edge similarity functions in practical applications. As a result, it has been also proposed to *learn* domain-dependent graph matching models (Fey *et al.*, 2020a).

4.4.1.2 Deep Graph Matching. The problem of graph matching has been recently investigated using deep neural networks as well. For example, Wang *et al.* (2019c) and Zhang & Lee (2019) developed supervised deep graph matching networks based on displacement and combinatorial objectives, respectively. Zanfir & Sminchisescu (2018) model the graph matching affinity via a differentiable, but unlearnable spectral graph matching solver (Leordeanu & Hebert, 2005). Wang *et al.* (2019c) use node-wise features in combination with dense node-to-node cross-graph affinities, distribute them in a local fashion, and adopt sinkhorn normalization for the final task of linear assignment. Zhang & Lee (2019) propose a compositional message passing algorithm that maps point coordinates into a high-dimensional space. The final matching procedure is done by computing the pairwise inner product between point embeddings. However, neither of these approaches can naturally resolve violations of inconsistent neighborhood assignments as we do in our work (Fey *et al.*, 2020a).

Xu *et al.* (2019b) tackles the problem of graph matching by relating it to the Gromov-Wasserstein discrepancy (Peyré *et al.*, 2016). In addition, the optimal transport objective is enhanced by simultaneously learning node embeddings which shall account for the noise in both graphs. In a follow-up work, Xu *et al.* (2019a) extend this concept to the tasks of multi-graph partitioning and matching by learning a Gromov-Wasserstein barycenter. Our approach also resembles the optimal transport between nodes, but works in a supervised fashion for sets of graphs and is therefore able to generalize to unseen graph instances (Fey *et al.*, 2020a).

In addition, the task of network alignment has been recently investigated from multiple perspectives. Derr *et al.* (2019) leverage CycleGANs (Zhu *et al.*, 2017) to align Node2Vec embeddings (Grover & Leskovec, 2016) and find matchings based on the nearest neighbor in the embedding space. Zhang *et al.* (2019b) design a deep graph model based on global and local network topology preservation as auxiliary tasks. Heimann *et al.* (2018) utilize a fast, but purely local and greedy matching procedure based on local node embedding similarity.

Furthermore, Bia *et al.* (2019) use shared Graph Neural Networks to approximate the graph edit distance between two graphs. Here, a (non-differentiable) histogram of correspondence scores is used to fine-tune the output of the network. In a follow-up work, Bai *et al.* (2018) proposed to order the correspondence matrix in a breadth-first-search fashion and process it further with the help of traditional Convolutional Neural Networks (CNNs). Both approaches only operate on local node embeddings, and are hence prone to match correspondences inconsistently (Fey *et al.*, 2020a).

The concept of enhancing intra-graph node embeddings by inter-graph node embeddings has been already heavily investigated in practice (Li *et al.*, 2019b; Wang *et al.*, 2019c; Xu *et al.*, 2019d). Li *et al.* (2019b) and Wang *et al.* (2019c) enhance the GNN operator by not only aggregating information from local neighbors, but also from similar embeddings in the other graph by utilizing a cross-graph matching procedure. Xu *et al.* (2019d) leverage alternating GNNs to propagate local features of one graph throughout the second graph. Wang & Solomon (2019) tackle the problem of finding an unknown rigid motion between point clouds by relating it to a point cloud matching problem followed by a differentiable Singular Value Decomposition (SVD) module. However, neither of these approaches is designed to achieve a consistent matching, due to only operating on localized node embeddings which are alone not sufficient to resolve ambiguities in the matchings. Nonetheless, we argue that these methods can be used to strengthen the initial feature matching procedure, making our approach orthogonal to improvements in this field (Fey *et al.*, 2020a).

Methods to obtain consistency of correspondences in local neighborhoods have a rich history in computer vision, dating back several years (Sattler *et al.*, 2009; Sivic & Zisserman, 2003; Schmid & Mohr, 1997). They are known to heavily improve results of local feature matching procedures. Recently, a deep neural network for neighborhood consensus using 4D convolution was proposed in Rocco *et al.* (2018). However, the 4D convolution can not be efficiently transferred to the graph domain, since it would lead to applying a GNN on the product graph with $\mathcal{O}(|\mathcal{V}|^2)$ nodes and $\mathcal{O}(\mathcal{V}|^4)$ edges (Fey *et al.*, 2020a).

4.4.2 Local Feature Matching

We consider the problem of supervised and semi-supervised matching of graphs (Zanfir & Sminchisescu, 2018; Wang *et al.*, 2018b). Specifically, in the supervised setting, we are given pair-wise ground-truth correspondences for a set of graphs and want our machine learning model to generalize well to unseen graph pairs. In the semi-supervised setting, source and target graphs are fixed, and ground-truth correspondences are only given for a small subset of nodes. However, we are allowed to make use of the complete graph structures and all the features attached to their nodes and edges (Fey *et al.*, 2020a).

In the following, we describe our proposed end-to-end deep graph matching architecture in detail. Our method consists of two stages: a *local feature matching procedure* followed by an *iterative refinement strategy* using synchronous message passing networks. The aim of the feature matching step is to compute initial correspondence scores based on the similarity of local node embeddings. The iterative refinement strategy then takes these initial correspondences as input and iteratively refines them by aiming to reach a neighborhood consensus for correspondences using a differentiable validator for graph isomorphism. Here, we first describe the local feature matching procedure before introducing our novel iterative refinement strategy in Section 4.4.3.

Our local feature matching procedure is modelled in close analogy to related approaches (Bai *et al.*, 2018; Bia *et al.*, 2019; Wang *et al.*, 2019c; Zhang & Lee, 2019; Wang & Solomon, 2019) by computing similarities between nodes in the source graph $\mathcal{G}_s$ and the target graph $\mathcal{G}_t$ based on the node embeddings obtained by a GNN (Fey *et al.*, 2020a). That is, we first obtain latent node embeddings $\boldsymbol{h}_i^{(s)} = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i^{(s)})$ for $i \in \mathcal{V}_s$ and $\boldsymbol{h}_j^{(t)} = f_{\boldsymbol{\theta}}(\boldsymbol{x}_j^{(t)})$ for $j \in \mathcal{V}_t$ for source and target graphs $\mathcal{G}_s$ and $\mathcal{G}_t$, respectively, using a *siamese* Graph Neural Network $f_{\boldsymbol{\theta}}$ with shared weights (Fey *et al.*, 2020a). Here, $\boldsymbol{x}_i^{(s)}$ represents the initially given node features of node $i \in \mathcal{V}_s$ in the source graph, while $\boldsymbol{x}_j^{(t)}$ denotes the initially given node features of node $j \in \mathcal{V}_t$ in the target graph. Given those latent node embeddings, we obtain *soft* correspondences as

$$\boldsymbol{S} = \mathrm{sinkhorn}(\hat{\boldsymbol{S}}) \in [0,1]^{|\mathcal{V}_s| \times |\mathcal{V}_t|} \quad \text{with} \quad \hat{S}_{i,j} = \boldsymbol{h}_i^{(s)} \boldsymbol{h}_j^{(t)} \tag{4.13}$$

based on the dot-product between source and target node embeddings (Fey *et al.*, 2020a). Here, $\mathrm{sinkhorn}$ normalization is applied to obtain *rectangular doubly-stochastic* correspondence matrices that fulfill the constraints $\sum_{j \in \mathcal{V}_t} S_{i,j} = 1$ for all $i \in \mathcal{V}_s$ and $\sum_{i \in \mathcal{V}_s} S_{i,j} \leq 1$ for all $j \in \mathcal{V}_t$ (Sinkhorn & Knopp, 1967; Adams & Zemel, 2011; Cour *et al.*, 2006). Formally, the $\mathrm{sinkhorn}$ operator applies row-wise and column-wise softmax normalization iteratively until convergence, *i.e.*

$$S_{i,j} \leftarrow \frac{\exp(S_{i,j})}{\sum_{k=1}^{|\mathcal{V}_t|} \exp(S_{i,k})} \quad \text{and} \quad S_{i,j} \leftarrow \frac{\exp(S_{i,j})}{\sum_{k=1}^{|\mathcal{V}_s|} \exp(S_{k,j})}, \tag{4.14}$$

and is therefore well differentiable. However, due to its iterative nature, it quickly runs the risk of vanishing gradients $\partial \boldsymbol{S} / \partial \hat{\boldsymbol{S}}$ (Zhang *et al.*, 2019c), which we take a closer look into in Section 4.4.3.

We can interpret the i-th row vector $\boldsymbol{S}_{i,:} \in [0,1]^{|\mathcal{V}_t|}$ as a discrete distribution over potential correspondences in $\mathcal{G}_t$ for each node $i \in \mathcal{V}_s$. As such, we can train our GNN $f_{\boldsymbol{\theta}}$ in a discriminative, supervised fashion against ground truth correspondences $\pi_{\mathrm{gt}}(\cdot)$ by minimizing the negative log-likelihood of correct correspondence scores

$$\mathcal{L} = -\sum_{i \in \mathcal{V}_s} \log(S^{(0)}_{i, \pi_{\mathrm{gt}}(i)}). \tag{4.15}$$

By utilizing a GNN $f_{\boldsymbol{\theta}}$ for feature encoding, we obtain localized and permutation equivariant vectorial node representations that encode both feature information and structural properties of the local neighborhood around each node, *cf.* Chapter 3. As a result, obtained correspondence scores denote the similarity of local neighborhoods around nodes in the source and target graph. In addition, the wide range of available GNN operators (Kipf & Welling, 2017; Gilmer *et al.*, 2017; Veličković *et al.*, 2018;

Schlichtkrull *et al.*, 2018; Xu *et al.*, 2019c) makes such local feature matching procedure very flexible, as we have precise control over the properties of extracted features (Fey *et al.*, 2020a), *cf.* Section 3.2. However, due to the purely local nature of the node embeddings, such local feature matching procedure alone cannot resolve any ambiguities in matching, and is therefore prone to find false correspondences which are locally similar but spatially different to the correct one (Fey *et al.*, 2020a).

4.4.3 Iterative Message Passing for Neighborhood Consensus

Since the local feature matching stage is not sufficient to accurately obtain consistent correspondences, the DGMC approach utilizes a *second stage* to refine its initial matchings. Our key idea to overcome the aforementioned weaknesses of local feature matching is to utilize a second message passing network that has the capability to resolve ambiguities and false matchings made in the first phase. In particular, we propose to inject an inductive bias into our model that aims to reach a neighborhood consensus for predicted matches. Such intuition has a long history in graph matching theory (Anstreicher, 2003; Gold & Rangarajan, 1996; Caetano *et al.*, 2009; Cho *et al.*, 2013). For example, solving the graph matching problem has been originally formulated as an edge-preserving, quadratic assignment problem, *i.e.*

$$\operatorname*{argmax}_{\boldsymbol{S}} \sum_{\substack{i,i' \in \mathcal{V}_s \\ j,j' \in \mathcal{V}_t}} A^{(s)}_{i,i'} A^{(t)}_{j,j'} S_{i,j} S_{i',j'} \tag{4.16}$$

where $\boldsymbol{A}^{(s)}$ and $\boldsymbol{A}^{(t)}$ denote the adjacency matrices of $\mathcal{G}_s$ and $\mathcal{G}_t$, respectively, and $\boldsymbol{S}$ is subject to one-to-one mapping constraints. This formulation is based on the intuition of finding correspondences based on *neighborhood consensus* (Rocco *et al.*, 2018), which shall prevent adjacent nodes in the source graph from being mapped to different regions in the target graph. Formally, a neighborhood consensus is reached if, for all node pairs $(i, j) \in \mathcal{V}_s \times \mathcal{V}_t$ with $S_{i,j} = 1$, it holds that for every node $i' \in \mathcal{N}(i)$ there exists a node $j' \in \mathcal{N}(j)$ such that $S_{i',j'} = 1$ as well (Fey *et al.*, 2020a).

Our local feature matching procedure (Section 4.4.2) has no way to detect such violations of the neighborhood consensus criteria employed in Equation (4.16). Since finding a global optimum is NP-hard, we aim to detect violations of the criteria in local neighborhoods and resolve them in an iterative fashion (Fey *et al.*, 2020a). In particular, we utilize Graph Neural Networks to detect such violations in a neighborhood consensus step and iteratively refine correspondences $\boldsymbol{S}^{(\ell)}$, $\ell \in \{0, \ldots, L\}$, starting from the initial soft correspondences $\boldsymbol{S}^{(0)}$ given by our local feature matching procedure. Key to the proposed algorithm is the following observation: The soft correspondence matrix $\boldsymbol{S} \in [0,1]^{|\mathcal{V}_s| \times |\mathcal{V}_t|}$ is a map from the node function space $L(\mathcal{G}_s) = L(\mathbb{R}^{|\mathcal{V}_s|})$ to the node function space $L(\mathcal{G}_t) = L(\mathbb{R}^{|\mathcal{V}_t|})$. Therefore, we can use $\boldsymbol{S}$ to pass node functions $\boldsymbol{x}^{(s)} \in L(\mathcal{G}_s)$, $\boldsymbol{x}^{(t)} \in L(\mathcal{G}_t)$ along the soft correspondences by

$$\hat{\boldsymbol{x}}^{(t)} = \boldsymbol{S}^{\top} \boldsymbol{x}^{(s)} \quad \text{and} \quad \hat{\boldsymbol{x}}^{(s)} = \boldsymbol{S} \boldsymbol{x}^{(t)} \tag{4.17}$$

to obtain functions $\hat{\boldsymbol{x}}^{(t)} \in L(\mathcal{G}_t)$, $\hat{\boldsymbol{x}}^{(s)} \in L(\mathcal{G}_s)$ in the other domain, respectively (Fey *et al.*, 2020a).

Then, our consensus method works as follows and is further illustrated as part of the two-stage architecture of DGMC in Figure 4.7: We first generate unique node colorings $\boldsymbol{c}^{(s)}_i$ for every node $i \in \mathcal{V}_s$ in the source graph, *e.g.*, given by $\mathcal{V}_s \rightarrow \{0,1\}^{|\mathcal{V}_s|}$ in the

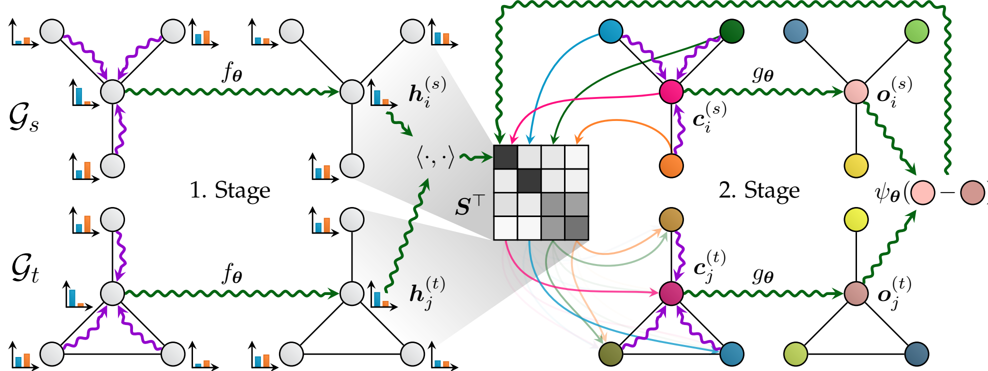


Figure 4.7: High-level illustration of our two-stage neighborhood consensus architecture. Node features $\boldsymbol{h}_i^{(s)}$ and $\boldsymbol{h}_j^{(t)}$ obtained from a GNN $f_{\boldsymbol{\theta}}$ are first locally matched based on their dot-product, before their correspondence scores get iteratively refined based on neighborhood consensus. Here, injective node colorings $\boldsymbol{c}_i^{(s)}$ on $\mathcal{G}_s$ are transferred to $\mathcal{G}_t$ via $\boldsymbol{S}$, and distributed by a second GNN $g_{\boldsymbol{\theta}}$ on both graphs to obtain individual outputs $\boldsymbol{o}_i^{(s)}$ and $\boldsymbol{o}_j^{(t)}$, respectively. Updates on $\boldsymbol{S}$ are performed by a neural network $\psi_{\boldsymbol{\theta}}$ based on the pair-wise color differences $\boldsymbol{o}_i^{(s)} - \boldsymbol{o}_j^{(t)}$ (Fey *et al.*, 2020a).

form of an identity matrix $\boldsymbol{I}_{|\mathcal{V}_s|}$. Using $\boldsymbol{S}^{(\ell)}$, we then map those unique node colorings from the source graph $\mathcal{G}_s$ to the target graph $\mathcal{G}_t$. Then, we distribute these colorings in corresponding neighborhoods by performing synchronous message passing on both graphs via a second shared Graph Neural Network $g_{\boldsymbol{\theta}}$ (Fey *et al.*, 2020a), *i.e.*

$$\boldsymbol{o}_i^{(s)} = g_{\boldsymbol{\theta}}\left(\boldsymbol{c}_i^{(s)}\right) \quad \text{and} \quad \boldsymbol{o}_j^{(t)} = g_{\boldsymbol{\theta}}\left(\left(\boldsymbol{S}_{(\ell)}^\top \boldsymbol{C}^{(s)}\right)_j\right). \tag{4.18}$$

We can then compare the results of both GNNs to recover a vector $\boldsymbol{d}_{i,j} = \boldsymbol{o}_i^{(s)} - \boldsymbol{o}_j^{(t)}$ which measures the neighborhood consensus between node pairs $(i,j) \in \mathcal{V}_s \times \mathcal{V}_t$. This measure can be used to perform trainable updates of the correspondence scores

$$S_{i,j}^{(\ell+1)} = \operatorname{sinkhorn}(\hat{\boldsymbol{S}}^{(\ell+1)})_{i,j} \quad \text{with} \quad \hat{S}_{i,j}^{(\ell+1)} = \hat{S}_{i,j}^{(\ell)} + \psi_{\boldsymbol{\theta}}(\boldsymbol{d}_{j,i}) \tag{4.19}$$

based on an Multi-Layer Perceptron (MLP) $\psi_{\boldsymbol{\theta}}$. The process can be applied L times to iteratively improve the consensus in neighborhoods (Fey *et al.*, 2020a). The final objective

$$\mathcal{L} = -\sum_{i \in \mathcal{V}_s} \underbrace{\log(S_{i,\pi_{\text{gt}}(i)}^{(0)})}_{\mathcal{L}^{(\text{initial})}} + \underbrace{\log(S_{i,\pi_{\text{gt}}(i)}^{(L)})}_{\mathcal{L}^{(\text{refined})}} \tag{4.20}$$

combines both the feature matching error introduced in Section 4.4.2 and the newly introduced neighborhood consensus error (Fey *et al.*, 2020a). Notably, this objective is fully-differentiable and can hence be optimized in an end-to-end fashion using stochastic gradient descent. Overall, the consensus stage distributes global node colorings to resolve ambiguities and false matchings made in the first stage of our architecture by only using purely local operators. Since an initial matching is needed to test for neighborhood consensus, this task cannot be fulfilled by $g_{\boldsymbol{\theta}}$ alone, which stresses the importance of our two-stage approach (Fey *et al.*, 2020a).

4.4.3.1 Theoretical Guarantees. The following two theorems show that the difference in color distributions $\boldsymbol{d}_{i,j}$ is a good measure of how well local neighborhoods around $i \in \mathcal{V}_s$ and $j \in \mathcal{V}_t$ are matched by the soft correspondence between $\mathcal{G}_s$ and $\mathcal{G}_t$.

Theorem 3. *Let $\mathcal{G}_s$ and $\mathcal{G}_t$ be two isomorphic graphs and let $g_{\boldsymbol{\theta}}$ be a permutation equivariant GNN. If $\boldsymbol{S} \in \{0,1\}^{|\mathcal{V}_s| \times |\mathcal{V}_t|}$ encodes an isomorphism between $\mathcal{G}_s$ and $\mathcal{G}_t$, then $\boldsymbol{d}_{i,\pi(i)} = \mathbf{0}$ for all $i \in \mathcal{V}_s$.*

Proof. Since $g_{\boldsymbol{\theta}}$ is permutation equivariant, for any node feature matrix $\boldsymbol{X}_s \in \mathbb{R}^{|\mathcal{V}_s| \times \cdot}$, it holds that $g_{\boldsymbol{\theta}}(\boldsymbol{S}^\top \boldsymbol{X}_s, \boldsymbol{S}^\top \boldsymbol{A}_s \boldsymbol{S}) = \boldsymbol{S}^\top g_{\boldsymbol{\theta}}(\boldsymbol{X}_s, \boldsymbol{A}_s)$. Since $\boldsymbol{S}$ encodes an isomorphism between $\mathcal{G}_s$ and $\mathcal{G}_t$, it immediately follows that

$$\boldsymbol{O}_t = g_{\boldsymbol{\theta}}(\boldsymbol{X}_t, \boldsymbol{A}_t) = g_{\boldsymbol{\theta}}(\boldsymbol{S}^\top \boldsymbol{X}_s, \boldsymbol{S}^\top \boldsymbol{A}_s \boldsymbol{S}) = \boldsymbol{S}^\top g_{\boldsymbol{\theta}}(\boldsymbol{X}_s, \boldsymbol{A}_s) = \boldsymbol{S}^\top \boldsymbol{O}_s.$$

Hence, $\boldsymbol{o}_i^{(s)} = \boldsymbol{o}_{\pi(i)}^{(t)}$ for any node $i \in \mathcal{V}_s$, resulting in $\boldsymbol{d}_{i,\pi(i)} = \mathbf{0}$. □

Theorem 4. *Let $\mathcal{G}_s$ and $\mathcal{G}_t$ be two graphs and let $g_{\boldsymbol{\theta}}$ be a permutation equivariant, maximally expressive L-layer GNN. Let $\mathcal{N}_L(i)$ describe the L-hop neighborhood around node i. If $\boldsymbol{d}_{i,j} = \mathbf{0}$, then $\boldsymbol{S}_{\mathcal{N}_L(i),\mathcal{N}_L(j)} \in [0,1]^{|\mathcal{N}_L(i)| \times |\mathcal{N}_L(j)|}$ is a permutation matrix describing an isomorphism between the L-hop subgraph $\mathcal{G}_s[\mathcal{N}_L(i)]$ around $i \in \mathcal{V}_s$ and the L-hop subgraph $\mathcal{G}_t[\mathcal{N}_L(j)]$ around $j \in \mathcal{V}_t$.*

Proof. Since $\boldsymbol{o}_i^{(s)} = \boldsymbol{o}_j^{(t)}$, the L-layer GNN $g_{\boldsymbol{\theta}}$ has mapped both L-hop neighborhoods around node $i \in \mathcal{V}_s$ and node $j \in \mathcal{V}_t$ to the same vectorial representation:

$$\boldsymbol{o}_i^{(s)} = g_{\boldsymbol{\theta}}\left(\boldsymbol{c}_i^{(s)}\right) = g_{\boldsymbol{\theta}}\left(\left(\boldsymbol{S}^\top \boldsymbol{C}^{(s)}\right)_j\right) = \boldsymbol{o}_j^{(t)}. \tag{4.21}$$

Since $g_{\boldsymbol{\theta}}$ is maximally expressive, *i.e.* it is as powerful as the WL heuristic in distinguishing graph structures (Xu *et al.*, 2019c; Morris *et al.*, 2019), and is operating on injective node colorings, it has the power to distinguish *any* graph structure, *cf.* Murphy *et al.* (2019b). Hence, $\boldsymbol{S} \in [0,1]^{|\mathcal{N}_L(i)| \times |\mathcal{N}_L(j)|}$ needs to be a permutation matrix describing an isomorphism between $\mathcal{G}_s[\mathcal{N}_L(i)]$ and $\mathcal{G}_t[\mathcal{N}_L(j)]$. □

A GNN $g_{\boldsymbol{\theta}}$ that satisfies both criteria in Theorem 3 and Theorem 4 provides equal node embeddings $\boldsymbol{o}_i^{(s)}$ and $\boldsymbol{o}_j^{(t)}$ if and only if nodes in a local neighborhood are correctly matched to each other. A value $\boldsymbol{d}_{i,j} \neq \mathbf{0}$ indicates the existence of inconsistent matchings in the local neighborhoods around i and j, and can hence be used to refine the correspondence score $\hat{S}_{i,j}$ (Fey *et al.*, 2020a).

Note that both requirements, permutation equivariance and injectivity, are easily fulfilled: **(1)** All common Graph Neural Network architectures following the message passing scheme of Equation (3.5) are equivariant due to the use of permutation invariant neighborhood aggregators. **(2)** There provenly exists maximally expressive GNN architectures that are as powerful as the WL heuristic (Weisfeiler & Lehman, 1968) in distinguishing graph structures, *e.g.*, by using sum aggregation in combination with MLPs on the multiset of neighboring node features (Xu *et al.*, 2019c; Morris *et al.*, 2019), *cf.* Section 3.4.

While the expressive power and limitations of the WL heuristic are well understood (Arvind *et al.*, 2015), it is important to note that our DGMC architecture generally

inherits its capabilities and limitations. Hence, one possible limitation of our approach is that whenever two nodes are assigned the same color by WL, our approach may fail to converge to one of the possible solutions, *e.g.*, in case there exists two nodes $i, j \in \mathcal{V}_t$ with equal neighborhood sets $\mathcal{N}(i) = \mathcal{N}(j)$. In this case, the initial feature matching procedure will generate equal initial correspondence distributions $\boldsymbol{S}^{(0)}_{:,i} = \boldsymbol{S}^{(0)}_{:,j}$, leading to both nodes receiving the same color distributions $\boldsymbol{c}^{(t)}_i = \boldsymbol{c}^{(t)}_j$ from $\mathcal{G}_s$. Since both nodes share the same neighborhood, $g_{\boldsymbol{\theta}}$ also produces the same distributed functions $\boldsymbol{o}^{(t)}_i = \boldsymbol{o}^{(t)}_j$. As a result, both column vectors $\hat{\boldsymbol{S}}^{(\ell)}_{:,i}$ and $\hat{\boldsymbol{S}}^{(\ell)}_{:,j}$ receive the same update, leading to non-convergence. In theory, one might resolve these ambiguities by adding a small amount of noise to $\hat{\boldsymbol{S}}^{(0)}$, which helps the model to randomly decide between one of the two options. However, the general amount of feature noise present in real-world datasets already ensures that this scenario is unlikely to occur (Fey *et al.*, 2020a).

4.4.3.2 Relation to the Graduated Assignment Algorithm. Theoretically, we can relate our proposed approach to classical graph matching techniques that consider a doubly-stochastic relaxation of the problem defined in Equation (4.16), *cf.* Lyzinski *et al.* (2016) and Section 4.4.1. A seminal work following this method is the *graduated assignment algorithm* (Gold & Rangarajan, 1996). By starting from an initial feasible solution $\boldsymbol{S}^{(0)}$, a new solution $\boldsymbol{S}^{(l+1)}$ is iteratively computed from $\boldsymbol{S}^{(l)}$ by approximately solving a linear assignment problem according to

$$\boldsymbol{S}^{(\ell+1)} \leftarrow \operatorname*{softassign}_{\boldsymbol{S}} \sum_{i \in \mathcal{V}_s} \sum_{j \in \mathcal{V}_t} Q_{i,j} S^{\ell}_{i,j} \quad \text{with} \quad Q_{i,j} = 2 \sum_{i' \in \mathcal{V}_s} \sum_{j' \in \mathcal{V}_t} A^{(s)}_{i,i'} A^{(t)}_{j,j'} S^{(\ell)}_{i',j'} \tag{4.22}$$

where $\boldsymbol{Q}$ denotes the gradient of Equation (4.16) at $\boldsymbol{S}^{(\ell)}$ (Fey *et al.*, 2020a).[3] Here, the softassign operator is implemented by applying sinkhorn normalization on rescaled inputs, where the scaling factor grows in every iteration to increasingly encourage integer solutions. Our approach also resembles the approximation of the linear assignment problem via sinkhorn normalization (Fey *et al.*, 2020a).

Moreover, the gradient $\boldsymbol{Q}$ is closely related to our neighborhood consensus scheme for the particular simple, non-trainable GNN instantiation $g(\boldsymbol{X}, \boldsymbol{A}) = \boldsymbol{A}^{\top}\boldsymbol{X}$. Given $\boldsymbol{O}_s = \boldsymbol{A}_s^{\top}\boldsymbol{I}_{|\mathcal{V}_s|} = \boldsymbol{A}_s^{\top}$ and $\boldsymbol{O}_t = \boldsymbol{A}_t^{\top}\boldsymbol{S}^{\top}\boldsymbol{I}_{|\mathcal{V}_s|} = \boldsymbol{A}_t^{\top}\boldsymbol{S}^{\top}$, we obtain $\boldsymbol{Q} = 2\,\boldsymbol{O}_s\boldsymbol{O}_t^{\top}$ by substitution. However, instead of updating $\boldsymbol{S}^{(l)}$ based on the similarity between $\boldsymbol{O}_s$ and $\boldsymbol{O}_t$ obtained from a fixed-function GNN g, we choose to update correspondence scores via trainable neural networks $g_{\boldsymbol{\theta}}$ and $\psi_{\boldsymbol{\theta}}$ based on the difference between $\boldsymbol{o}^{(s)}_i$ and $\boldsymbol{o}^{(t)}_j$. This allows us to interpret our model as a deep parameterized generalization of the graduated assignment algorithm (Fey *et al.*, 2020a). In addition, specifying node and edge attribute similarities in graph matching is often difficult and complicates its computation (Zhou & De la Torre, 2016; Zhang *et al.*, 2019d), whereas our approach naturally supports continuous node and edge features via established GNN models.

[3] For clarity of presentation, we closely follow the original formulation of the method for simple graphs but ignore the edge similarities and adapt the constant factor of the gradient according to our objective function.

4.4.3.3 Scaling to Large Input. We further propose a number of optimizations to let DGMC scale to large input domains (Fey *et al.*, 2020a):

- In order to scale to larger input, we propose to **sparsify initial correspondences** by filtering out low score correspondences before neighborhood consensus takes place. That is, we sparsify $\boldsymbol{S}^{(0)}$ by computing top k correspondences with the help of the KeOps library (Feydy *et al.*, 2020) without ever storing its dense version, reducing its required memory footprint from $\mathcal{O}(|\mathcal{V}_s||\mathcal{V}_t|)$ to $\mathcal{O}(k|\mathcal{V}_s|)$ (Fey *et al.*, 2020a). In addition, the time complexity of the refinement phase is reduced from $\mathcal{O}(|\mathcal{V}_s||\mathcal{V}_t| + |\mathcal{E}_s| + |\mathcal{E}_t|)$ to $\mathcal{O}(k|\mathcal{V}_s| + |\mathcal{E}_s| + |\mathcal{E}_t|)$, where $|\mathcal{E}_s|$ and $|\mathcal{E}_t|$ denote the number of edges in $\mathcal{G}_s$ and $\mathcal{G}_t$, respectively. Note that sparsifying initial correspondences assumes that the feature matching procedure ranks the correct correspondence within the top k elements for each node $i \in \mathcal{V}_s$. Hence, also optimizing the initial feature matching loss is crucial, and can be further accelerated by training only against sparsified correspondences with ground-truth entries $\mathrm{top}_k(\boldsymbol{S}^{(0)}_{i,:}) \cup \{S^{(0)}_{i,\pi_{\mathrm{gt}}(i)}\}$ (Fey *et al.*, 2020a).

- Although applying the GNN $g_{\boldsymbol{\theta}}$ on unique node colorings is computationally efficient due to their sparse nature, it nonetheless requires a parameter complexity of $\mathcal{O}(|\mathcal{V}_s|)$. Hence, we propose to replace unique node colorings with randomly drawn node functions $\boldsymbol{r}^{(s)}_i \sim \mathcal{N}(0,1)$, where $\boldsymbol{r}^{(s)}_i \in \mathbb{R}^R$ with $R \ll |\mathcal{V}_s|$. By sampling from a continuous distribution, these node colorings are still guaranteed to be injective (DeGroot & Schervish, 2012). Note that Theorem 3 still holds because it does not impose any restrictions on the function space $L(\mathcal{G}_s)$. However, Theorem 4 does not necessarily hold anymore, but we expect our refinement strategy to resolve any ambiguities by re-sampling $\boldsymbol{r}^{(s)}_i$ in every iteration ℓ (Fey *et al.*, 2020a).

- The $\mathrm{sinkhorn}$ normalization fulfills the requirements of obtaining rectangular doubly-stochastic solutions (Sinkhorn & Knopp, 1967; Adams & Zemel, 2011; Cour *et al.*, 2006). However, it may eventually push correspondences to inconsistent integer solutions very early on from which the neighborhood consensus method cannot effectively recover. Furthermore, it is inherently inefficient to compute and runs the risk of vanishing gradients $\partial \boldsymbol{S}^{(\ell)} / \partial \hat{\boldsymbol{S}}^{(\ell)}$ (Zhang *et al.*, 2019c). Here, we propose to relax this constraint by only applying row-wise softmax normalization on $\hat{\boldsymbol{S}}^{(\ell)}$, and expect our supervised refinement procedure to naturally resolve violations of $\sum_{i \in \mathcal{V}_s} S^{(\ell)}_{i,j} \leq 1$ on its own by re-ranking false correspondences via neighborhood consensus (Fey *et al.*, 2020a).

- Instead of holding the number of refinement iterations L fixed, we propose to differ the number of iterations $L^{(\text{train})}$ and $L^{(\text{test})}$, $L^{(\text{train})} \ll L^{(\text{test})}$, for training and testing, respectively. This does not only speed up training runtime, but it also encourages the refinement procedure to reach convergence with as few steps as necessary while we can run the refinement procedure until convergence during testing (Fey *et al.*, 2020a).

The final algorithm of DGMC is given in Algorithm 5. Specifically, DGMC first computes initial correspondences based on local feature matching (line 1 — line 3), and picks the top k correspondences for further refinement (line 4). Iteratively, DGMC

Algorithm 5 Optimized Deep Graph Matching Consensus Algorithm

Require: Graphs $\mathcal{G}_s = (\mathcal{V}_s, \mathcal{E}_s)$, $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t)$, Node features $\boldsymbol{X}_s$, $\boldsymbol{X}_t$

1: $\boldsymbol{H}_s \leftarrow f_{\boldsymbol{\theta}}(\boldsymbol{X}_s, \boldsymbol{A}_s)$ — Compute node embeddings $\boldsymbol{H}_s \in \mathbb{R}^{|\mathcal{V}_s| \times \cdot}$
2: $\boldsymbol{H}_t \leftarrow f_{\boldsymbol{\theta}}(\boldsymbol{X}_t, \boldsymbol{A}_t)$ — Compute node embeddings $\boldsymbol{H}_t \in \mathbb{R}^{|\mathcal{V}_t| \times \cdot}$
3: $\hat{\boldsymbol{S}}^{(0)} \leftarrow \boldsymbol{H}_s \boldsymbol{H}_t^{\top}$ — Local feature matching
4: $\hat{\boldsymbol{S}}^{(0)}_{i,:} \leftarrow \mathrm{top}_k(\hat{\boldsymbol{S}}^{(0)}_{i,:})$ — Sparsify to top k candidates for all $i \in \mathcal{V}_s$
5: **for** ℓ in $\{1, \ldots, L\}$ **do**
6: $\quad \boldsymbol{S}^{(\ell-1)}_{i,:} \leftarrow \mathrm{softmax}(\hat{\boldsymbol{S}}^{(\ell-1)}_{i,:})$ — Normalize scores for all $i \in \mathcal{V}_s$
7: $\quad \boldsymbol{R}_s \sim \mathcal{N}(0, 1)$ — Sample random node colorings $\boldsymbol{R}_s \in \mathbb{R}^{|\mathcal{V}_s| \times R}$
8: $\quad \boldsymbol{R}_t \leftarrow \boldsymbol{S}^{\top} \boldsymbol{R}_s$ — Map random node colorings $\boldsymbol{R}_s$ from $\mathcal{G}_s$ to $\mathcal{G}_t$
9: $\quad \boldsymbol{O}_s \leftarrow g_{\boldsymbol{\theta}}(\boldsymbol{R}_s, \boldsymbol{A}_s)$ — Distribute colorings $\boldsymbol{R}_s$ on $\mathcal{G}_s$
10: $\quad \boldsymbol{O}_t \leftarrow g_{\boldsymbol{\theta}}(\boldsymbol{R}_t, \boldsymbol{A}_t)$ — Distribute colorings $\boldsymbol{R}_t$ on $\mathcal{G}_t$
11: $\quad \boldsymbol{d}_{i,j} \leftarrow \boldsymbol{o}^{(s)}_i - \boldsymbol{o}^{(t)}_j$ — Compute neighborhood consensus measure
12: $\quad \hat{S}^{(\ell)}_{i,j} \leftarrow \hat{S}^{(\ell-1)}_{i,j} + \psi_{\boldsymbol{\theta}}(\boldsymbol{d}_{i,j})$ — Perform trainable correspondence update
13: **end for**
14: $\boldsymbol{S}^{(L)}_{i,:} \leftarrow \mathrm{softmax}(\hat{\boldsymbol{S}}^{(L)}_{i,:})$ — Normalize scores for all $i \in \mathcal{V}_s$

samples unique node colorings on the source graph from a continuous distribution (line 7), maps them to the target graph (line 8), and performs updates to the correspondence matrix by measuring the neighborhood consensus error given by the distributed colorings in both graphs (line 9 — line 12).

4.4.4 Evaluation

We verify the performance of our DGMC model on three different tasks. We first show the benefits of our approach in various ablation studies on synthetic graphs (Section 4.4.4.1), and apply it to the real-world tasks of supervised keypoint matching (Section 4.4.4.2) and semi-supervised cross-lingual knowledge graph alignment (Section 4.4.4.3) afterwards.

Our method is implemented in PyTorch (Paszke *et al.*, 2019) and utilizes the PyTorch Geometric (Fey & Lenssen, 2019) and the KeOps (Feydy *et al.*, 2020) libraries. Our implementation can process sparse mini-batches with parallel GPU acceleration and minimal memory footprint in all algorithm steps. The code is publicly available on GitHub[4].

For all experiments, optimization is done via the Adam optimizer (Kingma & Ba, 2015) with a fixed learning rate of 10^{-3}. We use similar architectures for initial and refinement GNN architectures $f_{\boldsymbol{\theta}}$ and $g_{\boldsymbol{\theta}}$, respectively, except that we omit dropout (Srivastava *et al.*, 2014) in the latter. For all experiments, we report Hits@k to evaluate and compare our model to previous lines of work, where Hits@k measures the proportion of correctly matched entities ranked in the top k (Fey *et al.*, 2020a).

4.4.4.1 Ablation Studies on Synthetic Graphs. In our first experiment, we evaluate our method on synthetic graphs where we aim to learn a matching for pairs of

[4] `https://github.com/rusty1s/deep-graph-matching-consensus` (last access: August 25, 2022)

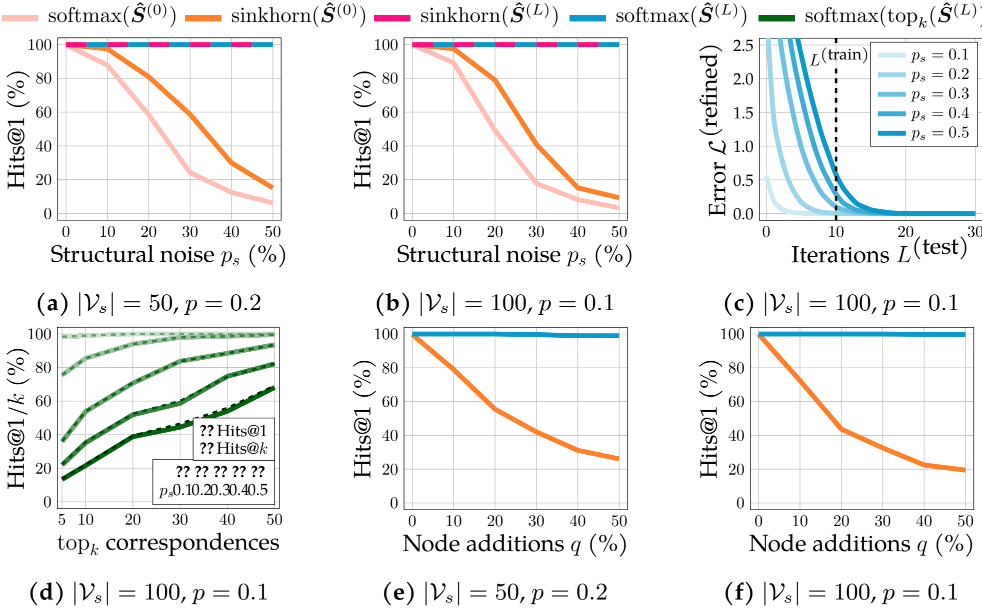


Figure 4.8: The performance of our method on synthetic data with structural noise. While a purely local matching approach fails to find the correct structural correspondences with increasing noise p_s, DGMC recovers *all* correspondences consistently. Furthermore, DGMC is robust towards both structural noise induced in the edge- and node-level (Fey *et al.*, 2020a).

graphs in a supervised fashion. Each pair of graphs consists of an undirected Erdős-Rényi graph $\mathcal{G}_s$ (Erdős & Rényi, 1959) with $|\mathcal{V}_s| \in \{50, 100\}$ nodes and edge probability $p \in \{0.1, 0.2\}$, and a target graph $\mathcal{G}_t$ which is constructed from $\mathcal{G}_s$ by removing edges with probability p_s without disconnecting any nodes (Heimann *et al.*, 2018). Training and evaluation is done on 1 000 graphs each for different configurations $p_s \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}$ (Fey *et al.*, 2020a).

We implement the Graph Neural Network operators $f_{\boldsymbol{\theta}}$ and $g_{\boldsymbol{\theta}}$ by stacking three layers of the GIN operator (Xu *et al.*, 2019c)

$$\boldsymbol{h}_v^{(\ell+1)} = \mathrm{MLP}^{(\ell+1)}\left(\left(1 + \epsilon^{(t+1)}\right) \cdot \boldsymbol{h}_v^{(\ell)} + \sum_{w \in \mathcal{N}(v)} \boldsymbol{h}_w^{(\ell)}\right) \tag{4.23}$$

due to its expressiveness in distinguishing raw graph structures, *cf.* Section 3.4. The number of layers and hidden dimensionality of all MLPs is set to 2 and 32, respectively, and we apply ReLU activation (Glorot *et al.*, 2011) and Batch Normalization (Ioffe & Szegedy, 2015) after each of its layers. Input features are initialized with one-hot encodings of node degrees. We employ a Jumping Knowledge style concatenation $\boldsymbol{h}_v = \boldsymbol{W}[\boldsymbol{h}_v^{(1)}, \ldots, \boldsymbol{h}_v^{(L)}]$ (Xu *et al.*, 2018) to compute final node representations $\boldsymbol{h}_v$. We train and test our procedure with $L^{(\text{train})} = 10$ and $L^{(\text{test})} = 20$ refinement iterations, respectively (Fey *et al.*, 2020a).

Figure 4.8a and Figure 4.8b show the matching accuracy Hits@1 for different choices of $|\mathcal{V}_s|$ and p. We observe that the purely local matching approach via $\mathrm{softmax}(\hat{\boldsymbol{S}}^{(0)})$

starts decreasing in performance with the structural noise p_s increasing. Notably, this also holds when applying global $\mathrm{sinkhorn}$ normalization on $\hat{\boldsymbol{S}}^{(0)}$. However, our proposed two-stage architecture can recover *all* correspondences, independent of the applied structural noise p_s. This applies to both variants discussed in the previous sections, *i.e.*, our initial formulation $\mathrm{sinkhorn}(\hat{\boldsymbol{S}}^{(L)})$, and our optimized architecture using random node indicator sampling and row-wise normalization $\mathrm{softmax}(\hat{\boldsymbol{S}}^{(L)})$. This highlights the overall benefits of applying matching consensus and justifies the usage of the enhancements made towards scalability in Section 4.4.3.3 (Fey *et al.*, 2020a).

In addition, Figure 4.8c visualizes the test error $\mathcal{L}^{(\text{refined})}$ for varying number of iterations $L^{(\text{test})}$. We observe that even when training to non-convergence, our procedure is still able to converge by increasing the number of iterations $L^{(\text{test})}$ during testing (Fey *et al.*, 2020a).

Moreover, Figure 4.8d shows the performance of our refinement strategy when operating on sparsified top k correspondences. In contrast to its dense version, it cannot match all nodes correctly due to the poor initial feature matching quality. However, it consistently converges to the perfect solution of Hits@1 $\approx$ Hits@k in case the correct match is included in the initial top k ranking of correspondences. Hence, with increasing k, we can recover most of the correct correspondences, making it an excellent option to scale our algorithm to large graphs, *cf.* Section 4.4.4.3 (Fey *et al.*, 2020a).

Furthermore, to experimentally validate the robustness of DGMC towards node addition (or removal), we conduct additional synthetic experiments in a similar fashion to Xu *et al.* (2019b). In particular, the target graph $\mathcal{G}_t$ is constructed by first adding $q\%$ noisy nodes to the source graph, *i.e.*, $|\mathcal{V}_t| = (1+q)|\mathcal{V}_s|$, and generating edges between these nodes and all other nodes based on the edge probability p afterwards (Fey *et al.*, 2020a). Figure 4.8e and Figure 4.8f visualize the Hits@1 for different choices of $|\mathcal{V}_s|$, p and $q \in \{0.0, 0.1, \ldots, 0.5\}$. Notably, our consensus stage is extremely robust to the addition or removal of nodes while the first stage alone has major difficulties in finding the right matching. This can be explained by the fact that unmatched nodes do not have any influence on the neighborhood consensus error since those nodes do not obtain a color from the functional map given by $\boldsymbol{S}$. Our neural architecture is able to detect and gradually decrease any false positive influence of these nodes in the refinement stage (Fey *et al.*, 2020a).

4.4.4.2 Supervised Keypoint Matching. We now apply our DGMC architecture to the real-world task of supervised keypoint matching in natural images. For this, we perform experiments on the PascalVOC (Everingham *et al.*, 2010) with Berkeley annotations (Bourdev & Malik, 2009) dataset and the WILLOW-ObjectClass (Cho *et al.*, 2013) dataset, which contain sets of image categories with labeled keypoint locations. For PascalVOC, we follow the experimental setups of Zanfir & Sminchisescu (2018) and Wang *et al.* (2019c) and use the training and test splits provided by Choy *et al.* (2016). We pre-filter the dataset to exclude difficult, occluded and truncated objects, and require examples to have at least one keypoint, resulting in $6,953$ and $1,671$ annotated images for training and testing, respectively. The PascalVOC dataset contains instances of varying scale, pose and illumination, and the number of keypoints ranges from 1 to 19. In contrast, the WILLOW-ObjectClass dataset contains at least 40 images with consistent orientations for each of its five categories, and each image consists of exactly 10 keypoints. Following the experimental setup of peer methods (Cho *et al.*,

Method																						Mean
GMN		31.1	46.2	58.2	45.9	70.6	76.5	61.2	61.7	35.5	53.7	58.9	57.5	56.9	49.3	34.1	77.5	57.1	53.6	83.2	88.6	57.9
PCA-GM		40.9	55.0	**65.8**	47.9	76.9	77.9	63.5	67.4	33.7	**66.5**	63.6	**61.3**	58.9	**62.8**	44.9	77.5	67.4	57.5	86.7	**90.9**	63.8
$f_{\boldsymbol{\theta}} = \mathrm{MLP}$ (isotropic)	$L = 0$	34.7	42.6	41.5	50.4	50.3	72.2	60.1	59.4	24.6	38.1	86.2	47.7	56.3	37.6	35.4	58.0	45.8	74.8	64.1	75.3	52.8
	$L = 10$	45.8	58.2	45.5	57.6	68.2	82.1	75.3	60.2	31.7	52.9	**88.2**	56.2	68.2	50.7	46.5	66.3	58.8	89.0	85.1	79.9	63.3
	$L = 20$	45.3	57.1	54.9	54.7	71.7	82.6	75.3	65.9	31.6	50.8	86.1	56.9	67.1	53.1	49.2	77.3	59.2	91.7	82.0	84.2	64.8
$g_{\boldsymbol{\theta}} = \boldsymbol{A}^{\top}\boldsymbol{X}$ (isotropic)	$L = 0$	44.3	62.0	48.4	53.9	73.3	80.4	72.2	64.2	30.3	52.7	79.4	56.6	62.3	56.2	47.5	74.0	59.8	79.9	81.9	83.0	63.1
	$L = 10$	45.9	60.5	49.0	59.7	72.8	80.9	77.4	67.2	34.1	56.3	80.4	59.5	68.6	53.9	48.6	75.5	60.8	91.5	84.8	80.3	65.4
	$L = 20$	44.7	61.5	53.0	63.1	73.6	81.2	75.2	68.1	33.9	57.1	80.5	59.7	66.5	54.4	51.6	74.9	63.6	85.4	79.6	82.3	65.5
$f_{\boldsymbol{\theta}} = \mathrm{GNN}$ (isotropic)	$L = 0$	44.3	62.0	48.4	53.9	73.3	80.4	72.2	64.2	30.3	52.7	79.4	56.6	62.3	56.2	47.5	74.0	59.8	79.9	81.9	83.0	63.1
	$L = 10$	46.5	63.7	54.9	60.9	79.4	**84.1**	76.4	68.3	**38.5**	61.5	80.6	59.7	69.8	58.4	54.3	76.4	64.5	**95.7**	**87.9**	81.3	68.1
	$L = 20$	**50.1**	**65.4**	55.7	**65.3**	**80.0**	83.5	**78.3**	**69.7**	34.7	60.7	70.4	59.9	**70.0**	62.2	**56.1**	**80.2**	**70.3**	88.8	81.1	84.3	**68.3**
$f_{\boldsymbol{\theta}} = \mathrm{MLP}$ (anisotropic)	$L = 0$	34.3	45.9	37.3	47.7	53.3	75.2	64.5	61.7	27.7	40.5	85.9	46.6	50.2	39.0	37.3	58.0	49.2	82.9	65.0	74.2	53.8
	$L = 10$	44.6	51.2	50.7	58.5	72.3	83.3	76.6	65.6	31.0	57.5	91.7	55.4	69.5	56.2	47.5	85.1	57.9	92.3	86.7	85.9	66.0
	$L = 20$	**48.7**	57.2	47.0	65.3	73.9	87.6	76.7	70.0	30.0	55.5	**92.8**	59.5	67.9	56.9	48.7	87.2	58.3	94.9	87.9	86.0	67.6
$f_{\boldsymbol{\theta}} = \mathrm{GNN}$ (anisotropic)	$L = 0$	42.1	57.5	49.6	59.4	83.8	84.0	78.4	67.5	37.3	60.4	85.0	58.0	66.0	54.1	52.6	93.9	60.2	85.6	87.8	82.5	67.3
	$L = 10$	45.5	**67.6**	56.5	66.8	**86.9**	85.2	84.2	**73.0**	**43.6**	66.0	92.3	**64.0**	**79.8**	56.6	56.1	95.4	64.4	**95.0**	91.3	86.3	72.8
	$L = 20$	47.0	65.7	56.8	**67.6**	**86.9**	**87.7**	**85.3**	72.6	42.9	**69.1**	84.5	63.8	78.1	55.6	**58.4**	**98.0**	**68.4**	92.2	**94.5**	85.5	**73.0**

Table 4.5: Hits@1 (%) performance of DGMC on the PascalVOC dataset with Berkeley keypoint annotations (Fey *et al.*, 2020a)**.**

2013; Wang *et al.*, 2019c), we pre-train our model on PascalVOC and fine-tune it over 20 random splits with 20 per-class images used for training. We construct graphs via the Delaunay triangulation of keypoints. For fair comparison with Zanfir & Sminchisescu (2018) and Wang *et al.* (2019c), input features of keypoints are given by the concatenated output of `relu4_2` and `relu5_1` of a pre-trained VGG16 model (Simonyan & Zisserman, 2014) on the ImageNet dataset (Deng *et al.*, 2009; Fey *et al.*, 2020a).

We adopt SplineCNN as our Graph Neural Network operator (Fey *et al.*, 2018)

$$\boldsymbol{h}_v^{(\ell+1)} = \sigma\left(\boldsymbol{W}^{(t+1)}\boldsymbol{h}_v^{(\ell)} + \sum_{w \in \mathcal{N}(v)} g_{\boldsymbol{\theta}}^{(\ell+1)}(\boldsymbol{e}_{w,v}) \cdot \boldsymbol{h}_w^{(\ell)}\right) \tag{4.24}$$

whose trainable B-spline based kernel function $g_{\boldsymbol{\theta}}$ is conditioned on edge features $\boldsymbol{e}_{w,v}$ between node-pairs, *cf.* Section 3.3. To align our results with the related work, we evaluate both isotropic and anisotropic edge features which are given as relative distances and 2D Cartesian coordinates, respectively. For SplineCNN, we use a kernel size of 5 in each dimension, a hidden dimensionality of 256, and apply ReLU as our non-linearity function σ (Glorot *et al.*, 2011). Our network architecture consists of two graph convolutional layers, followed by dropout with probability 0.5 (Srivastava *et al.*, 2014), and a final linear layer. During training, we form pairs between any two training examples of the same category, and evaluate our model by sampling a fixed number of test graph pairs belonging to the same category (Fey *et al.*, 2020a).

We follow the experimental setup of Wang *et al.* (2019c) and train our models using negative log-likelihood due to its superior performance in contrast to the *displacement loss* used in Zanfir & Sminchisescu (2018), and compare our DGMC approach against the GMN (Zanfir & Sminchisescu, 2018) and PCA-GM (Wang *et al.*, 2019c) graph matching models. We evaluate our complete architecture using isotropic and anisotropic GNNs for $L \in \{0, 10, 20\}$, and include ablation results obtained from using $f_{\boldsymbol{\theta}} = \mathrm{MLP}$ for the local node matching procedure (Fey *et al.*, 2020a). Furthermore, as stated in Section 4.4.3.2, our DGMC algorithm can be viewed as a generalization of the graduated assignment algorithm (Gold & Rangarajan, 1996), extending it by

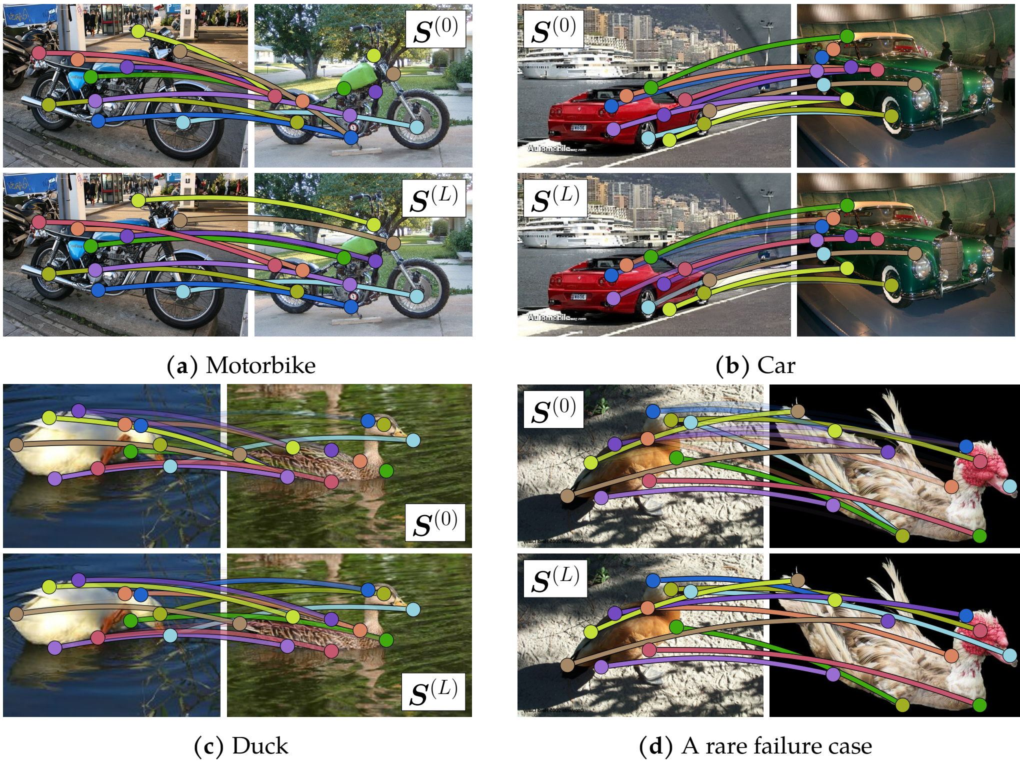
Method			Face	Motorbike	Car	Duck	Winebottle
GMN (Zanfir & Sminchisescu, 2018)			99.3	71.4	74.3	82.8	76.7
PCA-GM (Wang *et al.*, 2019e)			**100.0**	76.7	84.0	**93.5**	96.9
		$L = 0$	98.07±0.79	48.97±4.62	65.30±3.16	66.02±2.51	77.72±3.32
$f_{\boldsymbol{\theta}} = \mathrm{MLP}$	isotropic	$L = 10$	**100.00**±0.00	67.28±4.93	85.07±3.93	83.10±3.61	92.30±2.11
		$L = 20$	**100.00**±0.00	68.57±3.94	82.75±5.77	84.18±4.15	90.36±2.42
		$L = 0$	99.62±0.28	73.47±3.32	77.47±4.92	77.10±3.25	88.04±1.38
$f_{\boldsymbol{\theta}} = \mathrm{GNN}$	isotropic	$L = 10$	**100.00**±0.00	**92.05**±3.49	90.05±5.10	88.98±2.75	**97.14**±1.41
		$L = 20$	**100.00**±0.00	**92.05**±3.24	**90.28**±4.67	88.97±3.49	**97.14**±1.83
		$L = 0$	98.47±0.61	49.28±4.31	64.95±3.52	66.17±4.08	78.08±2.61
$f_{\boldsymbol{\theta}} = \mathrm{MLP}$	anisotropic	$L = 10$	**100.00**±0.00	76.28±4.77	86.70±3.25	83.22±3.52	93.65±1.64
		$L = 20$	**100.00**±0.00	76.57±5.28	89.00±3.88	84.78±2.73	95.29±2.22
		$L = 0$	99.96±0.06	91.90±2.30	91.28±4.89	86.58±2.99	98.25±0.71
$f_{\boldsymbol{\theta}} = \mathrm{GNN}$	anisotropic	$L = 10$	**100.00**±0.00	98.80±1.58	**96.53**±1.55	93.22±3.77	**99.87**±0.31
		$L = 20$	**100.00**±0.00	**99.40**±0.80	95.53±2.93	93.00±2.71	99.39±0.70

Table 4.6: Hits@1 (%) performance with standard deviations of DGMC on the WILLOW-ObjectClass dataset (Fey *et al.*, 2020a)**.**

trainable parameters. To evaluate the impact of a trainable refinement procedure, we perform additional ablation studies on PascalVOC by implementing $g_{\boldsymbol{\theta}}$ via a non-trainable, one-layer GNN instantiation $g_{\boldsymbol{\theta}}(\boldsymbol{X}, \boldsymbol{A}) = \boldsymbol{A}^{\top}\boldsymbol{X}$, and employing a trainable and isotropic $f_{\boldsymbol{\theta}}$ for the first stage.

Results of Hits@1 are shown in Table 4.5 and Table 4.6 for the PascalVOC and WILLOW-ObjectClass datasets, respectively. We observe that the refinement strategy of DGMC is able to significantly outperform competing methods as well as our non-refined baselines. On the WILLOW-ObjectClass dataset, our refinement stage at least reduces the error of the initial model ($L = 0$) by half across all categories. The benefits of the second stage are even more crucial when starting from a weaker initial feature matching baseline ($f_{\boldsymbol{\theta}} = \mathrm{MLP}$), with overall improvements of up to 14 percentage points on PascalVOC. However, good initial matchings do help our consensus stage to improve its performance further, as indicated by the usage of task-specific isotropic or anisotropic GNNs for the initial GNN $f_{\boldsymbol{\theta}}$ (Fey *et al.*, 2020a). Furthermore, using trainable neural networks $g_{\boldsymbol{\theta}}$ consistently improves upon the results of using non-trainable fixed-function message passing operators. While it is difficult to encode meaningful similarities between node and edge features in a fixed-function pipeline, our trainable DGMC approach is able to learn how to make use of those features to guide the refinement procedure further (Fey *et al.*, 2020a).

We further visualize examples of our method for the task of keypoint matching on the WILLOW-ObjectClass dataset in Figure 4.9, in which examples were selected as follows: Figure 4.9a, b and c show examples where the initial feature matching procedure fails, but where our refinement procedure is able to recover *all* correspondences successfully. Figure 4.9d visualizes a rare failure case. However, while the initial feature matching procedure maps most of the keypoints to the same target keypoint, our refinement strategy is still able to successfully resolve this violation. In addition, note that the target image contains wrong labels in this example, *e.g.*, the eye of the duck, so that some keypoint mappings are mistakenly considered to be wrong.



(**a**) Motorbike (**b**) Car (**c**) Duck (**d**) A rare failure case

Figure 4.9: Qualitative examples from the WILLOW-ObjectClass dataset. Images on the left represent the source, whereas images on the right represent the target. For each example, we visualize both the result of the initial feature matching procedure $\boldsymbol{S}^{(0)}$ (top) and the result obtained after refinement $\boldsymbol{S}^{(L)}$ (bottom) (Fey *et al.*, 2020a).

Lastly, we also evaluate our anisotropic DGMC architecture by tackling the task of *geometric keypoint matching*, where we only make use of point coordinates and no additional visual features are available. Here, we follow the experimental training setup of Zhang & Lee (2019), and test the generalization capabilities of our model on the PascalPF dataset (Ham *et al.*, 2016). For training, we generate a synthetic set of graph pairs: We first randomly sample 30–60 source points uniformly from $[-1, 1]^2$, and add Gaussian noise from $\mathcal{N}(0, 0.05^2)$ to these points to obtain the target points. Furthermore, we add 0–20 outliers from $[-1.5, 1.5]^2$ to each point cloud. Finally, we construct graphs by connecting each node with its k-nearest neighbors ($k = 8$). We train our anisotropic keypoint architecture with input $\boldsymbol{x}_v = \mathbf{1} \in \mathbb{R}^1$ for all $v \in \mathcal{V}_s \cup \mathcal{V}_t$ until it has seen exactly $32,000$ synthetic examples. We then evaluate our trained model on the PascalPF dataset (Ham *et al.*, 2016) which consists of $1,351$ image pairs within 20 classes, with the number of keypoints ranging from 4 to 17. Results of Hits@1 are shown in Table 4.7. Overall, our consensus architecture improves upon the state-of-the-art results of Zhang & Lee (2019) on almost all categories while our $L = 0$ baseline is weaker than the results reported in Zhang & Lee (2019), showing the benefits of applying our consensus stage. In addition, it shows that our method works also well even when not taking any visual information into account (Fey *et al.*, 2020a).

Method		Aeroplane	Bicycle	Bird	Boat	Bottle	Bus	Car	Cat	Chair	Cow	Table	Dog	Horse	Motorbike	Person	Plant	Sheep	Sofa	Train	TV	Mean
Zhang & Lee (2019)		76.1	89.8	93.4	96.4	96.2	97.1	94.6	82.8	89.3	**96.7**	89.7	79.5	82.6	83.5	72.8	76.7	77.1	97.3	98.2	**99.5**	88.5
$f_{\boldsymbol{\theta}} = \text{GNN}$ (anisotropic)	$L = 0$	69.2	87.7	77.3	90.4	98.7	98.3	92.5	91.6	94.7	79.4	95.8	90.1	80.0	79.5	72.5	98.0	76.5	89.6	93.4	97.8	87.6
	$L = 10$	**81.3**	**92.2**	94.2	98.8	**99.3**	99.1	98.6	**98.2**	**99.6**	94.1	**100.0**	**99.4**	**86.6**	**86.6**	**88.7**	**100.0**	**100.0**	**100.0**	**100.0**	99.3	**95.8**
	$L = 20$	81.1	92.0	**94.7**	**100.0**	**99.3**	**99.3**	**98.9**	97.3	99.4	93.4	**100.0**	99.1	86.3	86.2	**87.7**	**100.0**	**100.0**	**100.0**	**100.0**	99.3	95.7

Table 4.7: Hits@1 (%) performance of DGMC on the PascalPF dataset, using a synthetic training setup (Fey *et al.*, 2020a)**.**

Method		ZH→EN @1	ZH→EN @10	EN→ZH @1	EN→ZH @10	JA→EN @1	JA→EN @10	EN→JA @1	EN→JA @10	FR→EN @1	FR→EN @10	EN→FR @1	EN→FR @10
GCN (Wang *et al.*, 2018b)		41.25	74.38	36.49	69.94	39.91	74.46	38.42	71.81	37.29	74.49	36.77	73.06
BootEA (Sun *et al.*, 2018)		62.94	84.75	—	—	62.23	85.39	—	—	65.30	87.44	—	—
MuGNN (Cao *et al.*, 2019)		49.40	84.40	—	—	50.10	85.70	—	—	49.60	87.00	—	—
NAEA (Zhu *et al.*, 2019a)		65.01	86.73	—	—	64.14	87.27	—	—	67.32	89.43	—	—
RDGCN (Wu *et al.*, 2019b)		70.75	84.55	—	—	76.74	89.54	—	—	88.64	95.72	—	—
GMNN (Xu *et al.*, 2019d)		67.93	78.48	65.28	79.64	73.97	87.15	71.29	84.63	89.38	95.25	88.18	94.75
$f_{\boldsymbol{\theta}} = \text{MLP}$	$L = 0$	58.53	78.04	54.99	74.25	59.18	79.16	55.40	75.53	76.07	91.54	74.89	90.57
$g_{\boldsymbol{\theta}} = \boldsymbol{A}^{\top}\boldsymbol{X}$	$L = 0$	67.59	**87.47**	64.38	**83.56**	71.95	**89.74**	68.88	**86.84**	83.36	**96.03**	82.16	**95.28**
	$L = 10$	71.61	**87.47**	68.52	**83.56**	77.18	**89.74**	76.53	**86.84**	85.69	**96.03**	85.96	**95.28**
$f_{\boldsymbol{\theta}} = \text{GNN}$	$L = 0$	67.59	**87.47**	64.38	**83.56**	71.95	**89.74**	68.88	**86.84**	83.36	**96.03**	82.16	**95.28**
	$L = 10$	**80.12**	**87.47**	**76.77**	**83.56**	**84.80**	**89.74**	**81.09**	**86.84**	**93.34**	**96.03**	**91.95**	**95.28**

Table 4.8: Hits@1 (%) and Hits@10 (%) performance of DGMC on the DBP15K datasets (Fey *et al.*, 2020a)**.**

4.4.4.3 Semi-Supervised Cross-Lingual Knowledge Graph Alignment. We further evaluate our model on the DBP15K datasets (Sun *et al.*, 2017), which link entities of the Chinese, Japanese and French knowledge graphs of DBpedia into their English version and vice versa. Each dataset contains of exactly 15,000 ground-truth links between equivalent entities in different languages, and we split those links into training and testing following upon previous works. For obtaining entity input features, we follow the experimental setup of Xu *et al.* (2019d): We retrieve monolingual fastText embeddings (Bojanowski *et al.*, 2017) for each language separately, and align those into the same vector space afterwards (Lample *et al.*, 2018). We use the sum of word embeddings as the final entity input representation (although more sophisticated approaches are just as conceivable) (Fey *et al.*, 2020a).

Our utilized Graph Neural Network operator mostly matches the one proposed in Xu *et al.* (2019d) where the direction of edges is retained, but not their specific relation type:

$$\boldsymbol{h}_i^{(\ell+1)} = \sigma\left(\boldsymbol{W}_1^{(\ell+1)}\boldsymbol{h}_i^{(\ell)} + \sum_{(j,i)\in\mathcal{E}} \boldsymbol{W}_2^{(\ell+1)}\boldsymbol{h}_j^{(\ell)} + \sum_{(i,j)\in\mathcal{E}} \boldsymbol{W}_3^{(\ell+1)}\boldsymbol{h}_j^{(\ell)}\right). \tag{4.25}$$

We use ReLU followed by dropout with probability 0.5 as our non-linearity σ, and obtain final node representations via $\boldsymbol{h}_i = \boldsymbol{W}_4[\boldsymbol{h}_i^{(1)}, \ldots, \boldsymbol{h}_i^{(L)}]$. We use a three-layer GNN both for obtaining initial similarities and for refining alignments with dimensionality 256 and 32, respectively. Training is performed using negative log-likelihood in a semi-supervised fashion: For each training node i in $\mathcal{V}_s$, we train $\mathcal{L}^{(\text{initial})}$ sparsely by using the corresponding ground-truth node in $\mathcal{V}_t$, the top $k = 10$ entries in $\boldsymbol{S}_{i,:}$ and

k randomly sampled entities in $\mathcal{V}_t$. For the refinement phase, we update the sparse top k correspondence matrix 10 times. For efficiency reasons, we train $\mathcal{L}^{(\text{initial})}$ and $\mathcal{L}^{(\text{refined})}$ sequentially for 100 epochs each (Fey *et al.*, 2020a).

We report Hits@1 and Hits@10 to evaluate and compare our model to previous lines of work, see Table 4.8. In addition, we report results of a simple three-layer MLP which matches nodes purely based on initial word embeddings, a variant of our model without refinement ($L = 0$), as well as a variant modeling the graduated assignment algorithm via fixed-function refinement networks $g_{\boldsymbol{\theta}}(\boldsymbol{X}, \boldsymbol{A}) = \boldsymbol{A}^{\top}\boldsymbol{X}$. Notably, our approach improves upon the state-of-the-art on all categories with gains of up to 9.38 percentage points. In addition, our trainable refinement strategy consistently improves upon the Hits@1 obtained from initial correspondences and fixed-function refinement by a significant margin, while results of Hits@10 are shared due to the refinement operating only on sparsified top 10 initial correspondences. Due to the scalability of our approach, we can easily apply a multitude of refinement iterations while still retaining large hidden feature dimensionalities (Fey *et al.*, 2020a).

5

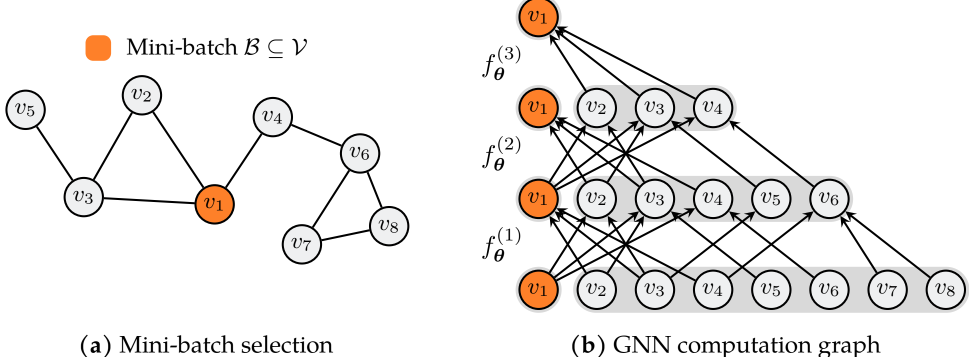
Scalable Graph Neural Networks for Large-Scale Graph Learning

A major challenge of Graph Neural Networks is the difficulty to scale them to large graphs due to the exponentially increasing dependency of nodes over layers. While scalability techniques are indispensable for applying GNNs to large graphs, existing approaches based on graph sub-sampling weaken the expressive power of message passing. In order to overcome this restriction, we propose GNNAutoScale, a framework for scaling arbitrary message passing neural networks to large graphs that is provably able to maintain the expressive power of the original GNN. Furthermore, in order to accelerate GNN research on large-scale graphs, we propose the Open Graph Benchmark, which includes a diverse set of challenging, realistic and large-scale graph benchmark datasets across three different learning tasks. Our experiments suggest that our datasets present significant challenges regarding scalability and out-of-distribution generalization under realistic data splits.

5.1 Introduction 91
5.2 State-of-the-Art 93
5.3 Scaling Up Graph Neural Networks via Historical Embeddings 100
5.4 The Open Graph Benchmark Datasets for Large-Scale Graph Learning 110
5.5 Evaluation 116

5.1 Introduction

One of the challenges that have precluded the wide adoption of Graph Neural Networks (GNNs) in industrial and social applications is the difficulty to scale them to large graphs (Frasca *et al.*, 2020). While most of the research in this field has focused on boosting model performance on small-scale datasets, relatively little effort has been devoted to scaling these methods to gigantic web-scale graphs (Hamilton *et al.*, 2017).



(**a**) Mini-batch selection (**b**) GNN computation graph

Figure 5.1: Illustration of the neighbor explosion phenomenon in mini-batch processed GNNs. Figure (**a**) shows the underlying graph and the subset of mini-batch nodes ■. For simplicity, we assume $|\mathcal{B}| = 1$ with $\mathcal{B} = \{v_1\}$. Figure (**b**) illustrates the computation flow of a three-layer GNN in order to derive the final representation $\boldsymbol{h}_{v_1}^{(3)}$ of node v_1. After only three layers, all nodes of the graph contribute to its representation.

However, as GNNs become more well understood (Chapter 3) and their model instantiations become more sophisticated and data-demanding (Chapter 4), advancements in this field should be especially noticeable with access to an increasing amount of data, as it has been the case for other domains as well (Krizhevsky *et al.*, 2012; Devlin *et al.*, 2018; Brock *et al.*, 2019).

Traditional deep neural networks are known to scale well to large amounts of data by decomposing the training loss into individual samples (called a *mini-batch*) and approximating exact gradients stochastically (Goodfellow *et al.*, 2016). Somewhat surprisingly, the variance induced by such stochastic optimization is even known to improve generalization (Bottoue & Bousquet, 2007). In contrast, applying stochastic mini-batch training in GNNs is challenging since the embedding of a given node depends recursively on all its neighbor's embeddings, leading to high inter-dependency between nodes that grows exponentially with respect to the number of layers, *cf.* Figure 5.1. As a simple workaround, GNNs are typically executed in a *full-batch* fashion, with access to *all* hidden node representations of *all* layers, *cf.* Section 3.2.3. However, this is not feasible in large-scale graphs due to memory limitations and slow convergence (Ma & Tang, 2020). Therefore, it is desirable to approximate its full-batch gradient stochastically as well. For example, given a suitable loss function ϕ for tackling the task of node classification, the gradients of model parameters $\boldsymbol{\theta}$ can be approximated via

$$\nabla \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{V}|} \sum_{v \in \mathcal{V}} \nabla \phi(\boldsymbol{h}_v^{(L)}, y_v) \approx \frac{1}{|\mathcal{B}|} \sum_{v \in \mathcal{B} \subseteq \mathcal{V}} \nabla \phi(\boldsymbol{h}_v^{(L)}, y_v), \tag{5.1}$$

in which only a mini-batch $\mathcal{B} \subseteq \mathcal{V}$ of nodes is considered for loss computation. However, this stochastic gradient is still expensive to compute due to exponentially increasing dependencies of node representations over layers, a phenomenon framed as *neighbor explosion*. Specifically, the representation of a given node depends recursively on all its neighbor's representations, and the number of dependencies grows exponentially with respect to the number of layers. To be more precise, receiving the

final node embedding $\boldsymbol{h}_v^{(L)}$ of a single node $v \in \mathcal{V}$ requires access to $\bar{d}(\mathcal{G})^L$ additional nodes on average, where $\bar{d}(\mathcal{G})$ denotes the average node degree of $\mathcal{G}$.

As a result, scalability techniques are indispensable for applying GNNs to large-scale graphs in order to alleviate the neighbor explosion problem induced by mini-batch training. Here, we introduce our GNNAutoScale (GAS) framework in Section 5.3, which tackles the scalability problem of GNNs by building upon the idea of utilizing historical node embeddings acquired in prior training iterations as affordable approximations (Chen *et al.*, 2018c). As a significant advantage, GAS is provably able to maintain the expressive power of the utilized GNN model (Section 3.4), which is not the case for existing solutions that sub-sample edges or perform non-trainable propagations (Section 5.2.1).

An additional barrier for applying and evaluating GNNs on large-scale graphs (within their respective scalability technique) is the absence of realistic and large-scale graph benchmark datasets (Section 5.2.2). Historically, high-quality and large-scale datasets have played significant roles in advancing machine learning research in several domains, such as in computer vision, natural language processing or speech recognition (Deng *et al.*, 2009; Lin *et al.*, 2014; Wang *et al.*, 2018a; Rajpurkar *et al.*, 2016; Panayotov *et al.*, 2015; Barker *et al.*, 2015). In order to overcome the lack of large-scale graph benchmark datasets, we introduce the Open Graph Benchmark (OGB) in Section 5.4, which includes a diverse set of challenging and realistic benchmark datasets to facilitate scalable, robust, and reproducible graph machine learning research. OGB datasets are orders of magnitude larger than existing ones, encompass multiple important tasks, and cover a diverse range of domains, ranging from social and information networks to biological networks, molecular graphs, source code represented by Abstract Syntax Trees (ASTs), and Knowledge Graphs (KGs). We further show that our datasets present significant challenges of scalability and out-of-distribution generalization under realistic data splits, indicating fruitful opportunities for future research.

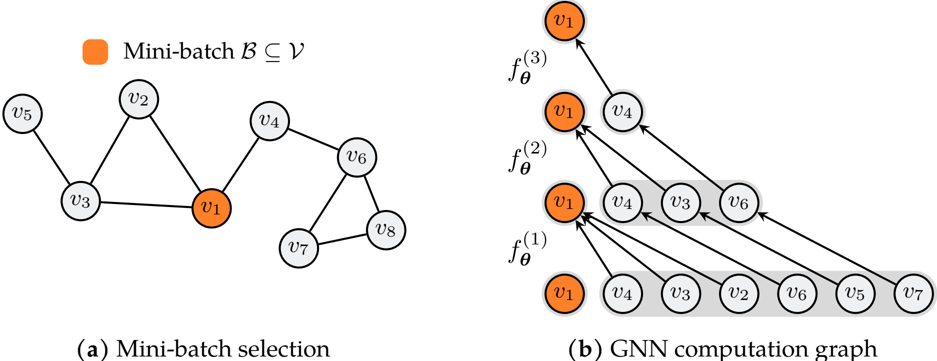
5.2 State-of-the-Art

We briefly review recently proposed state-of-the-art scalability techniques for GNNs (Section 5.2.1) as well as existing graph benchmark datasets (Section 5.2.2).

5.2.1 Scalable Graph Neural Networks

In order to alleviate and overcome the neighbor explosion problem induced by mini-batch GNN training, several works propose different sampling techniques (Hamilton *et al.*, 2017; Chen *et al.*, 2018b; Chiang *et al.*, 2019), *i.e.* node-wise, layer-wise or subgraph-wise sampling, or to decouple propagations from predictions (Wu *et al.*, 2019a; Bojchevski *et al.*, 2020; Chen *et al.*, 2020a).

5.2.1.1 Node-wise Sampling. *Node-wise sampling* approaches (Hamilton *et al.*, 2017; Chen *et al.*, 2018c; Zeng *et al.*, 2020a; Markowitz *et al.*, 2021) recursively sample a fixed number k of neighbors, *i.e.* $\tilde{\mathcal{N}} \subseteq \mathcal{N}(v)$ with $|\tilde{\mathcal{N}}^{(\ell)}(v)| = k$, for a node $v \in \mathcal{V}$, leading to

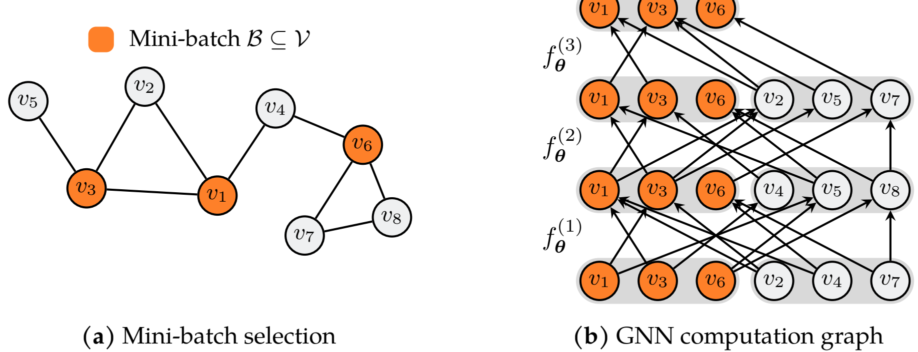


(**a**) Mini-batch selection (**b**) GNN computation graph

Figure 5.2: Node-wise sampling: In each layer and for each node in the previous layer, a fixed number of neighbors k is sampled, leading to a pruned version of the original computation graph. For simplicity, we show examples for $|\mathcal{B}| = 1$ and $k = 1$ here.

an overall bounded L-hop neighborhood of $\mathcal{O}(k^L)$ for each node, *cf.* Figure 5.2. With this, the original computation graph is pruned to contain at most k connections between neurons in different layers. Notably, node-wise sampling approaches can only mitigate the neighbor explosion problem to some extend, as the overall neighborhood size still increases exponentially with the number of layers. As such, in order to allow for scalable and deep GNNs, constant neighborhood sizes in respect to the input node size are required (Fey *et al.*, 2021). Furthermore, the sample size k needs to be large in order to reduce bias in sampling and to keep the predictive performance comparable to the exact algorithm (Chen *et al.*, 2018c). As a result, sampling for more than two iterations is generally not applicable (Hamilton *et al.*, 2017). Chen *et al.* (2018c) try to mitigate this problem by allowing for smaller sample sizes via a control variate based estimator. Alternative solutions (Markowitz *et al.*, 2021; Addanki *et al.*, 2021) still sample a shallow neighborhood patch but make all edges bidirectional, which naturally allows to run GNNs that are deeper than the actual number of sampled hops. In a similar fashion, Zeng *et al.* (2020a) propose to use deep GNNs to pass messages only in shallow, localized subgraphs around individual nodes in the mini-batch.

5.2.1.2 Layer-wise Sampling. In contrast to tracking down inter-layer connections, *layer-wise sampling* techniques (Chen *et al.*, 2018b; Zou *et al.*, 2019; Huang *et al.*, 2018; Hu *et al.*, 2020c) independently sample nodes for each layer, leading to a constant sampled size in each layer (Chen *et al.*, 2018b), *cf.* Figure 5.3. For example FastGCN (Chen *et al.*, 2018b) samples the receptive field for each layer via importance sampling based on node degree. While this layer-wise importance sampling method discards the neighbor-dependent constraints, nodes sampled across layers now suffer from a sparse connection problem (Zou *et al.*, 2019). To counteract, LADIES (Zou *et al.*, 2019) and Huang *et al.* (2018) propose layer-dependent and adaptive samplers, respectively, to constrain neighborhood dependencies, in which nodes are more likely to be sampled if they are connected to already sampled nodes. This guarantees the connectivity of the sampled adjacency matrix (Chen *et al.*, 2020a), and reduces variance in return. In a similar fashion, Hu *et al.* (2020c) extends this technique to heterogeneous graphs.



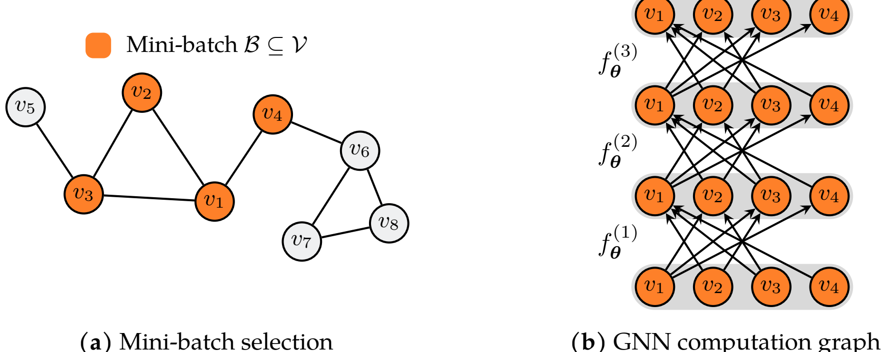
(**a**) Mini-batch selection (**b**) GNN computation graph

Figure 5.3: Layer-wise sampling: In each layer, a distinct set of nodes is sampled and involved in the computation, leading to constant sample sizes.

One major disadvantage of layer-wise sampling approaches is that layer-wise connections between nodes in the computation graph may be too sparse, and intermediate representations may not be learned well in return (Ma & Tang, 2020). Hence, it is required to sample a reasonable amount of nodes per layer in order to obtain accurate embeddings.

5.2.1.3 Subgraph-wise Sampling. In *subgraph-wise sampling* (Chiang *et al.*, 2019; Zeng *et al.*, 2020b), a GNN is applied on the isolated subgraph $\mathcal{G}[\mathcal{B}]$ induced by a sampled mini-batch of nodes $\mathcal{B} \subseteq \mathcal{V}$, *cf.* Figure 5.4. With this, subgraph-wise sampling approaches only need to sample a subgraph at the beginning of each training iteration and propagation is performed on the *same* subgraph across all layers (Chen *et al.*, 2020a). In particular, all sampled nodes of the current mini-batch $\mathcal{B}$ are being used for loss computation, making its formulation and implementation similar to full-batch training. As a result, this technique gets rid of the neighbor explosion problem and is therefore well suitable for the application of deep GNNs in large-scale graphs. However, in contrast to node-wise and layer-wise sampling techniques, subgraph-wise sampling approaches do not allow access to nodes and neighbors outside the current mini-batch, which may contain crucial information for the given down-stream task and therefore may fail to preserve the edges that represent a meaningful topological structure (Fey *et al.*, 2021). As a result, inference still needs to be performed on the full graph in order to achieve reasonable performance (Zeng *et al.*, 2020b). While the application of deep GNNs is possible in subgraph-wise sampling approaches, their receptive field is bounded by the size of the induced subgraph. This prevents the capturing of long-range dependencies despite utilizing deep GNNs.

Various subgraph-wise sampling methods have been proposed (Chiang *et al.*, 2019; Zeng *et al.*, 2020b) which mainly differ in how mini-batches of nodes are acquired: Cluster-GCN (Chiang *et al.*, 2019) uses graph clustering techniques (Karypis & Kumar, 1998; Dhillon *et al.*, 2007) to partition the original graph into several subgraphs, and uses these individual subgraphs to perform feature propagation. With this, mini-batches of nodes are formed based on maximizing their *intra-connectivity*, in which edges within clusters appear much more frequently than edges between different clusters. Intuitively, a node and its neighbors have a high change to be located in the same



(**a**) Mini-batch selection (**b**) GNN computation graph

Figure 5.4: Subgraph-wise sampling: Subgraph-wise sampling approaches perform feature propagation on the induced subgraph of sampled mini-batch nodes ■ across all layers.

cluster (Chiang *et al.*, 2019), leading to the maintenance of most of the original edges in the induced subgraph $\mathcal{G}[\mathcal{B}]$. However, the usage of deterministic clustering algorithms to partition the graph in a pre-processing step inherently *biases* the GNN model. A GNN can no longer infer by itself which neighbors are important for the given downstream task. GraphSAINT (Zeng *et al.*, 2020b) tries to eliminate the bias of Cluster-GCN by stochastically sampling the mini-batch of nodes $\mathcal{B}$. Bias and variance are reduced via normalization techniques and specialized sampling algorithms, respectively. Besides random node and edge samplers, GraphSAINT proposes to make use of a random walk based sampler in order to strengthen the intra-connectivity of mini-batches (Zeng *et al.*, 2020b). However, due to the normalization technique requirement, GraphSAINT can only be applied in inductive learning scenarios.

In Section 5.3, we introduce our own general scalability framework named GNNAutoScale (GAS). Notably, GAS can also be seen as a subgraph-wise sampling strategy, but overcomes the downsides of the aforementioned methods, *e.g.*, the restrictive learning from shallow subgraphs, by incorporating out-of-mini-batch information via historical embeddings. It was further extended in Ding *et al.* (2021) to mitigate the overhead of historical embedding storage via vector quantization.

5.2.1.4 Decoupling Predictions from Propagations. Another line of work is based on the idea of decoupling propagations from predictions (Klicpera *et al.*, 2019a) by computing L-hop subgraph representations of a node in a *pre-processing* step (Wu *et al.*, 2019a; Frasca *et al.*, 2020; Yu *et al.*, 2020; Chen *et al.*, 2020a). The idea is simple yet powerful: Instead of obtaining node representations by learning proper parameters of a GNN in a data-dependent fashion, these approaches utilize *non-trainable* GNN variants, *i.e.* both Message and Update denote fixed-functions. As such, the representation $\boldsymbol{h}_v^{(L)}$ of each node $v \in \mathcal{V}$ can be computed in a pre-processing step that only needs to be conducted once before the actual training procedure starts. The training procedure then involves learning the parameters $\boldsymbol{\theta}$ of an $\mathrm{MLP}_{\boldsymbol{\theta}}$ that performs the final prediction, taking the pre-computed representations $\boldsymbol{h}_v^{(L)}$ as input rather than the raw input features $\boldsymbol{h}_v^{(0)}$. This scheme easily scales to a large number of nodes via stochastic gradient descent, as the training loss can be decomposed into individual

Technique	Pre-Processing		Training		Inference	
	Runtime	Memory	Runtime	Memory	Runtime	Memory
Full-batch	—	—	$\mathcal{O}(L \cdot \lvert\mathcal{E}\rvert)$	$\mathcal{O}(L \cdot \lvert\mathcal{V}\rvert)$	$\mathcal{O}(L \cdot \lvert\mathcal{E}\rvert)$	$\mathcal{O}(\lvert\mathcal{V}\rvert)$
Node-wise	—	—	$\mathcal{O}(k^L \cdot \lvert\mathcal{B}\rvert)$	$\mathcal{O}(k^L \cdot \lvert\mathcal{B}\rvert)$	$\mathcal{O}(\bar{d}^L \cdot \lvert\mathcal{B}\rvert)$	$\mathcal{O}(\bar{d}^L \cdot \lvert\mathcal{B}\rvert)$
Layer-wise	—	—	$\mathcal{O}(L \cdot k \cdot \lvert\mathcal{B}\rvert)$	$\mathcal{O}(L \cdot k \cdot \lvert\mathcal{B}\rvert)$	$\mathcal{O}(L \cdot \lvert\mathcal{E}\rvert)$	$\mathcal{O}(\lvert\mathcal{V}\rvert)$
Subgraph-wise	—	—	$\mathcal{O}(L \cdot \lvert\mathcal{E}[\mathcal{B}]\rvert)$	$\mathcal{O}(L \cdot \lvert\mathcal{B}\rvert)$	$\mathcal{O}(L \cdot \lvert\mathcal{E}\rvert)$	$\mathcal{O}(\lvert\mathcal{V}\rvert)$
Decoupling	$\mathcal{O}(L \cdot \lvert\mathcal{E}\rvert)$	$\mathcal{O}(\lvert\mathcal{V}\rvert)$	$\mathcal{O}(\lvert\mathcal{B}\rvert)$	$\mathcal{O}(\lvert\mathcal{B}\rvert)$	$\mathcal{O}(\lvert\mathcal{B}\rvert)$	$\mathcal{O}(\lvert\mathcal{B}\rvert)$

Table 5.1: Summary of time and memory complexities for pre-processing, training and inference across different GNN scalability techniques. We set the dimensionality of node embeddings to be constant. Here, $|\mathcal{E}[\mathcal{B}]|$ denotes the number of edges in the induced subgraph $\mathcal{G}[\mathcal{B}]$.

samples that do not longer share inter-dependencies with each other (just like in traditional deep neural networks). Different instantiations of fixed Message and Update functions for performing the initial feature propagation have been proposed, *e.g.*, via the normalized Laplacian matrix (Wu *et al.*, 2019a; Frasca *et al.*, 2020) or the personalized PageRank matrix (Bojchevski *et al.*, 2020; Chen *et al.*, 2020a). In contrast to the aforementioned pre-processing approaches, Huang *et al.* (2021) utilize the idea of decoupling predictions from propagations in a *post-processing* step. Here, an Multi-Layer Perceptron (MLP) first learns to predict node-wise labels in a graph-agnostic fashion, after which the predictions are then smoothed and adjusted across the given graph via a fixed-function GNN pipeline.

The major disadvantage of such an approach is that the utilized fixed-function GNN is no longer trainable, and as such, the obtained feature representations do not necessarily need to align with the given task at hand and are provably less expressive, *cf.* Section 3.4. In addition, powerful GNNs that utilize, *e.g.*, attention for feature aggregation are no longer applicable. However, it is worth noting that the performance of such simplified models is usually quite competitive w.r.t. more sophisticated GNN instantiations, in particular due to their significant advantages regarding time and memory complexities.

5.2.1.5 Comparison. Table 5.1 gives a high-level overview over the individual runtime and memory complexities of the aforementioned GNN scalability techniques. For node-wise sampling approaches, the exponentially increasing neighborhood size over layers k^L dominates both the runtime and memory complexity of training. During inference, predictions needs to be made on the full neighborhood without any stochasticity, further increasing complexities to $\bar{d}^L$, with $\bar{d}$ denoting the average node degree. In contrast, both layer-wise and subgraph-wise sampling techniques achieve linear runtime and memory complexities during training. However, for inference, computation still needs to be applied on the full graph. In addition, the larger the input graph is, the more challenging it becomes for layer-wise and subgraph-wise approaches to construct meaningful neighborhoods. Approaches that decouple predictions from propagations achieve the best runtime and memory complexities both for the training and inference phase since heavy computations are pre-processed. However, they miss out on the benefits of differentiable message passing.

Our proposed scalability approach as introduced Section 5.3 can be seen as a subgraph-wise technique that scales linearly with the number of layers L. However, in contrast to existing approaches, it is able to incorporate all available information into its message passing formulation. As such, it is not subject to the problem of finding meaningful neighborhoods in the first place. In addition, it can be used for accelerating the inference stage of a GNN as well.

Furthermore, all existing approaches are still restricted to shallow graph-structures and non-exchangeable GNN operators. In particular, all techniques consider only specific GNN operators and it is an open question whether these techniques can be successfully applied to the wide range of GNN architectures available (Veličković *et al.*, 2018; Xu *et al.*, 2019c; Corso *et al.*, 2020; Chen *et al.*, 2020b). Notably, our scalability framework can be applied to any message passing GNN backbone.

5.2.2 Shortcomings of Existing Graph Benchmark Datasets

In order to track progress in graph machine learning, there is an urgent need for standardized and realistic graph datasets in which model performances are reproducible and statistically significant. In particular, small datasets make it hard to rigorously evaluate data-hungry models such as GNNs (Li *et al.*, 2016b; Duvenaud *et al.*, 2015; Gilmer *et al.*, 2017; Xu *et al.*, 2019c), leading to unstable performances. Furthermore, most studies adopt their own dataset splits, evaluation metrics, and cross-validation protocols, making it challenging to compare performance reported across various studies (Shchur *et al.*, 2018; Errica *et al.*, 2020; Dwivedi *et al.*, 2020). In addition, many studies follow the convention of using random splits to generate training and test sets (Kipf & Welling, 2017; Xu *et al.*, 2019c; Bordes *et al.*, 2013), which is not realistic or useful for real-world applications and generally leads to overly optimistic performance results (Lohr, 2009). As such, both fixed and realistic data splits as well as standardized evaluation metrics are important so that progress can be measured in a consistent and reproducible way.

We now review commonly-used graph benchmark datasets, and organize the discussion around the three main categories of graph machine learning tasks: predictions at the level of nodes, links, and graphs.

5.2.2.1 Node Property Prediction. Currently, the three citation graphs Cora, CiteSeer and PubMed (Sen *et al.*, 2008; Yang *et al.*, 2016) have been widely used as semi-supervised node classification datasets, particularly for evaluating GNN performance (Section 3.2.3). However, the sizes of these graphs are rather small, ranging from 2,700 to 20,000 nodes. Recent studies suggest that datasets at this small scale can be solved quite well with simple GNN architectures (Shchur *et al.*, 2018; Wu *et al.*, 2019a), and the performance of different GNNs on these datasets is nearly statistically identical (Dwivedi *et al.*, 2020). Furthermore, there is no consensus on the splitting procedures for these datasets, which makes it hard to fairly compare different model designs (Shchur *et al.*, 2018). Finally, a recent study (Zou *et al.*, 2020) shows that these datasets have some fundamental data quality issues. For example, in Cora, 42% of the nodes leak information between their features and labels, and 1% of the nodes are duplicated. The situation for CiteSeer is even worse, with leakage rates of 62% and duplication rates of 5%.

Some recent works in Graph Representation Learning have proposed relatively large datasets, such as PPI (56,944 nodes), Reddit (334,863 nodes) (Hamilton *et al.*, 2017) and Amazon (2,449,029 nodes) (Chiang *et al.*, 2019). However, there exist some inherent issues with the proposed data splits. Specifically, 83%, 65% and 90% of the nodes are used for training in the PPI, Reddit and Amazon datasets, respectively, which results in an artificially small distribution shift across the training/validation/test sets. Consequently, as may be expected, the performance improvements on these benchmarks have quickly saturated. For example, recent GNN models (Chiang *et al.*, 2019; Zeng *et al.*, 2020b) can already yield F1 scores of 99.5 for PPI and 97.0 for Reddit, and 90.4% accuracy for Amazon, with extremely small generalization gaps between training and test accuracy. Finally, it is also practically required for GNNs to handle web-scale graphs (*e.g.*, beyond 100 million nodes and 1 billion edges) in industrial applications (Ying *et al.*, 2018a). However, there have been no publicly available graph datasets of such scale with sufficient label information.

In summary, several factors (*e.g.*, size, leakage, splits, ...) associated with the current use of existing datasets make them unsuitable as benchmark datasets for graph machine learning.

5.2.2.2 Link Property Prediction. Broadly, there are two lines of efforts for the link-level task: link prediction in homogeneous networks (Liben-Nowell & Kleinberg, 2007; Zhang & Chen, 2018) and relation completion in (heterogeneous) Knowledge Graphs (KGs) (Bordes *et al.*, 2013; Nickel *et al.*, 2015; Hu *et al.*, 2020c). There are several problems with the current benchmark datasets in these areas:

First, representative datasets are either extremely small or do not come with input node features. For example, while the well-known recommender system datasets used in van den Berg *et al.* (2017) include node features, their sizes are very small, with the largest having only 6,000 nodes. On the other hand, although the Open Academic Graph used in Qiu *et al.* (2019) comprises tens of millions of nodes, there are no associated node features. Regarding the KG completion, the widely-used dataset FB15k is very small, containing only 14,951 entities, which is a tiny subset of the original Freebase KG with more than 50 million entities (Bollacker *et al.*, 2008).

Second, similar to the node-level task, random splits are predominantly used in link-level prediction (Bordes *et al.*, 2013; Grover & Leskovec, 2016). The random splits are not realistic in many practical applications such as friend recommendation in social networks, in which test edges (friend relations *after* a certain timestamp) naturally follow a different distribution from training edges (friend relations *before* a certain timestamp).

Finally, the existing datasets are mostly oriented to applications in recommender systems, social media and KGs, in which the graphs are typically very sparse. This may result in techniques specialized for sparse link inference that are not generalizable to domains with dense graphs, such as the protein-protein association graphs and drug-drug interaction networks typically found in biology and medicine (Szklarczyk *et al.*, 2019; Wishart *et al.*, 2018; Davis *et al.*, 2019; Szklarczyk *et al.*, 2016; Piñero *et al.*, 2020). Very recently, Sinha *et al.* (2020) proposed a synthetic link prediction benchmark to diagnose the logical generalization capability of the model. Their focus is on synthetic tasks, which is complementary to OGB that focuses on realistic tasks.

5.2.2.3 Graph Property Prediction. Graph-level prediction tasks are found in important applications in natural sciences, such as predicting molecular properties in chemistry (Duvenaud *et al.*, 2015; Gilmer *et al.*, 2017; Hu *et al.*, 2020b), where molecules are naturally represented as molecular graphs.

In graph classification, the most widely-used graph-level datasets from the TU collection (Morris *et al.*, 2020a) are known to have many issues, such as small sizes (*i.e.*, most of the datasets only contain less than 1,000 graphs),[1] unrealistic settings (*e.g.*, no bond features for molecules), random data splits, inconsistent evaluation protocols, and isomorphism bias (Ivanov *et al.*, 2019). A very recent attempt (Dwivedi *et al.*, 2020) to address these issues mainly focuses on benchmarking the building blocks of GNNs rather than developing application-oriented realistic datasets. In fact, five out of the six proposed datasets are purely synthetic.

Recent works in Graph Representation Learning (Hu *et al.*, 2020b; Ishiguro *et al.*, 2019) have started to adopt MoleculeNet (Wu *et al.*, 2018) which contains a set of realistic and large-scale molecular property prediction datasets. However, there is limited consensus in the dataset splitting and molecular graph features, making it hard to compare different models in a fair manner.

5.3 Scaling Up Graph Neural Networks via Historical Embeddings

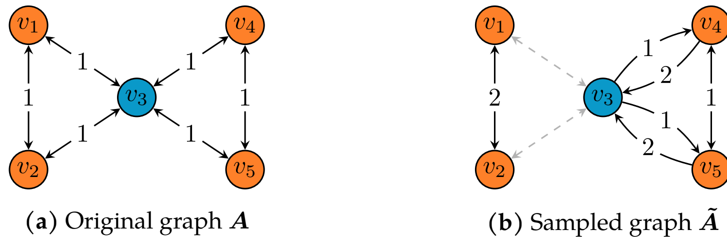
Scalability techniques are indispensable for applying GNNs to large-scale graphs. Although empirical results suggest that existing solutions as described in Section 5.2.1 can scale the training of GNNs to large-scale graphs (Section 5.5), they actually weaken the expressive power of message passing due to sub-sampling of edges or non-trainable propagations. In particular, it is well known that the most powerful GNNs adhere to the same representational power as the Weisfeiler-Lehman (WL) test (Weisfeiler & Lehman, 1968) in distinguishing non-isomorphic subgraph structures, *i.e.* $\boldsymbol{h}_v^{(L)} \neq \boldsymbol{h}_w^{(L)}$ in case $c_v^{(L)} \neq c_w^{(L)}$ (Xu *et al.*, 2019c; Morris *et al.*, 2019), where $c_v^{(L)}$ denotes a node's coloring after L rounds of color refinement, *cf.* Section 3.4. However, in order to leverage such expressiveness, a GNN needs to be able to reason about structural differences across neighborhoods directly *during* training. We now show that GNNs that scale by sampling edges are not capable of doing so (Fey *et al.*, 2021):

Proposition 5. *Let $f_{\boldsymbol{\theta}}^{(L)} \colon \mathcal{V} \to \mathbb{R}^d$ be an L-layer GNN that is as expressive as the WL test in distinguishing the L-hop neighborhood around each node $v \in \mathcal{V}$. Then, there exists a graph $\boldsymbol{A} \in \{0,1\}^{|\mathcal{V}| \times |\mathcal{V}|}$ and a sampled variant $\tilde{\boldsymbol{A}} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ of it, where*

$$\tilde{A}_{v,w} = \begin{cases} \frac{|\mathcal{N}(v)|}{|\tilde{\mathcal{N}}(v)|}, & \textit{if } w \in \tilde{\mathcal{N}}(v) \\ 0, & \textit{otherwise} \end{cases}$$

with $\tilde{\mathcal{N}}(v) \subseteq \mathcal{N}(v)$ denoting the sampled neighborhood around $v \in \mathcal{V}$, such that $f_{\boldsymbol{\theta}}^{(L)}$ operating on $\tilde{\boldsymbol{A}}$ produces a non-equivalent coloring, i.e. *$\tilde{\boldsymbol{h}}_v^{(L)} \neq \tilde{\boldsymbol{h}}_w^{(L)}$ while $c_v^{(L)} = c_w^{(L)}$ for nodes $v, w \in \mathcal{V}$.*

[1] Recently, some progress has been made to increase the dataset sizes: `http://graphlearning.io` (last access: August 25, 2022). Nevertheless, most of them are still small compared to the OGB datasets, and evaluation protocols are not standardized.



(**a**) Original graph $\boldsymbol{A}$ (**b**) Sampled graph $\tilde{\boldsymbol{A}}$

Figure 5.5: Example of a graph $\boldsymbol{A}$ and its sampled variant $\tilde{\boldsymbol{A}}$ for which a maximally powerful GNN produces non-equivalent node embeddings, *i.e.* $f_{\boldsymbol{\theta}}(\boldsymbol{A})_{v_1} = f_{\boldsymbol{\theta}}(\boldsymbol{A})_{v_4}$ while $f_{\boldsymbol{\theta}}(\tilde{\boldsymbol{A}})_{v_1} \neq f_{\boldsymbol{\theta}}(\tilde{\boldsymbol{A}})_{v_4}$. As such, operating on sampled graph variants will lose out on graph structural information (Fey *et al.*, 2021).

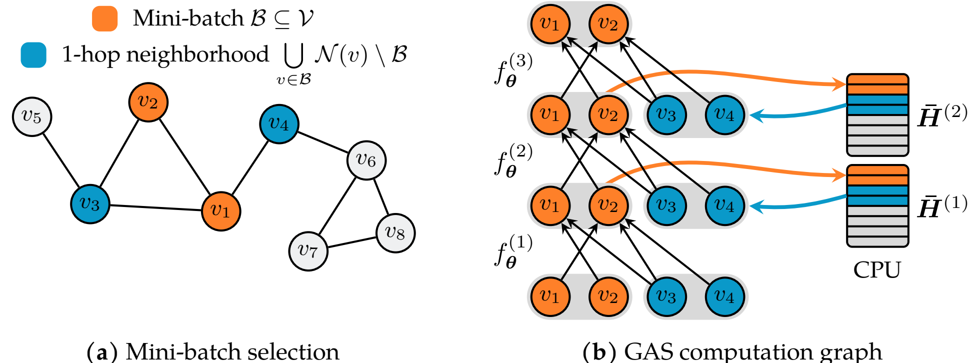
Proof. Consider the colored graph $\boldsymbol{A}$ and its sampled variant $\tilde{\boldsymbol{A}}$ as shown in Figure 5.5. In this counter-example, there exists nodes v_1 and v_4 that receive a different coloring due to sampling of edges despite being indistinguishable by the WL test, *cf.* Section 3.4.1. In particular, it holds that $f_{\boldsymbol{\theta}}(\boldsymbol{A})_{v_1} = f_{\boldsymbol{\theta}}(\boldsymbol{A})_{v_4}$ while $f_{\boldsymbol{\theta}}(\tilde{\boldsymbol{A}})_{v_1} \neq f_{\boldsymbol{\theta}}(\tilde{\boldsymbol{A}})_{v_4}$. □

By only considering a subset of connections of the actual computation graph, GNNs that utilize sampling strategies will lose expressive power in reasoning about graph structures. Although they will encounter every edge at least once during the whole training procedure with high probability, this does not hold in the case of a single optimization step, leading to decreased model power in return. As long as a GNN does not have access to its full graph structure, it cannot learn about discriminating structural graph properties at full capacity. This can be seen as a major limitation of existing scalability solutions, in particular w.r.t. recent lines of work that design GNNs that are as equally powerful as the WL test (Xu *et al.*, 2019c; Corso *et al.*, 2020), as well as higher-order variants to increase their representational power even further (Morris *et al.*, 2019; Murphy *et al.*, 2019b; Maron *et al.*, 2019a; Bouritsas *et al.*, 2020; Morris *et al.*, 2020a; Fey *et al.*, 2020b).

Thus, a natural question to ask is whether there exists a scalable GNN technique that is provably able to maintain the expressive power of the original GNN. Such a solution is condensed in our *GNNAutoScale* (*GAS*) framework (Fey *et al.*, 2021), which scales arbitrary message-passing GNNs to large-scale graphs without the necessity of dropping any edges or neighborhood information. As a result, GAS is provably able reason about graph structural properties at scale. GAS achieves this by pruning entire sub-trees of the computation graph while approximating the missing neighborhood information via the usage of historical embeddings from prior training iterations, and tightening proven approximation error bounds in practice. Here, we first derive the foundations of our GAS framework (Section 5.3.1) before analyzing its theoretical properties in detail (Sections 5.3.2 and 5.3.3).

5.3.1 Historical-based Sub-Tree Pruning

Let $\boldsymbol{h}_v^{(\ell)}$ denote the node embedding in layer $\ell \in \{1, \ldots, L\}$ of a node $v \in \mathcal{B}$ in a mini-batch $\mathcal{B} \subseteq \mathcal{V}$. Then, for the general message scheme given in Equation (3.5), the



(**a**) Mini-batch selection (**b**) GAS computation graph

Figure 5.6: Mini-batch processing of GNNs with historical embeddings. ■ denotes the nodes in the current mini-batch and ■ represents their direct 1-hop neighbors. For a given mini-batch (a), the usage of historical embeddings avoids the neighbor explosion problem by *pruning* entire sub-trees of the computation graph, leading to constant GPU memory consumption in respect to input node size (c). Here, nodes in the current mini-batch *push* their updated embeddings to the history $\bar{H}^{(\ell)}$, while direct neighbors *pull* their most recent historical embeddings from $\bar{H}^{(\ell)}$ (Fey *et al.*, 2021).

execution of $f_{\theta}^{(\ell+1)}$ can be formulated as:

$$\begin{aligned} h_v^{(\ell+1)} &= f_{\theta}^{(\ell+1)}\Big(h_v^{(\ell)}, \{\{h_w^{(\ell)} : w \in \mathcal{N}(v)\}\}\Big) && (5.2)\\ &= f_{\theta}^{(\ell+1)}\Big(h_v^{(\ell)}, \{\{h_w^{(\ell)} : w \in \mathcal{N}(v) \cap \mathcal{B}\}\} \cup \{\{h_w^{(\ell)} : w \in \mathcal{N}(v) \setminus \mathcal{B}\}\}\Big) && (5.3)\\ &\approx f_{\theta}^{(\ell+1)}\Big(h_v^{(\ell)}, \{\{h_w^{(\ell)} : w \in \mathcal{N}(v) \cap \mathcal{B}\}\} \cup \underbrace{\{\{\bar{h}_w^{(\ell)} : w \in \mathcal{N}(v) \setminus \mathcal{B}\}\}}_{\text{Historical embeddings}}\Big). && (5.4) \end{aligned}$$

Here, we separate the neighborhood information of the multiset into *two* parts: (1) the local information of neighbors $\mathcal{N}(v)$ which are part of the current mini-batch $\mathcal{B}$, and (2) the information of neighbors which are not included in the current mini-batch (Fey *et al.*, 2021). For out-of-mini-batch nodes, we approximate their embeddings via *historical embeddings*, which are defined as node embeddings acquired in previous training iterations (Chen *et al.*, 2018c), denoted by $\bar{h}_w^{(\ell)}$. Historical embeddings act as an offline storage and are used to accurately fill in the inter-dependency information of out-of-mini-batch nodes. After each step of training, the newly computed embeddings $h_v^{(\ell+1)}$ are *pushed* to the history and serve as historical embeddings $\bar{h}_w^{(\ell+1)}$ in future iterations. The separation of in-mini-batch nodes and out-of-mini-batch nodes, and their approximation via historical embeddings represent the foundation of our GAS framework.

A high-level illustration of its computation flow is visualized in Figure 5.6. For a given mini-batch of nodes, GAS prunes the GNN computation graph so that only nodes inside the current mini-batch and their direct 1-hop neighbors are retained, *independent* of GNN depth. The required historical embeddings are pulled from an offline storage, instead of being re-computed in each iteration, which keeps the required information for each batch local while still accounting for *all* available neighborhood information.

Algorithm 6 GAS mini-batch execution

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, node features $\boldsymbol{H}^{(0)}$, number of batches B, number of layers L
$\{\mathcal{B}_1, \ldots, \mathcal{B}_B\} \leftarrow \textsc{Split}(\mathcal{G}, B)$
$\mathcal{V}_b \leftarrow \bigcup_{v \in \mathcal{B}_b} \mathcal{N}(v) \cup \{v\}$ $\quad \forall b \in \{1, \ldots, B\}$
$\mathcal{G}_b \leftarrow \mathcal{G}[\mathcal{V}_b]$ $\quad \forall b \in \{1, \ldots, B\}$
for $\mathcal{B}_b \in \{\mathcal{B}_1, \ldots, \mathcal{B}_B\}$ **do**
  for $\ell \in \{1, \ldots, L-1\}$ **do**
    $\boldsymbol{h}_v^{(\ell)} \leftarrow f_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{h}_v^{(\ell-1)}, \{\!\{\boldsymbol{h}_w^{(\ell-1)} : w \in \mathcal{N}(v)\}\!\})$ $\quad \forall v \in \mathcal{B}_b$
    $\textsc{Push}^{(\ell)}(\boldsymbol{h}_v^{(\ell)})$ $\quad \forall v \in \mathcal{B}_b$
    $\boldsymbol{h}_w^{(\ell)} \leftarrow \textsc{Pull}^{(\ell)}(w)$ $\quad \forall w \in \mathcal{V}_b \setminus \mathcal{B}_b$
  end for
  $\boldsymbol{h}_v^{(L)} \leftarrow f_{\boldsymbol{\theta}}^{(L)}(\boldsymbol{h}_v^{(L-1)}, \{\!\{\boldsymbol{h}_w^{(L-1)} : w \in \mathcal{N}(v)\}\!\})$ $\quad \forall v \in \mathcal{B}_b$
end for

In particular, this avoids the neighbor explosion problem and leads to constant GPU memory consumption in respect to input node size. For a single batch $\mathcal{B} \subseteq \mathcal{V}$, the GPU memory footprint for one training step is given by $\mathcal{O}(|\bigcup_{v \in \mathcal{B}} \mathcal{N}(v) \cup \{v\}| \cdot L)$ and thus only scales linearly with the number of layers L. The vast majority of data (the histories) can be stored in RAM or hard drive storage rather than GPU memory, which is usually available in larger scale.

In the following, we are going to use $\tilde{\boldsymbol{h}}_v^{(\ell)}$ to denote embeddings estimated via GAS (Equation (5.4)) to differentiate them from the exact embeddings obtained without historical approximation (Equation (5.2)). In contrast to existing scalability solutions based on sub-sampling edges, the usage of historical embeddings as utilized in GAS provides the following additional advantages (Fey *et al.*, 2021):

- **GAS trains over all data:** In GAS, a GNN will make use of all available graph information, *i.e. no* edges are dropped, which results in low variance and more accurate estimations (since $\|\bar{\boldsymbol{h}}_v^{(\ell)} - \boldsymbol{h}_v^{(\ell)}\| \ll \|\boldsymbol{h}_v^{(\ell)}\|$). Importantly, for a single epoch and layer, each edge is still only processed once, putting its time complexity $\mathcal{O}(|\mathcal{E}|)$ on par with its full-batch counterpart. Notably, more accurate estimations will further strengthen gradient estimation during backpropagation. Specifically, the model parameters will be updated based on the node embeddings of *all* neighbors since $\partial\tilde{\boldsymbol{h}}_v^{(\ell+1)}/\partial\boldsymbol{\theta}$ also depends on $\{\!\{\bar{\boldsymbol{h}}_w^{(\ell)} : w \in \mathcal{N}(v) \setminus \mathcal{B}\}\!\}$.
- **GAS enables constant inference time complexity:** The time complexity of model inference is reduced to a constant factor, since we can directly use the historical embeddings of the last layer to derive predictions for test nodes. This is a major advantage compared to sampling approaches, as they rely on full-batch inference in order to derive accurate predictions (Zeng *et al.*, 2020b).
- **GAS is simple to implement:** Our scheme does not need to maintain recursive layer-wise computation graphs, which makes GAS straightforward to implement comparable to full-batch execution. Only minor modifications are required to *pull* information from and *push* information to the histories after each application of a GNN layer, *cf.* Algorithm 6. Furthermore, this easily allows us to utilize advanced techniques of GNNs as well, such as applying Jumping Knowledge networks (Xu *et al.*, 2018).

- **GAS provides theoretical guarantees:** In particular, if the model weights are kept fixed, the estimated node embeddings $\tilde{\boldsymbol{h}}_v^{(\ell)}$ eventually equal the exact node embeddings $\boldsymbol{h}_v^{(\ell)}$ after a fixed amount of iterations (Chen *et al.*, 2018c).

Notably, the usage of historical embeddings was originally introduced by Chen *et al.* (2018c) in the *Variance Reduction Graph Convolutional Networks* (*VR-GCNs*). VR-GCN aims to reduce the variance in estimation during node-wise sampling (Hamilton *et al.*, 2017), which avoids the need to sample a large amount of neighbors in return. Here, we generalize the idea of utilizing historical embeddings by simplifying it into a *one-shot sampling* scenario, where nodes no longer need to recursively explore neighborhoods in each layer. Furthermore, we do not restrict our framework and analysis (Sections 5.3.2 and 5.3.3) to specific GNN operators, allowing application to the wide range of GNN architectures available (Veličković *et al.*, 2018; Xu *et al.*, 2019c; Corso *et al.*, 2020; Chen *et al.*, 2020b), *cf.* Section 5.5.

5.3.2 Analysis of Historical-caused Approximation Errors

The advantages of utilizing historical embeddings $\bar{\boldsymbol{h}}_v^{(\ell)}$ to compute an approximation $\tilde{\boldsymbol{h}}_v^{(\ell)}$ of the exact embedding $\boldsymbol{h}_v^{(\ell)}$, as described in Section 5.3.1, come at the cost of an approximation error $\|\tilde{\boldsymbol{h}}_v^{(\ell)} - \boldsymbol{h}_v^{(\ell)}\|$ on the output, which can be decomposed into two sources of variance: (1) The *closeness* of estimated inputs to their exact values, *i.e.* $\|\tilde{\boldsymbol{h}}_v^{(\ell-1)} - \boldsymbol{h}_v^{(\ell-1)}\| \geq 0$, and (2) the *staleness* of historical embeddings to their estimated values, *i.e.* $\|\bar{\boldsymbol{h}}_v^{(\ell-1)} - \tilde{\boldsymbol{h}}_v^{(\ell-1)}\| \geq 0$. In the following, we show concrete bounds for this error, which we then tighten using specific procedures (Fey *et al.*, 2021). Notably, our analysis focuses on arbitrary message-passing GNN layers $f_{\boldsymbol{\theta}}^{(\ell)}$ as described in Equation (3.5) in Section 3.2.2, but we restrict both $\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}$ and $\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}$ to model k-Lipschitz continuous functions due to their potentially highly non-linear nature. In particular, a function $f \colon \mathbb{R}^n \to \mathbb{R}^m$ is *Lipschitz continuous* if it satisfies

$$\|f(\boldsymbol{x}) - f(\boldsymbol{y})\| \leq k\|\boldsymbol{x} - \boldsymbol{y}\| \quad \text{for all} \quad \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n \tag{5.5}$$

for some real-valued *Lipschitz constant* $k \geq 0$. Furthermore, we call f *δ-locally Lipschitz continuous* in $\boldsymbol{x}$ if f is Lipschitz continuous in the closed ball $\mathcal{B}_\delta(\boldsymbol{x})$ with radius δ centered around $\boldsymbol{x}$, *i.e.* $\mathcal{B}_\delta(\boldsymbol{x}) = \{\boldsymbol{y} \in \mathbb{R}^n \colon \|\boldsymbol{x} - \boldsymbol{y}\| \leq \delta\}$. In case f can be modeled as a series of function compositions $f = f^{(L)} \circ f^{(L-1)} \circ \ldots \circ f^{(1)}$, then $k \leq \prod_{\ell=1}^{L} k_\ell$ denotes an upper bound on its Lipschitz constant (Szegedy *et al.*, 2014). However, it is possible that a tighter bound exists by considering the entire composition as a whole rather than each function in isolation. For a fully-connected layer $\boldsymbol{W}^{(\ell)}\boldsymbol{x} + \boldsymbol{b}^{(\ell)}$ with weight matrix $\boldsymbol{W}^{(\ell)}$ and bias $\boldsymbol{b}^{(\ell)}$, its smallest Lipschitz constant is given by the operator norm $\|\boldsymbol{W}^{(\ell)}\|$. For $\|\cdot\|_\infty$, this can be efficiently computed by the maximum absolute row sum. In practice, AutoLip and SeqLip are two effective methods to estimate the Lipschitz constant of any automatically differentiable function (Scaman & Virmuax, 2018). Furthermore, training a neural network with a bounded Lipschitz constant k can, *e.g.*, either be done by projected gradient descent via $\boldsymbol{W}^{(\ell)} \leftarrow \min\left(1, \frac{k}{\|\boldsymbol{W}^{(\ell)}\|}\right) \boldsymbol{W}^{(\ell)}$ (Gouk *et al.*, 2018), or by regularizing the change of outcome $\mathcal{L}_{\text{reg}} = \max\left(0, \frac{\|f(\boldsymbol{x}) - f(\boldsymbol{x}+\boldsymbol{\epsilon})\|}{\|\boldsymbol{\epsilon}\|} - k\right)$ for small input perturbations $\boldsymbol{\epsilon}$ (Usama & Chang, 2018).

With this, we can now show that the output error of a single historical-based GNN layer $f_{\boldsymbol{\theta}}^{(\ell)}$ has an upper bound that is solely dependent on the Lipschitz constants of the utilized message passing functions $\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}$ and $\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}$, as well as on the closeness of estimated inputs and the staleness of historical embeddings (Fey *et al.*, 2021):

Lemma 6. *Let $f_{\boldsymbol{\theta}}^{(\ell)}$ be a GNN layer, containing Lipschitz continuous $\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}$ and $\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}$ functions with Lipschitz constants k_1 and k_2, respectively. If, for all $v \in \mathcal{V}$, the inputs are close to the exact input,* i.e. $\|\tilde{\boldsymbol{h}}_v^{(\ell-1)} - \boldsymbol{h}_v^{(\ell-1)}\| \leq \delta$, *and the historical embeddings do not run too stale,* i.e. $\|\bar{\boldsymbol{h}}_v^{(\ell-1)} - \tilde{\boldsymbol{h}}_v^{(\ell-1)}\| \leq \epsilon$, *then the output error is bounded by*

$$\|\tilde{\boldsymbol{h}}_v^{(\ell)} - \boldsymbol{h}_v^{(\ell)}\| \leq \delta\, k_2 + (\delta + \epsilon)\, k_1\, k_2\, |\mathcal{N}(v)|.$$

Proof. By triangular inequality, it holds that $\|\bar{\boldsymbol{h}}_v^{(\ell-1)} - \boldsymbol{h}_v^{(\ell-1)}\| \leq \delta + \epsilon$. Furthermore,

$$\begin{aligned}\|\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{x}) - \textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{y})\| &\leq k_1\|\boldsymbol{x} - \boldsymbol{y}\| \quad \text{for all } \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^D \quad \text{and} \\ \|\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{x}) - \textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{y})\| &\leq k_2\|\boldsymbol{x} - \boldsymbol{y}\| \quad \text{for all } \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^D.\end{aligned}$$

Lipschitz constants for sum aggregation $\sum_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{x}$, mean aggregation $\frac{1}{|\mathcal{X}|}\sum_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{x}$ and max aggregation $\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{x}$ are given by $|\mathcal{X}|$, 1 and 1, respectively. Then,

$$\begin{aligned}&\|f_{\boldsymbol{\theta}}^{(\ell)}(\tilde{\boldsymbol{h}}^{(\ell-1)}, \{\!\{\bar{\boldsymbol{h}}_w^{(\ell-1)} : w \in \mathcal{N}(v)\}\!\}) - f_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{h}^{(\ell-1)}, \{\!\{\boldsymbol{h}_w^{(\ell-1)} : w \in \mathcal{N}(v)\}\!\}) \\ =&\|\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}(\tilde{\boldsymbol{h}}_v^{(\ell-1)}, \bigoplus_{w \in \mathcal{N}(v)} \textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}(\bar{\boldsymbol{h}}_w^{(\ell-1)})) - \textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{h}_v^{(\ell-1)}, \bigoplus_{w \in \mathcal{N}(v)} \textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{h}_w^{(\ell-1)}))\| \\ \leq& k_2\,(\delta + |\mathcal{N}(v)|\,(k_1\,(\delta + \epsilon))) = \delta\, k_2 + (\delta + \epsilon)\, k_1\, k_2\, |\mathcal{N}(v)|. \qquad \square\end{aligned}$$

Due to the behavior of Lipschitz constants in a series of function compositions, we obtain an upper bound that is dependent on k_1, k_2 and $|\mathcal{N}(v)|$, as well as dependent on the errors δ and ϵ of the inputs. Interestingly, sum aggregation, the most expressive aggregation function (Xu *et al.*, 2019c), introduces a factor of $|\mathcal{N}(v)|$ to the upper bound, while we can obtain a much tighter upper bound for mean or max aggregation. Next, we take a look at the final output error produced by a L-layer GNN (Fey *et al.*, 2021):

Theorem 7. *Let $f_{\boldsymbol{\theta}}^{(L)}$ be an L-layer GNN, containing only Lipschitz continuous $\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}$ and $\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}$ functions with Lipschitz constants k_1 and k_2, respectively. If, for all $v \in \mathcal{V}$ and all $\ell \in \{1, \ldots, L-1\}$, the historical embeddings do not run too stale,* i.e. $\|\bar{\boldsymbol{h}}_v^{(\ell)} - \tilde{\boldsymbol{h}}_v^{(\ell)}\| \leq \epsilon^{(\ell)}$, *then the final output error is bounded by*

$$\|\tilde{\boldsymbol{h}}_{v,j}^{(L)} - \boldsymbol{h}_{v,j}^{(L)}\| \leq \sum_{\ell=1}^{L-1} \epsilon^{(\ell)}\, k_1^{L-\ell}\, k_2^{L-\ell}\, |\mathcal{N}(v)|^{L-\ell}.$$

Proof. For layer $\ell = 1$, the inputs are exact, *i.e.* $\delta^{(0)} = \|\tilde{\boldsymbol{h}}_v^{(0)} - \boldsymbol{h}_v^{(0)}\| = 0$, and, as a result, the output is exact as well, *i.e.* $\delta^{(1)} = \|\tilde{\boldsymbol{h}}_v^{(1)} - \boldsymbol{h}_v^{(1)}\| = 0$. With $\|\bar{\boldsymbol{h}}_v^{(1)} - \tilde{\boldsymbol{h}}_v^{(1)}\| \leq \epsilon^{(1)}$, it directly follows via Lemma 6 that the approximation error of layer $\ell = 2$ is bounded by $\|\tilde{\boldsymbol{h}}_v^{(2)} - \boldsymbol{h}_v^{(2)}\| \leq \epsilon^{(1)}\, k_1\, k_2\, |\mathcal{N}(v)| = \delta^{(2)}$. Recursively replacing

$$\delta^{(\ell)} = \delta^{(\ell-1)}\, k_2 + (\delta^{(\ell-1)} + \epsilon^{(\ell-1)})\, k_1\, k_2\, |\mathcal{N}(v)|$$

in $\|\tilde{\boldsymbol{h}}_v^{(L)} - \boldsymbol{h}_v^{(L)}\| \leq \delta^{(L-1)}\, k_2 + (\delta^{(L-1)} + \epsilon^{(L-1)})\, k_1\, k_2\, |\mathcal{N}(v)|$ (*cf.* Lemma 6) yields

$$\|\tilde{\boldsymbol{h}}_v^{(L)} - \boldsymbol{h}_v^{(L)}\| \leq \sum_{\ell=1}^{L-1} \epsilon^{(\ell)}\, k_1^{L-\ell}\, k_2^{L-\ell}\, |\mathcal{N}(v)|^{L-\ell}. \qquad \square$$

Notably, the obtained upper bound of the approximation error of the final output does not longer depend on the closeness of estimations $\|\tilde{\boldsymbol{h}}_v^{(\ell)} - \boldsymbol{h}_v^{(\ell)}\| \leq \delta^{(\ell)}$, and is instead solely conditioned on the staleness of histories $\|\bar{\boldsymbol{h}}_v^{(\ell)} - \tilde{\boldsymbol{h}}_v^{(\ell)}\| \leq \epsilon^{(\ell)}$. However, it depends *exponentially* on the Lipschitz constants k_1 and k_2 as well as $|\mathcal{N}(v)|$ w.r.t. to GNN depth (Fey *et al.*, 2021). In particular, each additional layer introduces a less restrictive bound since the errors made in the first layers get immediately propagated to later ones, leading to potentially high inaccuracies for histories in deeper GNNs. Furthermore, Theorem 7 lets us immediately derive an upper error bound of gradients as well, *i.e.*

$$\|\nabla_{\boldsymbol{\theta}} \mathcal{L}(\tilde{\boldsymbol{h}}_v^{(L)}) - \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{h}_v^{(L)})\| \leq \lambda \|\tilde{\boldsymbol{h}}_v^{(L)} - \boldsymbol{h}_v^{(L)}\| \tag{5.6}$$

in case the loss criterion $\mathcal{L}$ is λ-Lipschitz continuous. As such, assuming low approximation errors, GAS encourages low variance and bias in the learning signal as well. However, parameters are not guaranteed to converge to the same optimum since we explicitly consider arbitrary GNNs solving non-convex problems (Cong *et al.*, 2020).

Although the exponential error bound given in Theorem 7 seems to be a negative result at first glance for the application of deep and highly non-linear GNNs, our theoretical analysis gives clear guidance for tightening. In practice, we have two degrees of freedom to tighten the upper bounds, leading to a lower approximation error in return: (1) Minimizing the staleness of historical embeddings, and (2) maximizing the closeness of estimated inputs to their exact values by controlling the Lipschitz constants of UPDATE and MESSAGE functions. In what follows, we derive a list of procedures to achieve these goals (Fey *et al.*, 2021).

5.3.2.1 Minimizing Inter-Connectivity between Mini-Batches. As formulated in Equation (5.4), the output embeddings of $f_{\boldsymbol{\theta}}^{(\ell+1)}$ are exact if $|\bigcup_{v \in \mathcal{B}} \mathcal{N}(v) \cup \{v\}| = |\mathcal{B}|$, *i.e.* all neighbors of nodes in $\mathcal{B}$ are as well part of $\mathcal{B}$. However, in practice, this can only be guaranteed for full-batch GNNs. Motivated by this observation, we aim to minimize the inter-connectivity between sampled mini-batches, *i.e.* $\min |\bigcup_{v \in \mathcal{B}} \mathcal{N}(v) \setminus \mathcal{B}|$, which minimizes the amount of history accesses, and increases closeness and reduces staleness in return.

In order to minimize the inter-connectivity between mini-batches, we make use of graph clustering techniques, *e.g.*, METIS (Karypis & Kumar, 1998; Dhillon *et al.*, 2007). Graph clustering algorithms aim to construct partitions over the nodes in a graph such that intra-links within clusters occur much more frequently than inter-links between different clusters. Intuitively, this results in a high chance that neighbors of a node are located in the same cluster. Notably, modern graph clustering methods are both fast and scalable with time complexities given by $\mathcal{O}(|\mathcal{E}|)$, and only need to be applied once, which leads to an unremarkable computational overhead in the pre-processing stage. In particular, we argue that the METIS clustering technique is highly scalable, as it is in the heart of many large-scale distributed graph storage layers such as (Zhu *et al.*, 2019b; Zheng *et al.*, 2020a) that are known scale to billion-sized graphs. Furthermore,

the additional overhead in the pre-processing stage is quickly compensated by an acceleration of training, since the number of neighbors outside of $\mathcal{B}$ is heavily reduced, and pushing information to the histories now leads to contiguous memory transfers.

Notably, utilizing graph clustering techniques for mini-batch selection was first introduced in the subgraph-wise sampling approach CLUSTER-GCN (Chiang *et al.*, 2019). CLUSTER-GCN leverages clustering in order to infer meaningful isolated subgraphs, while GAS aims to minimize the amount of history accesses. Furthermore, CLUSTER-GCN limits message passing to intra-connected nodes, and therefore ignores potentially useful information outside the current mini-batch. This inherently limits the model to learn from nodes nearby. In contrast, our GAS framework makes use of *all* available neighborhood data for aggregation, and therefore avoids this downside.

5.3.2.2 Enforcing Local Lipschitz Continuity. To guide our neural network in learning a function with controllable error, we can enforce its intermediate output layers $f_{\boldsymbol{\theta}}^{(\ell)}$ to be invariant to small input perturbations. In particular, following upon Usama & Chang (2018), we found it useful to apply the auxiliary loss

$$\mathcal{L}_{\text{reg}}^{(\ell)} = \| f_{\boldsymbol{\theta}}^{(\ell)}(\tilde{\boldsymbol{h}}_v^{(\ell-1)}) - f_{\boldsymbol{\theta}}^{(\ell)}(\tilde{\boldsymbol{h}}_v^{(\ell-1)} + \boldsymbol{\epsilon}) \|, \quad \boldsymbol{e} \sim \mathcal{B}_{\delta}(\mathbf{0}), \tag{5.7}$$

in highly non-linear message passing phases, *e.g.*, as utilized in GIN (Xu *et al.*, 2019c). Such regularization enforces equal outputs for small perturbations $\boldsymbol{\epsilon}$ inside closed balls of radius δ. Notably, we do not restrict $\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}$ and $\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}$ to separately model global k-Lipschitz continuous functions, but rather aim for *local* Lipschitz continuity at each $\boldsymbol{h}_v^{(\ell-1)}$ for $f_{\boldsymbol{\theta}}^{(\ell)}$ as a whole. For other message passing GNNs, *e.g.*, for GCN (Kipf & Welling, 2017), L_2 regularization is usually sufficient to ensure closeness of historical embeddings. Further, we found gradient clipping to be an effective method to restrict the parameters from changing too fast, regularizing history changes in return.

We evaluate GAS and the benefits of its individual techniques in Section 5.5.

5.3.3 Expressiveness of Historical-based Graph Neural Networks

The most powerful GNNs adhere to the same representational power as the WL test in distinguishing non-isomorphic structures, *i.e.* $\boldsymbol{h}_v^{(L)} \neq \boldsymbol{h}_w^{(L)}$ if the L-hop rooted subtrees around the nodes $v, w \in \mathcal{V}$ are distinguishable by the WL test (Xu *et al.*, 2019c; Morris *et al.*, 2019), *cf.* Section 3.4. In order to leverage such expressiveness, a GNN needs to be able to reason about structural differences across neighborhoods *during* training. However, as shown in Section 5.3, scalability techniques based on sub-sampling of edges (Section 5.2.1) will weaken such expressive power of message passing. In contrast, our GAS framework based on historical embeddings (Fey *et al.*, 2021) is leveraging *all* edges during neighborhood aggregation. Thus, a special interest lies in the question if historical-based GNNs can be as expressive as their full-batch equivalent. For this, a maximally powerful *and* scalable GNN needs to fulfill the following two requirements: (1) It needs to be as expressive as the WL test in distinguishing non-isomorphic structures, and (2) it needs to account for the approximation error $\|\bar{\boldsymbol{h}}_v^{(\ell)} - \boldsymbol{h}_v^{(\ell)}\|$ induced by the usage of historical embeddings, *cf.* Section 5.3.2. Since it is known that there exists a wide range of maximally powerful GNNs (Xu *et al.*, 2019c;

Morris *et al.*, 2019; Corso *et al.*, 2020), we can restrict our analysis to the latter question. Following upon Xu *et al.* (2019c), we focus on the case where input node features are from a countable set $\mathbb{P}^d \subset \mathbb{R}^d$ of bounded size. We first show that a single GNN layer utilizing historical embeddings can still distinguish non-equal inputs, given that they are sufficiently far apart from each other (Fey *et al.*, 2021).

Lemma 8. *Let* $f_{\boldsymbol{\theta}}^{(\ell)}$ *be a GNN layer and* $\{\{\boldsymbol{h}_v^{(\ell-1)} \colon v \in \mathcal{V}\}\}$ *be a countable multiset such that*

$$\|\boldsymbol{h}_v^{(\ell-1)} - \boldsymbol{h}_w^{(\ell-1)}\| > 2(\delta + \epsilon) \quad \textit{for all } v, w \in \mathcal{V} \quad \textit{where} \quad \boldsymbol{h}_v^{(\ell-1)} \neq \boldsymbol{h}_w^{(\ell-1)}.$$

If the inputs are close to the exact input, i.e. $\|\tilde{\boldsymbol{h}}_v^{(\ell-1)} - \boldsymbol{h}_v^{(\ell-1)}\| \leq \delta$, *and the historical embeddings do not run too stale,* i.e. $\|\bar{\boldsymbol{h}}_v^{(\ell-1)} - \tilde{\boldsymbol{h}}_v^{(\ell-1)}\| \leq \epsilon$, *then there exist* $\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}$ *and* $\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}$ *functions, such that*

$$\|f_{\boldsymbol{\theta}}^{(\ell)}(\tilde{\boldsymbol{h}}_v^{(\ell-1)}) - f_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{h}_v^{(\ell-1)})\| \leq \delta + \epsilon$$

and

$$\|f_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{h}_v^{(\ell-1)}) - f_{\boldsymbol{\theta}}^{(\ell)}(\boldsymbol{h}_w^{(\ell-1)})\| > 2(\delta + \epsilon + \lambda)$$

for all $v, w \in \mathcal{V}$ *where* $\boldsymbol{h}_v^{(\ell-1)} \neq \boldsymbol{h}_w^{(\ell-1)}$, *and all* $\lambda > 0$.

Proof. Define $\phi \colon \mathbb{R}^D \to \mathbb{R}^D$ as the Voronoi tessellation induced by the given multiset:

$$\phi(\boldsymbol{x}) = \boldsymbol{h}_v^{(\ell-1)} \quad \text{if} \quad \|\boldsymbol{x} - \boldsymbol{h}_v^{(\ell-1)}\| \leq \|\boldsymbol{x} - \boldsymbol{h}_w^{(\ell-1)}\| \text{ for all } w \in \mathcal{V} \text{ where } \boldsymbol{h}_v^{(\ell-1)} \neq \boldsymbol{h}_w^{(\ell-1)}.$$

Such Voronoi tessellation is guaranteed to exist. Therefore, it holds that

$$\|f_{\boldsymbol{\theta}}^{(\ell)}\big(\phi\big(\tilde{\boldsymbol{h}}_v^{(\ell-1)}\big)\big) - f_{\boldsymbol{\theta}}^{(\ell)}\big(\phi\big(\boldsymbol{h}_v^{(\ell-1)}\big)\big)\| = 0 \leq \delta + \epsilon \quad \text{for all } v \in \mathcal{V}.$$

Furthermore, we know that there exists $\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}$ and $\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}$ functions such that $f_{\boldsymbol{\theta}}^{(\ell)}$ is injective for all countable multisets (Zaheer *et al.*, 2017; Xu *et al.*, 2019c; Morris *et al.*, 2019; Maron *et al.*, 2019a). Since $\{\{\boldsymbol{h}_v^{(\ell-1)} \colon v \in \mathcal{V}\}\}$ is countable and $f_{\boldsymbol{\theta}}^{(\ell)}$ is injective, there exists a $\kappa > 0$ such that $\|f_{\boldsymbol{\theta}}^{(\ell)}\big(\phi\big(\boldsymbol{h}_v^{(\ell-1)}\big)\big) - f_{\boldsymbol{\theta}}^{(\ell)}\big(\phi\big(\boldsymbol{h}_w^{(\ell-1)}\big)\big)\| > \kappa$ for all $v, w \in \mathcal{V}$ where $\boldsymbol{h}_v^{(\ell-1)} \neq \boldsymbol{h}_w^{(\ell-1)}$. Due to the homogeneity of $\| \cdot \|$, it directly follows that there exists $\alpha > 0$ such that

$$\|\alpha \cdot f_{\boldsymbol{\theta}}^{(\ell)}\big(\phi\big(\boldsymbol{h}_v^{(\ell-1)}\big)\big) - \alpha \cdot f_{\boldsymbol{\theta}}^{(\ell)}\big(\phi\big(\boldsymbol{h}_w^{(\ell-1)}\big)\big)\| > \alpha \cdot \kappa \geq 2(\delta + \epsilon + \lambda)$$

for all $v, w \in \mathcal{V}$ where $\boldsymbol{h}_v^{(\ell-1)} \neq \boldsymbol{h}_w^{(\ell-1)}$, and all $\lambda > 0$. □

Informally, Lemma 8 tells us that if **(1)** exact input embeddings are sufficiently far apart from each other and **(2)** historical embeddings are sufficiently close to the exact embeddings, there exist historical-based GNN operators which can distinguish equal from non-equal inputs, *cf.* Figure 5.7. Key to the proof is that $(\delta+\epsilon)$-balls around exact inputs do not intersect each other and are therefore well separated. Furthermore, $f_{\boldsymbol{\theta}}^{(\ell)}$ needs to be able to ensure that exact outputs do not intersect each other as well, which is a necessary condition to mitigate the approximation effects of $\|\bar{\boldsymbol{h}}_v^{(\ell)} - \tilde{\boldsymbol{h}}_v^{(\ell)}\| \leq \epsilon^{(\ell)}$ induced in the upcoming layer, *e.g.*, by setting $\lambda = \epsilon^{(\ell)}$. Notably, we do not require $f_{\boldsymbol{\theta}}^{(\ell)}$ to model strict injectivity since it is sufficient for $f_{\boldsymbol{\theta}}^{(\ell)}$ to be $2(\delta + \epsilon)$-injective, in which $2(\delta + \epsilon)$ is used to denote the accuracy of discrimination (Seo *et al.*, 2019).

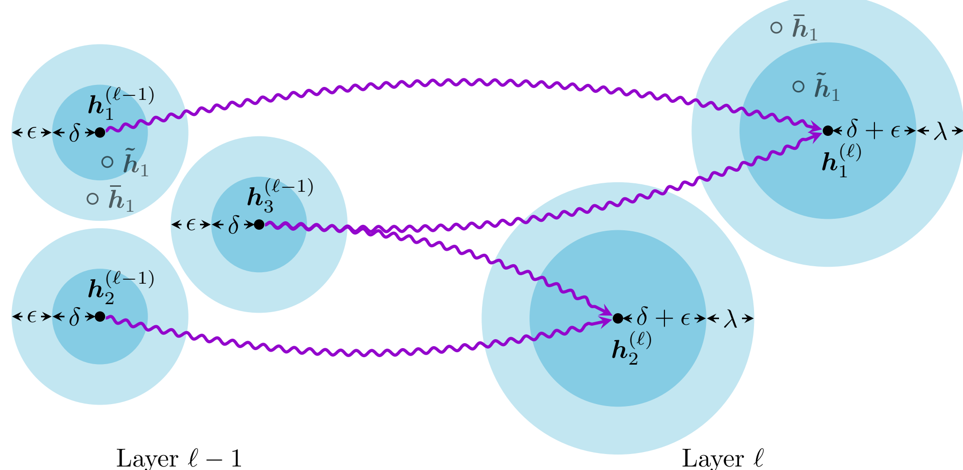


Figure 5.7: Requirements to obtain maximally expressive historical-based GNNs. Exact inputs need to be well separated while accounting for the approximation errors δ and ϵ caused by historical embeddings. Furthermore, exact outputs (denoted by aggregating along the ■ connections) need to be able to mitigate the approximation effects induced in the upcoming layer as well (Fey *et al.*, 2021).

Following Xu *et al.* (2019c), one can leverage MLPs to model and learn such Message and Update functions due to the universal approximation theorem (Hornik *et al.*, 1989; Hornik, 1991). However, the theory behind Lemma 8 holds for any maximally powerful GNN operator. Finally, we can use this insight to relate the expressiveness of scalable GNNs to the WL test color refinement procedure (Fey *et al.*, 2021):

Theorem 9. *Let $f_{\boldsymbol{\theta}}^{(L)}$ be an L-layer GNN in which all* $\textsc{Message}_{\boldsymbol{\theta}}^{(\ell)}$ *and* $\textsc{Update}_{\boldsymbol{\theta}}^{(\ell)}$ *functions fulfill the conditions of Lemma 8. Then, there exists a map $\phi\colon \mathbb{R}^D \to \Sigma$ so that $\phi(\tilde{\boldsymbol{h}}_v^{(L)}) = c_v^{(L)}$ for all $v \in \mathcal{V}$.*

Proof. Define $\phi\colon \mathbb{R}^D \to \Sigma$ as the Voronoi tessellation induced by the exact output set:

$$\phi(\boldsymbol{x}) = c_v^{(L)} \quad \text{if} \quad \|\boldsymbol{x} - \boldsymbol{h}_v^{(L)}\| \leq \|\boldsymbol{x} - \boldsymbol{h}_w^{(L)}\| \quad \text{for all } w \in \mathcal{V} \text{ where } \boldsymbol{h}_v^{(L)} \neq \boldsymbol{h}_w^{(L)}.$$

Since $f_{\boldsymbol{\theta}}^{(\ell)}$ can be maximally expressive, such Voronoi tessellation is guaranteed to exist (Xu *et al.*, 2019c; Morris *et al.*, 2019). Therefore, it is sufficient to show that there exists a $\delta^{(L)} > 0$ such that $\|\tilde{\boldsymbol{h}}_v^{(L)} - \boldsymbol{h}_v^{(L)}\| \leq \delta^{(L)}$ and $\|\boldsymbol{h}_v^{(L)} - \boldsymbol{h}_w^{(L)}\| > 2\delta^{(L)}$ for all $v, w \in \mathcal{V}$, $\boldsymbol{h}_v^{(L)} \neq \boldsymbol{h}_w^{(L)}$. For layer $\ell = 1$, the inputs are exact, and as a result $\|\tilde{\boldsymbol{h}}_v^{(1)} - \boldsymbol{h}_v^{(1)}\| = 0$. Due to Lemma 8, there exists message passing functions such that $\|\tilde{\boldsymbol{h}}_v^{(2)} - \boldsymbol{h}_v^{(2)}\| \leq \epsilon^{(1)}$. The next layer introduces an increased error, *i.e.* $\|\bar{\boldsymbol{h}}_v^{(2)} - \boldsymbol{h}_v^{(2)}\| \leq \epsilon^{(1)} + \epsilon^{(2)}$, and to compensate, we set $\lambda^{(2)} = \epsilon^{(2)}$ such that $\|\boldsymbol{h}_v^{(2)} - \boldsymbol{h}_w^{(2)}\| > 2\,(\epsilon^{(1)} + \epsilon^{(2)})$ for all $v, w \in \mathcal{V}$, $\boldsymbol{h}_v^{(L)} \neq \boldsymbol{h}_w^{(L)}$. By recursively applying Lemma 8 with $\lambda^{(\ell)} = \epsilon^{(\ell)}$, it immediately follows that $\|\tilde{\boldsymbol{h}}_v^{(L)} - \boldsymbol{h}_v^{(L)}\| \leq \sum_{\ell=1}^{L-1} \epsilon^{(\ell)}$, and $\|\tilde{\boldsymbol{h}}_v^{(L)} - \boldsymbol{h}_w^{(L)}\| > \sum_{\ell=1}^{L-1} 2\,\epsilon^{(\ell)}$ for all $v, w \in \mathcal{V}$, $\boldsymbol{h}_v^{(L)} \neq \boldsymbol{h}_w^{(L)}$. □

Theorem 9 extends the insights of Lemma 8 to multi-layer GNNs, and indicates that scalable GNNs using historical embeddings are still able to distinguish non-isomorphic structures (that are distinguishable by the WL test) directly during training, which is what makes reasoning about structural properties possible. As such, our GAS framework is the first scalable solution that can provably maintain the expressive power of the underlying GNN (Fey *et al.*, 2021).

Notably, recent proposals such as DropEdge (Rong *et al.*, 2020b) are still applicable for data augmentation and message reduction. However, through the given theoretical analysis, we disentangle scalability and expressiveness from regularization via edge dropping. Furthermore, our approach is *orthogonal* to many methodological advancements in the field of Graph Representation Learning, such as unifying GNNs and label propagation (Shi *et al.*, 2020b), graph diffusion (Klicpera *et al.*, 2019b), or random wiring (Valsesia *et al.*, 2020). Although our method is focused around node-level tasks, our work is technically able to scale GNNs for edge-level and graph-level tasks as well. Furthermore, while our method tackles the task of scaling the training and inference phase of GNNs on large-scale graphs by utilizing only a single GPU, only minor modifications are necessary to fuse GAS into a distributed training algorithm (Jia *et al.*, 2020; Ma *et al.*, 2019; Zhu *et al.*, 2016; Tripathy *et al.*, 2020; Wan *et al.*, 2020; Angerd *et al.*, 2020; Zheng *et al.*, 2020a). We evaluate the benefits of applying our GAS framework on large-scale graphs in Section 5.5.

5.4 The Open Graph Benchmark Datasets for Large-Scale Graph Learning

Enabling effective and efficient machine learning algorithms over large-scale graph data, *e.g.*, graphs with billions of edges, can have a huge impact on both industrial and scientific applications (Hu *et al.*, 2021b). However, applying and evaluating GNNs on large-scale graphs (within their respective scalability technique such as within our GAS framework, *cf.* Section 5.3) requires the presence of realistic and large-scale graph benchmark datasets. Historically, high-quality and large-scale datasets have played significant roles in advancing machine learning research, as exemplified by ImageNet (Deng *et al.*, 2009) and MS COCO (Lin *et al.*, 2014) in computer vision, the GLUE Benchmark (Wang *et al.*, 2018a) and SQuAD (Rajpurkar *et al.*, 2016) in natural language processing, and LibriSpeech (Panayotov *et al.*, 2015) and CHiME (Barker *et al.*, 2015) in speech processing. However, most of the frequently-used graph datasets in research (Sen *et al.*, 2008; Yang *et al.*, 2016; Yanardag & Vishwanathan, 2015; Morris *et al.*, 2020a; Bordes *et al.*, 2013) are extremely small compared to graphs found in real applications (Wang *et al.*, 2020; Ying *et al.*, 2018a; Wu *et al.*, 2018; Husain *et al.*, 2019; Bhatia *et al.*, 2016; Vrandečić & Krötzsch, 2014b), leading to non-scalable solutions, and unstable and nearly statistically identical performance (Dwivedi *et al.*, 2020; Shchur *et al.*, 2018; Errica *et al.*, 2020), *cf.* Section 5.2.2.

In order to overcome the lack of large-scale graph benchmark datasets, we propose the *Open Graph Benchmark* (*OGB*) (Hu *et al.*, 2020a, 2021b), which includes a diverse set of challenging and realistic benchmark datasets to facilitate scalable, robust, and reproducible Graph Representation Learning research. OGB datasets are orders of magnitude larger than existing ones, encompass multiple important tasks, and cover a diverse range of domains, ranging from social and information networks to biological

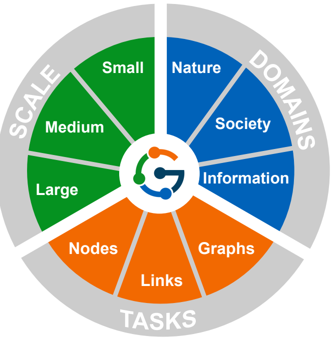


Figure 5.8: OGB provides datasets that are diverse in scale (small, medium, large)**, domains** (nature, society, information) **and tasks** (nodes, links, graphs) (Hu *et al.*, 2020a)**.**

networks, molecular graphs, source code ASTs, and KGs. Here, we first present the design principles of OGB in Section 5.4.1 before introducing our set of benchmark datasets in Section 5.4.2 in detail.

5.4.1 Benchmark Design Principles

The premise of OGB is to support and catalyze research in Graph Representation Learning by developing a diverse set of challenging and realistic benchmark datasets that cover a variety of real-world applications and span several important domains (Hu *et al.*, 2020a, 2021b). As illustrated in Figure 5.8, OGB datasets are designed to have the following three characteristics:

- **Large-scale:** OGB datasets are orders of magnitude larger than existing benchmarks (Sen *et al.*, 2008; Yang *et al.*, 2016; Yanardag & Vishwanathan, 2015; Morris *et al.*, 2020a; Bordes *et al.*, 2013) and can be categorized into three different scales: small, medium, and large. Even the "small" OGB graphs are designed to have more than 100 thousand nodes or more than 1 million edges, but are still small enough to fit into the memory of a single GPU, making them suitable for testing computationally intensive algorithms. Additionally, OGB introduces "medium" (more than 1 million nodes or more than 10 million edges) and "large" (on the order of 100 million nodes or 1 billion edges) datasets, which can facilitate the development of scalable models based on mini-batching and distributed training (Hu *et al.*, 2020a).

- **Diverse domains:** OGB datasets aim to include graphs that are representative of a wide range of domains. The broad coverage of domains in OGB empowers the development and demonstration of general-purpose models, and can be used to distinguish them from domain-specific techniques. Furthermore, for each dataset, OGB adopts domain-specific data splits (*e.g.*, based on time, species, molecular structure, GitHub project, ...) that are more realistic and meaningful than conventional random splits which may lead to overly optimistic performance results (Lohr, 2009). In particular, such realistic domain-specific data

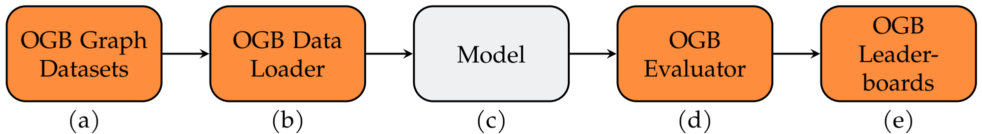


Figure 5.9: Overview of the OGB pipeline: (a) OGB provides realistic graph benchmark datasets that cover different prediction tasks (node, link, graph), are from diverse application domains, and are at different scales. (b) OGB fully automates dataset processing and splitting. That is, the OGB data loaders automatically download and process graphs, and provide graph objects that are compatible with PyTorch (Paszke *et al.*, 2019) and its associated graph libraries such as our PyTorch Geometric library (Fey & Lenssen, 2019) or the Deep Graph Library (Wang *et al.*, 2019b). (c) After an machine learning model is developed, (d) OGB evaluates the model in a dataset-dependent manner, and outputs the model performance appropriate for the task at hand. Finally, (e) OGB provides public leaderboards to keep track of recent advances (Hu *et al.*, 2020a).

splits help to stress test the out-of-distribution generalization performance of the underlying machine learning model (Hu *et al.*, 2020a).

- **Multiple task categories:** Besides data diversity, OGB supports three categories of fundamental graph machine learning tasks, *i.e.* node, link, and graph property predictions, each of which requires the models to make predictions at different levels of graphs, *i.e.*, at the level of a node, link, and entire graph, respectively (Hu *et al.*, 2020a).

Finally, OGB presents an auomated end-to-end graph machine learning pipeline that simplifies and standardizes the process of graph data loading, experimental setup, and model evaluation, in a similar spirit to OpenML (Vanschoren *et al.*, 2013; Feurer *et al.*, 2019), *cf.* Figure 5.9 (Hu *et al.*, 2020a). Specifically, given an OGB dataset (a), the end-user can focus on developing their graph machine learning model (c) by using the OGB data loaders (b) and evaluators (d), both of which are provided in our OGB Python package.[2] In particular, the OGB data loaders automatically download and process graphs, provide graph objects that are compatible with PyTorch (Paszke *et al.*, 2019) and its associated graph libraries such as our PyTorch Geometric library (Fey & Lenssen, 2019) or the Deep Graph Library (Wang *et al.*, 2019b). In addition, OGB also hosts a public leaderboard (e) for publicizing state-of-the-art, reproducible graph machine learning research. For this, individual experiments should be repeated 10 times using different random seeds, for which the mean and standard deviation of test results corresponding to the best validation results should be reported. The documentation, example scripts and public leaderboards are condensed on our OGB website[3] (Hu *et al.*, 2020a).

[2]OGB Python package: `https://github.com/snap-stanford/ogb` (last access: August 25, 2022)
[3]OGB website: `https://ogb.stanford.edu` (last access: August 25, 2022)

Domain	Nature	Society	Information
	Node property prediction		
Small		`arxiv`	
Medium	`proteins`	`products`	`mag`
Large		`papers100M`	`mag240M`
	Link property prediction		
Domain	Nature	Society	Information
Small	`ddi`	`collab`	`biokg`
Medium	`ppa`	`citation`	`wikikg`
Large			`wikikg90M`
	Graph property prediction		
Domain	Nature	Society	Information
Small	`molhiv`		
Medium	`molpcba` / `ppa`		`code`
Large	`pcqm4M`		

Table 5.2: Overview of available OGB datasets (denoted by ■). The nature domain includes biological networks and molecular graphs, the society domain includes academic graphs and e-commerce networks, and the information domain includes knowledge graphs. More datasets will be added in the future to increase the coverage (denoted by ■) (Hu *et al.*, 2020a, 2021b).

5.4.2 Realistic Datasets for Diverse Task Categories

OGB aims to provide meaningful datasets that try to help solving real-world problems in a standardized and reproducible manner. The available datasets in OGB are categorized in Table 5.2 according to their task categories, application domains and scales. Currently, OGB includes 18 diverse graph datasets, with at least five datasets for each task category. All the datasets are constructed by ourselves, except for `products`, `molpcba` and `molhiv`, whose graphs are adopted from Chiang *et al.* (2019); Wu *et al.* (2018). For these datasets, we resolve critical issues of the existing data splits by providing more meaningful and standardized splitting schemes (Hu *et al.*, 2020a).

Further, we highlight the diversity of our graph datasets in Table 5.3. Importantly, we obverse a diversity in graph structure beyond the diversity in dataset scale. For example, we see that biology-related graphs, *e.g.*, `proteins`, `ddi` or `ppa`, are typically much denser than the social and information networks. These differences in graph structure result in inherent differences in how information propagates in graphs, which can significantly affect the behavior of graph machine learning models (Xu *et al.*, 2018). Datasets are also diverse in respect to the initially given input feature representations, *i.e.* datasets may include no initial features at all, either node features or edge features, or even both, and may also include both homogeneous and heterogeneous graph information. Furthermore, it is noteworthy to highlight the diversity of graph sizes for the graph property prediction datasets, ranging from small molecular graphs (`molhiv`, `molpcba` and `pcqm4M`), to medium-sized source code ASTs (`code`), up to large and dense protein-protein association subgraphs (`ppa`), which directly originates from the diverse utilized application domains (Hu *et al.*, 2020a).

	Name	#Graphs	Average #Nodes	Average #Edges	Node Features	Edge Features	Split Scheme
Node	proteins	1	132,534	39,561,252	—	✔	Species
	arxiv	1	169,343	1,166,243	✔	—	Time
	products	1	2,449,029	61,859,140	✔	—	Sales rank
	mag	1	1,939,743	21,111,007	✔	✔	Time
	papers100M	1	111,059,956	1,615,685,872	✔	—	Time
	mag240M	1	244,160,499	1,728,364,232	✔	✔	Time
Link	ddi	1	4,267	1,334,889	—	—	Protein target
	biokg	1	93,773	5,088,434	—	✔	Random
	collab	1	235,868	1,285,465	✔	✔	Time
	ppa	1	576,289	30,326,273	✔	—	Throughput
	wikikg	1	2,500,604	17,137,181	—	✔	Time
	citation	1	2,927,963	30,561,187	✔	—	Time
	wikikg90M	1	87,143,637	502,220,369	✔	✔	Time
Graph	molhiv	41,127	25.5	27.5	✔	✔	Scaffold
	ppa	158,100	243.4	2,266.1	—	✔	Species
	molpcba	437,929	26.0	28.1	✔	✔	Scaffold
	code	452,741	125.2	124.2	✔	✔	Project
	pcqm4M	3,803,453	26.5	29.1	✔	✔	Scaffold

Table 5.3: Statistics of available OGB datasets (Hu *et al.*, 2020a, 2021b)**.**

In the subsequent sections, we briefly introduce the available OGB datasets, while more detailed information about each dataset is available in Appendix A.1.

5.4.2.1 Node Property Prediction Datasets. OGB provides six datasets for predicting the properties of individiual nodes. Specifically, `products` is an Amazon products co-purchasing network Chiang *et al.* (2019), and the task is to predict the category of a given product. Notably, we use the *sales ranking* to split nodes into training, validation and test sets, which closely matches the real-world application where manual labeling is prioritized to important nodes, and machine learning models are subsequently used to make predictions on less important ones (Hu *et al.*, 2020a).

The `arxiv`, `papers100M`, `mag` and `mag240M` datasets are extracted from the Microsoft Academic Graph (MAG) (Wang *et al.*, 2020), utilizing different scales, tasks, and include both homogeneous and heterogeneous graph information. Specifically, `arxiv` and `papers100M` denote homogeneous paper citation networks. In contrast, `mag` and `mag240M` represent the MAG as a heterogeneous network (*e.g.*, with additional author or institution information), either as a subset or as its full set, respectively. The tasks are to predict the subject areas of arXiv papers (manually labeled by the paper's authors and arXiv moderators) and their venues (conference or journal).

Lastly, the `proteins` dataset denotes a protein-protein association network in which nodes represent proteins, and edges indicate different types of biologically meaning-

ful associations between proteins, *e.g.*, physical interactions, co-expression or homology (Szklarczyk *et al.*, 2019; Consortium, 2018; Hu *et al.*, 2020a). The task is to predict the presence of certain protein functions.

5.4.2.2 Link Property Prediction Datasets. OGB provides seven link property prediction datasets, adopted from diverse application domains, including biological, academic datasets as well as Knowledge Graphs (KGs). In particular, we provide the two biological datasets `ppa` and `ddi`, which represent protein-protein association and drug-drug interaction networks, respectively (Szklarczyk *et al.*, 2019; Wishart *et al.*, 2018). In `ppa`, the task is to predict new association edges given existing associations. In the `ddi` dataset, each node represents an FDA-approved or experimental drug. Edges represent interactions between drugs and can be interpreted as a phenomenon where the joint effect of taking the two drugs together is considerably different from the expected effect in which drugs act independently of each other (Hu *et al.*, 2020a). The task is to predict unknown drug-drug interactions.

Furthermore, the two academic datasets `collab` and `citation` describe author collaboration and paper citation networks, extracted from MAG (Wang *et al.*, 2020). The tasks are to predict author collaboration relationships given past collaborations and to infer missing citations given existing citations.

In addition, OGB provides three KGs named `biokg`, `wikikg` and `wikikg90M`, utilizing different tasks and scales (Hu *et al.*, 2020a, 2021b). As large KGs are known to be far from complete (Min *et al.*, 2013), the general task is to impute missing triplets (head, relation, tail). The `biokg` dataset represents a KG curated from a large number of biomedical data repositories with five types of entities and 51 types of relations. The `wikikg` and `wikikg90M` datasets describe KGs extracted from the Wikidata knowledge base (Vrandečić & Krötzsch, 2014a), and contain sets of triplets capturing the different types of relations between entities in the world, *e.g.*, "*Hinton* $\xrightarrow{citizen\ of}$ *Canada*".

5.4.2.3 Graph Property Prediction Datasets. OGB provides five datasets for predicting the properties of entire graphs or subgraphs, adopted from three distinct application domains. In particular, we provide the three molecular graph datasets `molhiv`, `molpcba` and `pcqm4M` for tackling the task of molecular property prediction (Hu *et al.*, 2020a, 2021b), *e.g.*, whether a molecule inhibits HIV virus replication or not. We adapt the *scaffold splitting* procedure for all molecular graph learning tasks, which splits the molecules based on their two-dimensional structural frameworks (Wu *et al.*, 2018; Yang *et al.*, 2019; Hu *et al.*, 2020b; Ishiguro *et al.*, 2019; Rong *et al.*, 2020a).

The `ppa` dataset contains a set of protein association neighborhoods extracted from the protein-protein association networks of 1,581 different species (Szklarczyk *et al.*, 2019; Hu *et al.*, 2020a) that cover 37 broad taxonomic groups, *e.g.*, mammals, bacterial families and archaeans (Hug *et al.*, 2016). The task is to predict the taxonomic group from which the graph originates from.

Lastly, the `code` dataset is a collection of ASTs obtained from approximately 450K Python functions extracted from a total of 13,587 different GitHub repositories (Husain *et al.*, 2019; Hu *et al.*, 2020a). Given the input arguments and body of a Python method represented by an AST, the task is to predict its method name as a set of subtokens ("code summarization") (Allamanis *et al.*, 2016, 2017; Alon *et al.*, 2018, 2019).

5.5 Evaluation

In this section, we perform an extensive benchmark analysis for each dataset in the Open Graph Benchmark suite, using representative graph-based machine learning models that utilize a diverse range of GNN scalability techniques (Section 5.5.1). We are further interested in how our GAS framework compares empirically against related scalable methods, and achieves favorable performance with deep and expressive GNNs on large-scale graphs (Section 5.5.2).

5.5.1 Open Graph Benchmark Analysis

We present an initial benchmark analysis for each task category and each dataset included in OGB (Hu *et al.*, 2020a, 2021b). For this, we are using representative node embedding models, GNNs, as well as scalable mini-batch-based GNNs (whenever applicable). We discuss our findings and highlight research challenges and opportunities in scaling models to large graphs, and improving the out-of-distribution generalization under realistic data split scenarios. We repeat each experiment 10 times using different random seeds, and report the mean and standard deviation of all training and test results corresponding to the best validation results (except for the experiments on the Open Graph Benchmark Large-Scale Challenge (OGB-LSC) datasets (Hu *et al.*, 2021b)). All experiments utilize the PyTorch Geometric library (Fey & Lenssen, 2019). The code for reproducing all results is available on GitHub[4] and is meant as a starting point to accelerate further research on the proposed datasets. The code further contains all specific details regarding model architectures and hyperparameter configurations.

5.5.1.1 Node Property Prediction. For the node-level datasets in OGB, we consider the following representative models as baselines (Hu *et al.*, 2020a, 2021b):

- **MLP**: A graph-agnostic MLP that uses raw node features directly as input.
- **Node2Vec**: An MLP that uses the concatenation of raw node features and the embeddings from a Node2Vec model (Grover & Leskovec, 2016) as input.
- **SGC**: The simplified Graph Neural Network (Wu *et al.*, 2019a) that decouples predictions from propagations.
- **GNN**: A full-batch Graph Neural Network (Kipf & Welling, 2017; Hamilton *et al.*, 2017; Veličković *et al.*, 2018).
- **GraphSAGE**: A GNN utilizing a node-wise sampling method (Hamilton *et al.*, 2017) that samples neighborhoods recursively.
- **Cluster-GCN**: A GNN utilizing a subgraph-wise sampling method (Chiang *et al.*, 2019) that pre-partitions the graph via graph clustering.
- **GraphSAINT**: A GNN utilizing a subgraph-wise sampling method (Zeng *et al.*, 2020b) that samples subgraphs via a random walk sampler.

[4] Code for OGB: `https://github.com/snap-stanford/ogb` (last access: August 25, 2022)

Method	Dataset					
	`products`	`arxiv`	`papers100M`	`mag`	`mag240M`	`proteins`
#Nodes	2.4M	169K	111M	1.9M	244M	132K
#Edges	62M	1.2M	1.6B	21M	1.7B	40M
Metric	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy	ROC-AUC
MLP	61.06±0.08	55.50±0.23	47.24±0.31	26.92±0.26	52.73	72.04±0.48
Node2Vec	72.49±0.10	70.07±0.13	—	35.44±0.36	—	68.81±0.65
SGC	—	—	**63.29**±0.19	—	65.29	—
GNN	78.50±0.14	**71.74**±0.29	—	39.77±0.46	—	**77.68**±0.20
GraphSAGE	78.70±0.36	—	—	**46.78**±0.67	**69.42**	—
Cluster-GCN	78.97±0.33	—	—	37.32±0.37	—	—
GraphSAINT	**79.08**±0.24	—	—	46.51±0.22	—	—

Table 5.4: Model performance on all node-level OGB datasets, using a diverse set of representative graph-based machine learning models.

For training the full-batch GNN models even on slightly larger graphs, we utilize a NVIDIA Quadro RTX 8000 with 48GB of memory. All other models fit into common GPU memory sizes of 11 GB. All models are trained with a fixed hidden dimensionality of 256, a tuned number of two or three layers, and a tuned dropout ratio $\in \{0.0, 0.5\}$ (Srivastava *et al.*, 2014). For learning in the heterogeneous graph datasets `mag` and `mag240M`, we make use of heterogeneous GNNs that learn distinct weights for each individual relation type (Schlichtkrull *et al.*, 2018). We obtain the input features of featureless node types by averaging features from their direct neighbor types. Furthermore, for the node embedding model Node2Vec, we adopt the Metapath2Vec algorithm as it is specifically designed for heterogeneous graphs. For each relation, *e.g.*, an author "writes" a paper, the reverse relation is added as well, *e.g.*, a paper "is written by" an author, in order to allow for a bidirectional message passing flow in GNNs.

Test performance on all node-level datasets included in OGB are presented in Table 5.4. Overall, all GNN-based models outperform the graph-agnostic MLP as well as the Node2Vec baselines. Notably, mini-batch processed GNNs via GraphSAGE, Cluster-GCN or GraphSAINT sampling techniques mostly outperform the full-batch processed GNNs, indicating that stochastic optimization indeed may improve generalization (Bottoue & Bousquet, 2007).

For example, on `products`, we see that the highest test performances are attained by GNNs, while the MLP baseline that solely relies on a product's description is not sufficient for accurately predicting the category of a product. Notably, even with GNNs, we observe a huge generalization gap between training and test performance (≈ 16 percentage points), which can be explained by the differing node distributions across splits, *cf.* Figure A.1. This is in stark contrast to a conventional *random* split. Even with the same split ratio, we find GNNs to be able to achieve 88.20±0.08% test accuracy (only ≈ 1 percentage point of generalization gap). This result indicates that the realistic split is indeed more challenging and offers important opportunities to improve out-of-distribution generalization (Hu *et al.*, 2020a).

Our initial benchmark analysis on `papers100M` focuses on the two simple models MLP and SGC (Wu *et al.*, 2019a), as most existing models have difficulty handling such a gigantic graph. In particular, we observe severe under-fitting of SGC (67.54±0.43 training accuracy), indicating that using more expressive GNNs is likely to improve both training and test accuracy, going beyond the simple pre-processing of node features (Hu *et al.*, 2020a).

On the heterogeneous graphs `mag` and `mag240M`, we observe that all models that do not utilize heterogeneous graph information perform much worse than their heterogeneous counterparts. Notably, the scalability techniques utilized in `mag` give surprisingly promising results, outperforming the full-batch GNN by a large margin. This is likely due to the regularization effect of the noise induced by mini-batch sampling and edge dropout (Rong *et al.*, 2020b). In contrast, Cluster-GCN (Chiang *et al.*, 2019) gives worse performance than its full-batch variant, indicating that the bias introduced by the pre-computed partitioning has a negative effect on the model's performance. This can be also observed by its highly over-fitting training performance (79.65±4.12%). On `mag240M`, we additionally evaluate the node-wise sampling approach GraphSAGE (Hamilton *et al.*, 2017) using a heterogeneous Graph Attention Network (GAT) variant (Veličković *et al.*, 2018; Schlichtkrull *et al.*, 2018), which yields significant improvements over the remaining baselines. In particular, the advanced attention-guided aggregation is favorable over alternative GNNs that perform mean aggregation (Hu *et al.*, 2021b). Overall, the `mag240M` experiments highlight the benefits of developing and evaluating advanced expressive models on larger scale. However, due to the exponential neighbor expansion in node-wise sampling, we were only able to train GNNs up to two layers. In contrast, the winning solutions of the OGB-LSC[5] utilize much deeper GNNs, pushing the final test performance to 75.49% (Hu *et al.*, 2021b).

On the `proteins` dataset, a simple graph-agnostic MLP performs surprisingly well. As a result, this dataset presents an interesting research question of how to utilize edge features in a more sophisticated way than just via naive averaging, *e.g.*, by the usage of attention or by treating the graph as a multi-relational graph (as there are 8 different association types between proteins). The challenge is to handle the huge number of edge features efficiently on GPUs, which might require clever graph partitioning based on the edge weights (Hu *et al.*, 2020a).

5.5.1.2 Link Property Prediction. For the link-level datasets in OGB, we consider the following representative models as baselines (Hu *et al.*, 2020a, 2021b):

- **MLP**: A graph-agnostic MLP that uses raw node features directly as input.
- **Node2Vec**: An MLP that uses the concatenation of raw node features and the embeddings from a Node2Vec model (Grover & Leskovec, 2016) as input.
- **MatrixFactorization**: Random node embeddings are assigned to nodes and learned in an end-to-end manner.
- **GNN**: A full-batch Graph Neural Network (Kipf & Welling, 2017; Hamilton *et al.*, 2017).
- **GraphSAGE**: A GNN utilizing a node-wise sampling method (Hamilton *et al.*, 2017) that samples neighborhoods recursively.

[5] `https://ogb.stanford.edu/kddcup2021/results` (last access: August 25, 2022)

Method	Dataset			
	ddi	ppa	collab	citation
#Nodes	4.3K	576K	254K	2.9M
#Edges	1.3M	30M	1.3M	30.6M
Metric	Hits@20	Hits@100	Hits@50	MRR
MLP	—	0.46±0.00	19.27±1.29	0.2895±0.0014
Node2Vec	23.26±1.35	22.26±0.83	48.88±0.54	0.6141±0.0011
MatrixFactorization	13.68±4.75	**32.29**±0.94	38.86±0.29	0.5186±0.0443
GNN	**53.90**±4.74	18.67±1.32	**54.63**±1.12	**0.8474**±0.0021
GraphSAGE	—	—	—	0.8044±0.0010
Cluster-GCN	—	—	—	0.8004±0.0025
GraphSAINT	—	—	—	0.7985±0.0040

Table 5.5: Model performance on all link-level OGB datasets (excluding KGs), using a diverse set of representative graph-based machine learning models.

- **Cluster-GCN**: A GNN utilizing a subgraph-wise sampling method (Chiang *et al.*, 2019) that pre-partitions the graph via graph clustering.
- **GraphSAINT**: A GNN utilizing a subgraph-wise sampling method (Zeng *et al.*, 2020b) that samples subgraphs via a random walk sampler.

Similar to the node property prediction experiments, mini-batch training of GNNs is only applied for graph datasets where full-batch GNN training is not feasible. All other models fit into common GPU memory sizes of 11 GB. We nonetheless report full-batch GNN performance utilizing a NVIDIA Quadro RTX 8000 with 48GB of memory. All models are trained with a fixed hidden dimensionality of 256, a tuned number of two or three layers, and a tuned dropout ratio $\in \{0.0, 0.5\}$ (Srivastava *et al.*, 2014). After computing node embeddings through a GNN, edge features are obtained by using the Hadamard operator $\odot$ between pair-wise node embeddings, which are then inputted into an MLP for the final prediction. During training, we randomly sample edges and use them as negative examples. We use the same number of negative edges as there are positive edges (Hu *et al.*, 2020a).

The obtained test performance on the OGB link-level datasets (excluding KGs) are given in Table 5.5. Overall, the GNN-based models outperform related methods such as Node2Vec or MatrixFactorization, while there also exist datasets where GNNs perform poorly, *e.g.*, on the ppa dataset (Hu *et al.*, 2020a).

As ddi does not contain any node features, we omit the graph-agnostic MLP baseline for this experiment. For the GNN model, node features are represented as distinct embeddings and learned in an end-to-end manner together with the GNN parameters, similar to the MatrixFactorization approach. Interestingly, both the GNN model and the MatrixFactorization approach achieve significantly higher training results than Node2Vec. However, only the GNN model is able to transfer this performance to the test set, suggesting that relational information is crucial to allow the model to generalize to unseen interactions. Notably, most of the models show high performance variance, which can be partly attributed to the dense nature of the graph and the challenging data split (Hu *et al.*, 2020a). We further perform a conventional random split

of edges, where we find the GNN to be able to achieve 80.88±2.42% test Hits@20. This indicates that the protein-target split is indeed more challenging than the conventional random split (Hu *et al.*, 2020a).

On the ppa dataset, both the Node2Vec and MatrixFactorization approaches heavily outperform the GNN baseline. The poor training performance of GNNs suggests that positional information, which cannot be captured by GNNs alone (You *et al.*, 2019), might be crucial to obtain meaningful node embeddings. On the other hand, we see that MatrixFactorization, which learns a distinct embedding for each node and hence can express positional information of nodes as well, is indeed able to achieve promising performance. However, it heavily over-fits on the training data (81.65±9.15% training Hits@100) which encourages the development of new research ideas to close this gap, *e.g.*, by injecting positional information into GNNs or by developing more sophisticated negative sampling techniques (Hu *et al.*, 2020a).

On `collab`, most methods perform reasonable well, with the GNN being the winner since it can handle the dynamic multi-graph scenario better than the remaining approaches. In particular, for the GNN model, we additionally incorporate the most recent edges (*i.e.* validation edges) as input to the models at test time (which is not possible for the remaining models). This increases test performance of the GNN model significantly, in comparison to the 48.10±0.81% test Hits@50 obtained when not incorporating validation edges during inference time. However, a promising direction to further increase the performance is to treat edges at different timestamps differently, as recent collaborations may be more indicative about the future collaborations than the past ones (Hu *et al.*, 2020a). Furthermore, as indicated by the good performance of the Node2Vec model, it is also fruitful to incorporate positional information into the GNN model. This can be explained by the fact that positional information, *i.e.* past collaborations, is typically a much more indicative feature for predicting future collaboration than what GNNs are able to capture, *i.e.* the same research interests (Hu *et al.*, 2020a).

Furthermore, on the `citation` dataset, the GNN model achieves the best results, followed by MatrixFactorization and node2Vec. However, the GNN uses full-batch training; thus, it is not scalable and requires more than 40GB of GPU memory to train, which is intractable on most of the GPUs available today. Hence, we also experiment with the scalable mini-batch training techniques GraphSAGE, Cluster-GCN and GraphSAINT. Interestingly, we see that these techniques give worse performance than their full-batch counterpart, which is in contrast to the node classification datasets (*e.g.*, `products` and `mag`), where the mini-batch-based models give stronger generalization performances. This limitation presents a unique challenge for applying the mini-batch techniques to link prediction, differently from those pertaining to node prediction (Hu *et al.*, 2020a).

For our experiments on the KG datasets, we consider the following KG embedding models (Hu *et al.*, 2020a, 2021b):

- **TransE**: The translation-based KG embedding model (Bordes *et al.*, 2013).
- **DistMult**: The multiplication-based KG embedding model (Yang *et al.*, 2015).
- **ComplEx**: The complex-valued KG embedding model (Trouillon *et al.*, 2016).
- **RotatE**: The rotation-based KG embedding model (Sun *et al.*, 2019).

Method	Dataset		
	`biokg`	`wikikg`	`wikigk90M`
#Nodes	94K	2.5M	87.1M
#Edges	5.1M	17.1M	502.2M
Metric	MRR	MRR	MRR
TransE	0.7452±0.0004	0.4256±0.0030	0.8548
DistMult	0.8043±0.0003	0.3729±0.0045	**0.8637**
ComplEx	**0.8095**±0.0007	0.4027±0.0027	—
RotatE	0.7989±0.0004	**0.4332**±0.0025	—

Table 5.6: Model performance on all KG datasets in OGB, using a diverse set of representative KG embedding models.

For KGs with many entities and relations, the embedding dimensionality can be limited by the available GPU memory, as the embeddings need to be loaded into GPU all at once. We therefore choose the dimensionality such that training can be performed on a fixed-budget of GPU memory. Our training procedure follows the one from Sun *et al.* (2019), where we perform negative sampling and use the margin-based logistic loss as our loss function (Hu *et al.*, 2020a). Final test performance of these embedding models on the KGs in OGB is given in Table 5.6.

Among the four models, ComplEx achieves the best test MRR on `biokg`, while TransE gives significantly worse performance compared to the other models. The worse performance of TransE can be explained by the fact that it cannot model symmetric relations (Trouillon *et al.*, 2016) that are prevalent in this dataset, *e.g.*, protein-protein and drug-drug relations are all symmetric. A promising direction is to develop more specialized methods to handle this heterogeneous knowledge graph, in which multiple node types exist (Hu *et al.*, 2020a).

On `wikikg`, all four models show similar performance. Nevertheless, the extremely low test MRR suggests that our realistic KG completion dataset is highly non-trivial. It presents a realistic generalization challenge of discovering new triplets based on existing ones, which necessitates the development of KG models with more robust and generalizable reasoning capability. Furthermore, this dataset presents an important challenge of effectively scaling embedding models to large KGs. Naively training the KG embedding models with reasonable dimensionality requires a high-end GPU, which is extremely costly and not scalable to even larger KGs. A promising approach to improve scalability is to distribute training across multiple commodity GPUs (Zheng *et al.*, 2020b; Zhu *et al.*, 2019c; Lerer *et al.*, 2019). A different approach is to share parameters across entities and relations, so that a smaller number of embedding parameters need to be put onto the GPU memory at once (Hu *et al.*, 2020a).

On `wikikg90M`, we only consider the two KG embedding models TransE and ComplEx due to the enormous size of the KG (Hu *et al.*, 2021b). In order to enhance the expressive power of the KG model, we further make use of the additional RoBERTa encodings of entities and relations by concatenating them to the distinct embeddings of the KG model. This way, the encoders can adaptively utilize the RoBERTa encodings as well as the distinct embeddings in order to derive a prediction. In an ablation study, we see that using only distinct embeddings or RoBERTa encodings gives way

Method	Virtual Node	Dataset molhiv	molpcba	pcqm4M	ppa	code
#Graphs		41K	438K	3.8M	158K	453K
Avg. #Nodes		27.5	26.0	26.5	243.4	125.2
Avg. #Edges		27.5	28.1	29.1	2,266.1	124.2
Metric		ROC-AUC	AP	MAE	Accuracy	F1 Score
MLP-Fingerprint	—	**80.60**±1.00	22.26±0.02	0.2068	—	—
GCN	—	76.06±0.97	20.20±0.24	0.1838	68.39±0.84	15.07±0.18
	✔	75.99±1.19	24.24±0.34	0.1579	68.57±0.61	**15.95**±0.18
GIN	—	75.58±1.40	22.66±0.28	0.1678	68.92±1.00	14.95±0.23
	✔	77.07±1.49	**27.03**±0.23	**0.1487**	**70.37**±1.07	15.81±0.26

Table 5.7: Model performance on all graph-level OGB datasets, using a diverse set of representative graph-based machine learning models.

weaker test performance (up to a difference of 0.28 in MRR). This suggests that combining both textual information and structural information gives the most promising performance (Hu *et al.*, 2021b). In the OGB-LSC[6], the winning solutions pushed the performance further up to an MRR of 0.9727.

5.5.1.3 Graph Property Prediction. For the graph-level datasets in OGB, we consider the following representative models as baselines (Hu *et al.*, 2020a, 2021b):

- **MLP-Fingerprint**: An MLP over the Morgan fingerprint (Morgan, 1965).
- **GCN**: A Graph Convolutional Network (Kipf & Welling, 2017).
- **GIN**: A Graph Isomorphism Network (Xu *et al.*, 2019c).
- **GCN+Virtual**: A GCN that performs additional message passing over virtual nodes, *i.e.* nodes that are connected to all other nodes in the original graph (Gilmer *et al.*, 2017; Li *et al.*, 2017; Ishiguro *et al.*, 2019).
- **GIN+Virtual**: A GIN that performs additional message passing over virtual nodes.

In the GNN models, node embeddings are further pooled into graph-level representations to obtain an embedding of the entire graph. Finally, a linear model is applied to the graph embedding in order to make predictions. To include edge features (such as the bond type in molecular graphs), we follow Hu *et al.* (2020b) and add transformed edge features into source node features. For all GNN experiments, we make use of 5-layer GNNs, global average pooling, a hidden dimensionality of 300, and a tuned dropout ratio $\in \{0.0, 0.5\}$ (Srivastava *et al.*, 2014; Hu *et al.*, 2020a).

Benchmarking results on all graph-level datasets included in OGB are shown in Table 5.7. Overall, we see that GNN models with message passing over virtual nodes perform the best across the different datasets, consistently improving generalization

[6] `https://ogb.stanford.edu/kddcup2021/results` (last access: August 25, 2022)

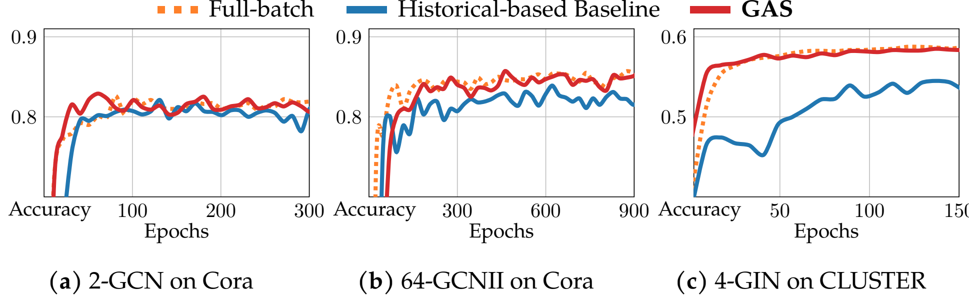
performance. Furthermore, expressive GNNs (GIN and GIN+Virtual) typically outperform less expressive ones (GCN and GCN+Virtual).

In particular, on the molecular graph datasets `molhiv`, `molpcba` and `pcqm4M`, GNNs achieve promising performance. While the MLP-FINGERPRINT variant achieves the overall best test performance on `molhiv`, this is not the case on the larger datasets which showcases the ability of GNNs to make well use of larger amount of data. Furthermore, we find the conventional random split to be much easier than our scaffold splitting strategy. For example, using random splits with the same split ratio, we achieve a ROC-AUC of 82.73±2.02% (5.66 percentage points higher than scaffold) and an AP of 34.40±0.90% (7.37 percentage points higher than scaffold) on the `molhiv` and `molpcba` datasets, respectively, using the best performing GNN model. These results highlight the challenges of the scaffold split compared to traditional random splits, and opens up fruitful research opportunities to increase the out-of-distribution generalization capabilities of GNNs. On the large-scale `pcqm4M` dataset, GIN+Virtual significantly outperforms the other graph-based machine learning models. Nonetheless, the current performance is still much worse than the desired chemical accuracy of 0.043eV — an indicator of practical usefulness established by the chemistry community (Hu *et al.*, 2021b). We further perform ablation studies regarding model size and validation performance on `pqm4M`. In particular, we see that the largest models always achieve the best performance, *e.g.*, 0.1410eV MAE using a GNN model with 5 layers and hidden dimensionality of 600 vs 0.1512eV MAE using a GNN model with 3 layers and hidden dimensionality of 300 (Hu *et al.*, 2021b). The winning solutions of the OGB-LSC[7] utilize even deeper GNNs with sophisticated self-supervision auxiliary tasks. The final test performance of the winning solution is 0.1200eV MAE.

On the `ppa` dataset, the GIN model with message passing over virtual nodes provides once again the best performance. Nevertheless, the generalization gap between training and test performance is huge (almost 30 percentage points). For reference, the same model in a random splitting scenario achieves a final test accuracy of 92.91±0.27%, which is more than 20 percentage points higher than the utilized species split. This again encourages future research to improve the out-of-distribution generalization on the more challenging and meaningful split procedure.

Our benchmark analysis on the `code` dataset utilizes "next-token edges" on top of the AST to better capture the semantics of code graphs (Dinella *et al.*, 2020). For the decoder, we use independent linear classifiers to predict sub-tokens at each position of the sub-token sequence. We find that this order-sensitive decoder performs slightly better than an order-insensitive decoder that simply predicts whether each vocabulary is included in the target sequence (Hu *et al.*, 2020a). On the random split, the test F1 score is 21.64±0.26% (approximately 6 percentage points higher than that of the project split), indicating that the project split is indeed harder than the random split. Nonetheless, the obtained performance is far from being of practical usefulness. As such, this dataset presents an interesting research opportunity to improve out-of-distribution generalization under the meaningful project split, with a number of fruitful future directions: how to leverage the fact that the original graphs are actually trees with well-defined root nodes, how to pre-train GNNs to improve generalization, and how to design better encoder-decoder architectures (Hu *et al.*, 2020a).

[7] `https://ogb.stanford.edu/kddcup2021/results` (last access: August 25, 2022)



(**a**) 2-GCN on Cora (**b**) 64-GCNII on Cora (**c**) 4-GIN on CLUSTER

Figure 5.10: Model performance comparison between full-batch, an unoptimized history-based baseline and our GAS approach. In contrast to the historical-based baseline, GAS reaches the quality of full-batch training, especially for (b) deep and (c) expressive models (Fey *et al.*, 2021).

5.5.2 Deep and Expressive GNNs on Large-Scale Graphs

Our GAS framework proposes a new method for scaling up the training and inference phase of GNNs on large-scale graphs (Fey *et al.*, 2021). GAS prunes entire sub-trees of the computation graph by utilizing historical embeddings from prior training iterations. As described in Section 5.3, it allows to scale up deep GNNs due to its constant GPU memory consumption in respect to input node size. Furthermore, it allows the application of expressive GNNs since GAS is provably able to maintain the expressive power of the underlying GNN. Here, we showcase these benefits of GAS empirically while comparing to related scalability techniques. Specifically, we show that GAS allows for deep and expressive GNNs while resembling full-batch performance (Section 5.5.2.1), is very memory-efficient (Section 5.5.2.2), and scales well to large graphs (Section 5.5.2.3). In our experiments, we utilize a total of 6 different GNN operators on 15 different datasets. The code for reproducing all experiments utilizes the PyTorch Geometric library (Fey & Lenssen, 2019) and is available on GitHub.[8] Please refer to the code base for a detailed description of hyperparameter configurations. All models were trained on a single GeForce RTX 2080 Ti (11 GB). In our experiments, we hold all histories in RAM, using a machine with 64GB of CPU memory (Fey *et al.*, 2021).

5.5.2.1 GAS allows for deep and expressive GNNs. We compare GAS against two different baselines: a regular full-batch variant and a history baseline, which naively integrates history-based mini-batch training without any of the additional GAS techniques (Section 5.3.2). To evaluate, we make use of a shallow 2-layer GCN (Kipf & Welling, 2017) and two recently introduced state-of-the-art models: a deep GCNII network with 64 layers (Chen *et al.*, 2020b), and a maximally expressive GIN network with 4 layers (Xu *et al.*, 2019c). We evaluate those models on tasks for which they are well suitable: classifying academic papers in a citation network (Cora), and identifying community clusters in Stochastic Block Models (CLUSTER) (Sen *et al.*, 2008; Yang *et al.*, 2016; Dwivedi *et al.*, 2020), *cf.* Figure 5.10. Since CLUSTER is a node classification task containing multiple graphs, we first convert it into a super graph (holding all the nodes of all graphs), and partition this super graph using twice as

[8]Code for GAS: `https://github.com/rusty1s/pyg_autoscale` (last access: August 25, 2022)

Dataset	GCN		GAT		APPNP		GCNII	
	Full	**GAS**	Full	**GAS**	Full	**GAS**	Full	**GAS**
Cora	81.88	82.29	82.80	83.32	83.28	83.19	85.04	85.52
CiteSeer	70.98	71.18	71.72	71.86	72.13	72.63	73.06	73.89
PubMed	78.73	79.23	78.03	78.42	80.21	79.82	79.72	80.19
Coauthor-CS	91.08	91.22	90.31	90.38	92.51	92.44	92.45	92.52
Coauthor-Physics	93.10	92.98	92.32	92.80	93.40	93.68	93.43	93.61
Amazon-Computer	81.17	80.84	—†	—†	81.79	81.66	83.04	83.05
Amazon-Photo	90.25	90.53	—†	—†	91.27	91.23	91.42	91.60
Wiki-CS	79.08	79.00	79.44	79.56	79.88	79.75	79.94	80.02
Δ Mean Acc.	**+0.13**		**+0.29**		**-0.01**		**+0.29**	

† Results omitted due to unstable performance, *cf.* Shchur *et al.* (2018).

Table 5.8: Full-batch vs GAS performance on small transductive graph benchmark datasets across 20 different initializations. Predictive performance of models trained via GAS closely matches those of full-batch gradient descent on all models for all datasets (Fey *et al.*, 2021).

many partitions as there are initial graphs. It can be seen that especially for deep (64-GCNII, *cf.* Figure 5.10b) and expressive (4-GIN, *cf.* Figure 5.10c) architectures, the naive historical-based baseline fails to reach the desired full-batch performance. This can be contributed to the high approximation error induced by deep and expressive models. In contrast, GAS shows far superior performance, reaching the quality of full-batch training in both cases (Fey *et al.*, 2021).

In general, we expect the model performances of our GAS mini-batch training to closely resemble the performances of their full-batch counterparts, except for the variance introduced by stochastic optimization (Bottoue & Bousquet, 2007). To validate, we compare GAS against full-batch performances on small transductive benchmark datasets for which full-batch training is easily feasible, namely the three citation datasets Cora, CiteSeer and PubMed (Sen *et al.*, 2008; Yang *et al.*, 2016), the two co-authorship graphs Coauthor-CS and Coauthor-Physics (Shchur *et al.*, 2018), the two co-purchase graphs Amazon-Computer and Amazon-Photo (Shchur *et al.*, 2018) as well as the Wikipedia graph Wiki-CS (Mernyei & Cangea, 2020) containing computer science articles. We evaluate on four GNN models that significantly advanced the field of Graph Representation Learning: GCN (Kipf & Welling, 2017), GAT (Veličković *et al.*, 2018), APPNP (Klicpera *et al.*, 2019a) and GCNII (Chen *et al.*, 2020b). For all experiments, we tried to follow the hyperparameter setup of the respective papers as closely as possible and perform an in-depth grid search on datasets for which best performing configurations are not known. We then apply GAS mini-batch training on the *same* set of hyperparameters. As shown in Table 5.8, all models that utilize GAS training perform as well as their full-batch equivalents (with slight gains overall), confirming the practical effectiveness of our approach. Notably, even for deep GNNs such as APPNP and GCNII, our approach is able to closely resemble the desired performance (Fey *et al.*, 2021).

We further conduct ablation studies to highlight the individual performance improvements of our GAS techniques within a deep GCNII model, *i.e.* minimizing inter-connectivity and applying regularization techniques such that enforcing Lipschitz con-

Method	Cora	CiteSeer	PubMed	Coauthor-CS	Coauthor-Physics	Amazon-Computer	Amazon-Photo	Wiki-CS
Baseline	-3.26	-5.66	-3.20	-0.79	-0.50	-5.76	-4.16	-3.19
Regularization	-2.12	-1.03	-1.24	-0.46	-0.24	-3.02	-1.19	-0.74
Metis	-1.57	-3.12	-1.50	-0.47	+0.13	-2.75	-1.02	-0.24
GAS	**+0.48**	**+0.83**	**+0.47**	**+0.07**	**+0.18**	**+0.01**	**+0.18**	**+0.08**

Table 5.9: Relative performance improvements of individual GAS techniques within a GCNII model. The performance improvement is measured in percentage points in relation to the corresponding model performance obtained by full-batch training (Fey *et al.*, 2021).

Method	**Accuracy** Training	Validation	Test
Full-batch	60.49	58.17	58.49
Regularization	55.66	54.86	55.15
Metis	58.97	57.79	57.82
GAS	**60.67**	**58.21**	**58.51**

Table 5.10: Ablation study for a 4-layer GIN model on the CLUSTER dataset. Combining both GAS techniques help in resembling full-batch performance for expressive GNN models (Fey *et al.*, 2021).

tinuity. Table 5.9 shows the relative performance improvements of individual GAS techniques in percentage points, compared to the corresponding model performance obtained by full-batch training. Notably, it can be seen that both techniques contribute to resembling full-batch performance, reaching their full strength when used in combination. The same trend holds true for expressive GNN variants, *i.e.* for a 4 layer GIN model on the CLUSTER dataset, *cf.* Table 5.10. Notably, both solutions achieve significant gains in training, validation and test performance, and together, they are able to closely resemble the performance of full-batch training (Fey *et al.*, 2021).

5.5.2.2 GAS is memory-efficient. For training large-scale GNNs, GPU memory consumption will directly dictate the scalability of the given approach. Here, we show how GAS maintains a low GPU memory footprint while, in contrast to other scalability approaches, accounts for *all* available information inside a GNN's receptive field in a single optimization step. We compare the memory usage of GCN+GAS training with the memory usage of full-batch GCN (Kipf & Welling, 2017), and mini-batch GraphSAGE (Hamilton *et al.*, 2017) and Cluster-GCN (Chiang *et al.*, 2019) training, *cf.* Table 5.11. For datasets, we utilize the Yelp dataset containing customers and their friendship relations (Zeng *et al.*, 2020b) as well as the two node-level datasets `arxiv` and `products` from OGB (Hu *et al.*, 2020a), *cf.* Section 5.4. Notably, GAS is easily able to fit the required data on the GPU, while memory consumption only increases linearly with the number of layers. Although Cluster-GCN maintains an overall lower memory footprint than GAS, it will only utilize a fraction of available information inside its receptive field, *i.e.* 23% on average (Fey *et al.*, 2021).

	Method	Yelp	arxiv	products
	#Nodes	717K	169K	2.4M
	#Edges	7.9M	1.2M	62M
2-layer	Full-batch	6.64GB / 100%	1.44GB / 100%	21.96GB / 100%
	GraphSAGE	0.76GB / 9%	0.40GB / 27%	0.92GB / 2%
	Cluster-GCN	0.17GB / 13%	0.15GB / 40%	0.16GB / 16%
	GAS	0.51GB / 100%	0.22GB / 100%	0.36GB / 100%
3-layer	Full-batch	9.44GB / 100%	2.11GB / 100%	31.53GB / 100%
	GraphSAGE	2.19GB / 14%	0.93GB / 33%	4.34GB / 5%
	Cluster-GCN	0.23GB / 13%	0.22GB / 40%	0.23GB / 16%
	GAS	0.79GB / 100%	0.34GB / 100%	0.59GB / 100%
4-layer	Full-batch	12.24GB / 100%	2.77GB / 100%	41.10GB / 100%
	GraphSAGE	4.31GB / 19%	1.55GB / 37%	11.23GB / 8%
	Cluster-GCN	0.30GB / 13%	0.29GB / 40%	0.29GB / 16%
	GAS	1.07GB / 100%	0.46GB / 100%	0.82GB / 100%

Table 5.11: GPU memory consumption (in GB) and the amount of data used (%) across different GNN execution techniques. GAS consumes low memory while making use of all available neighborhood information during a single optimization step (Fey *et al.*, 2021).

	Method	Reddit	PPI	Flickr	Yelp	arxiv	products
	#Nodes	230K	57K	89K	717K	169K	2.4M
	#Edges	11.6M	794K	450K	7.9M	1.2M	62M
	GraphSAGE	95.40	61.20	50.10	63.40	71.49	78.70
	FastGCN	93.70	—	50.40	—	—	—
	LADIES	92.80	—	—	—	—	—
	VR-GCN	94.50	85.60	—	61.50	—	—
	MVS-GNN	94.90	89.20	—	62.00	—	—
	Cluster-GCN	96.60	99.36	48.10	60.90	—	78.97
	GraphSAINT	97.00	**99.50**	51.10	65.30	—	79.08
	SGC	96.40	96.30	48.20	64.00	—	—
	SIGN	96.80	97.00	51.40	63.10	—	77.60
	GBP	—	99.30	—	**65.40**	—	—
Full	GCN	95.43	97.58	53.73	OOM	71.64	OOM
	GCNII	OOM	OOM	55.28	OOM	72.83	OOM
	PNA	OOM	OOM	56.23	OOM	72.17	OOM
GAS	GCN	95.45	98.92	54.00	62.94	71.68	76.66
	GCNII	96.77	**99.50**	56.20	65.14	**73.00**	77.24
	PNA	**97.17**	99.44	**56.67**	64.40	72.50	**79.91**

Table 5.12: Performance on large graph datasets. GAS is both scalable and general while achieving state-of-the-art performance (Fey *et al.*, 2021).

5.5.2.3 GAS scales to large-graphs. In order to demonstrate the scalability and generality of our GAS approach, we scale various GNN operators to common large-scale graph benchmark datasets. We evaluate on 6 datasets for diverse tasks: Predicting communities of online posts based on user comments (Reddit) (Hamilton *et al.*, 2017), classifying protein functions based on the interactions of human tissue proteins (PPI) (Hamilton *et al.*, 2017), categorizing types of images based on their descriptions and properties (Flickr) (Zeng *et al.*, 2020b), classifying business types based on customers and friendship relations (Yelp) (Zeng *et al.*, 2020b), predicting subject areas of arXiv Computer Science papers (`arxiv`) (Hu *et al.*, 2020a), and predicting product categories in an Amazon product co-purchasing network (`ogbn-products`) (Hu *et al.*, 2020a). Here, we focus our analysis on GNNs that are notorious hard to scale-up but have the potential to leverage the increased amount of available data to make more accurate predictions. In particular, we benchmark deep GNNs, *i.e.* GCNII (Chen *et al.*, 2020b), and expressive GNNs, *i.e.* PNA (Corso *et al.*, 2020). Note that it is not possible to run those models in full-batch mode on most of these datasets as they will run out of memory on common GPUs. We compare with 10 scalable GNN baselines: GraphSAGE (Hamilton *et al.*, 2017), FastGCN (Chen *et al.*, 2018b), LADIES (Zou *et al.*, 2019), VR-GCN (Chen *et al.*, 2018c), MVS-GNN (Cong *et al.*, 2020), Cluster-GCN (Chiang *et al.*, 2019), GraphSAINT (Zeng *et al.*, 2020b), SGC (Wu *et al.*, 2019a), SIGN (Frasca *et al.*, 2020) and GBP (Chen *et al.*, 2020a). Since results are hard to compare across different approaches due to differences in frameworks, model implementations, weight initializations and optimizers, we additionally report a shallow GCN+GAS baseline. GAS is able to train all models on all datasets on a single GPU, while holding corresponding histories in CPU memory. On the largest dataset, *i.e.* `products`, this will consume approximately $L \cdot$ 2GB of storage for L layers, which easily fits in RAM on most modern workstations (Fey *et al.*, 2021).

As can be seen from Table 5.12, the usage of deep and expressive models within our framework advances the state-of-the-art on Reddit and Flickr, while it performs equally well for others, *e.g.*, PPI. Notably, our approach outperforms the two historical-based variants VR-GCN and MVS-GNN by a wide margin. Interestingly, our deep and expressive variants reach superior performance than our GCN baseline on *all* datasets, which highlights the benefits of evaluating larger models on larger scale. Furthermore, our scalable GCNII variant is up to this date the top contender on the OGB `arxiv` leaderboard[9] across the models which do not make use of ground-truth labels during forward execution (Fey *et al.*, 2021).

[9] Leaderboard: `https://ogb.stanford.edu/docs/leader_nodeprop` (last access: January 11, 2022)

6

Efficient Realization of Graph Neural Networks

With the rise of Graph Neural Networks as a state-of-the-art technique for Graph Representation Learning, there is an urgent demand in both flexible and powerful libraries for accelerating research and putting existing models into production. Here, we introduce PyTorch Geometric, a well-known deep learning library for implementing and working with graph-based neural network building blocks. PyTorch Geometric leverages sparse GPU acceleration by providing dedicated CUDA kernels, and introduces efficient mini-batch handling for input examples of different size. Furthermore, we present PyGAS, an easy-to-use extension for PyTorch Geometric that converts common and custom Graph Neural Network models into their scalable variants by utilizing our GNNAutoScale framework. In particular, PyGAS optimizes the access pattern of historical embeddings in order to allow for both fast and memory-efficient mini-batch training on large-scale graphs.

6.1 Introduction 129

6.2 State-of-the-Art 130

6.3 Graph Neural Networks within PyTorch Geometric 131

6.4 PyGAS: Auto-Scaling Graph Neural Networks 152

6.5 Evaluation 154

6.1 Introduction

Graph Neural Networks (GNNs) emerged as a powerful approach for representation learning on graphs that unify and generalize the concepts of Convolutional Neural Networks (CNNs) (*cf.* Section 3.3) and Transformers (*cf.* Section 3.2.4) to arbitrarily structured data. As a result, the computation graph is no longer tied to its underlying model, but is instead given dynamically as part of its input in the form of a sparsely

given structure. However, modern deep learning software libraries are heavily designed with regular structures and dense tensor computation in mind, *e.g.*, in the form of efficient batch-wise dense matrix multiplications. This makes the efficient realization of GNNs challenging, as high GPU throughput needs to be achieved on highly sparse and irregular data of varying size (Gale *et al.*, 2020).

As such, there is an urgent demand in both flexible and efficient libraries to effectively realize the theoretical and methodological contributions of this thesis. Here, flexibility guarantees that the underlying tools and their design principles do not limit the realization of the proposed methods (and are able to accelerate future research as well). At the same time, efficiency and scalability via thoughtful computation and memory-access patterns ensure the practical use in real-world applications.

In order to accomplish these goals, we propose the PyTorch Geometric (PyG) library in Section 6.3, a library built upon PyTorch (Paszke *et al.*, 2019) to easily write and train Graph Neural Networks for a wide range of applications related to structured data. PyG elegantly marries the tensor-centric perspective of deep learning frameworks with the sparse and irregular design paradigm of GNNs. In particular, PyG achieves high data throughput by leveraging sparse GPU acceleration, by providing dedicated CUDA kernels and by introducing efficient mini-batch handling for input examples of different size. In addition to general graph data structures and processing methods, PyG further bundles a variety of state-of-the-art methods from the domains of relational learning and 3D data processing based on unified interfaces, which allows for rapid and clean prototyping of new research ideas.

Furthermore, we present the PyTorch Geometric AutoScale (PyGAS) extension in Section 6.4, the practical realization of of our GNNAutoScale framework, *cf.* Section 5.3. PyGAS is an easy-to-use extension for PyG to *auto-scale* any GNN to large-scale graphs while maintaining the properties of the original GNN. PyGAS makes it easy to convert common and custom GNN models into their scalable variants while requiring orders of magnitude less GPU memory, and provides a fully deterministic test bed for evaluating models on larger scale. In particular, PyGAS optimizes pulling and pushing to histories via *non-blocking* device transfers, such that no overhead occurs at all when accessing historical embeddings.

6.2 State-of-the-Art

Since PyG started to bundle and make state-of-the-art Graph Representation Learning available in a unified and efficient package, various additional graph-based machine learning libraries have been developed, mostly differing in scope, functionality and practical application. The *Deep Graph Library* (*DGL*) (Wang *et al.*, 2019b) allows the implementation of GNNs in a framework-agnostic fashion, and mostly differs from PyG in how graph data is represented internally: DGL utilizes a dedicated C++/CUDA graph storage layer, while PyG represents graphs as pure tensors. Despite divergent APIs, both libraries have emerged to a similar feature set over the years. GraphNet (Battaglia *et al.*, 2018), Spektral (Grattarola & Alippi, 2020), StellarGraph (CSIRO-Data61, 2018), Jraph (Godwin *et al.*, 2020), `ptgnn` (Microsoft, 2019) and `tf2-gnn` (Jackson-Flux *et al.*, 2019) provide alternative message passing implementations based on similar principles for all kinds of deep learning frameworks such as TensorFlow (Abadi *et al.*, 2015), Keras (Chollet *et al.*, 2015) and Jax (Bradbury *et al.*,

2018), but lack crucial functionality for large-scale graph learning support. In contrast, AliGraph (Zhu *et al.*, 2019b), Euler (Alibaba, 2019), Neugraph (Ma *et al.*, 2019) and PGL (Baidu, 2019) provide dedicated and optimized libraries for GNN training and deployment in industrial applications, *e.g.*, via fast samplers or multi-GPU support through partitioning. As a compromise, they lack the flexibility and comprehensiveness of libraries targeted towards research. TorchDrug (Zhu *et al.*, 2021b) focuses on integrating recent GNN advances in drug discovery to accelerate further development.

Recently, various high-level libraries emerged on top of PyG: PyTorch Geometric Temporal (Rozemberczki *et al.*, 2021b) provides support for temporal graphs in PyG. DIG (Liu *et al.*, 2021) provides a unified and PyG-based testbed for various graph-based deep learning tasks. Quiver (Quiver, 2021) enables distributed graph learning for scaling PyG models across many nodes and many GPUs. Furthermore, recent efforts bring a subset of PyG's functionality to other frameworks as well, *e.g.*, `tf_geometric` (Hu *et al.*, 2021a) to TensorFlow, and GeometricFlux (FluxML, 2020) to Julia (Bezanson *et al.*, 2017).

Efficient graph-based processing is highly related to the design and innovation of sparse matrix formats and GPU-based parallel computation strategies for sparse matrix multiplication within each format (Bell & Garland, 2008, 2009; Dalton *et al.*, 2015; Filippone *et al.*, 2017; Shi *et al.*, 2020a). These techniques introduce advanced coalesced memory access patterns (Yang *et al.*, 2018), tiling and reordering strategies (Yang *et al.*, 2011; Baskaran & Bordawekar, 2009), adaptive re-purposing of GPU threads (Winter *et al.*, 2019), and distributed computation at large scale (Hussain *et al.*, 2020) Furthermore, dedicated sparse matrix algorithms have been proposed in the context of graph analytics (Ashari *et al.*, 2014; Wang *et al.*, 2016), scientific computing applications (LeVeque, 2007) and deep learning (Gale *et al.*, 2020).

The rise of new "AI accelerator types" such as Google's Tensor Processing Units (TPUs) or NVIDIA's Volta architecture have also motivated the design of novel computation algorithms for sparse matrices and GNNs in particular (Zachariadis *et al.*, 2020). For example, (Balog *et al.*, 2019) utilize block sparse matrices obtained from bandwidth minimization to accelerate GNN performance on TPUs. Furthermore, there is a long line of research regarding effective *distributed* GNN execution (Jia *et al.*, 2020; Ma *et al.*, 2019; Zhu *et al.*, 2016; Tripathy *et al.*, 2020; Wan *et al.*, 2020; Angerd *et al.*, 2020; Zheng *et al.*, 2020a), which utilize various forms of graph partitioning strategies and memory management optimizations to allow for the efficient exchange of graph data in distributed environments.

6.3 Graph Neural Networks within PyTorch Geometric

Here, we introduce *PyTorch Geometric* (*PyG*) (Fey & Lenssen, 2019), a framework-specific Graph Neural Network library built upon PyTorch (Paszke *et al.*, 2019) that provides the necessary tools for processing and learning from irregularly structured data, and achieves high performance by leveraging dedicated CUDA kernels. Following a simple message passing API, it bundles most of the recently proposed convolutional and pooling layers into a single and unified framework. All implemented methods support both CPU and GPU computations and follow an immutable data flow paradigm that enables dynamic changes in graph structures through time. PyG

is fully open-sourced on GitHub.[1] It is thoroughly documented and provides accompanying tutorials, videos, notebooks and examples as a first starting point.[2] Up to this date, approximately 800 research papers were written using PyG, it has established an active Slack community with around 1,400 participants, and its future development involves a team of core developers extended by a community of more than 230 contributors across the world.

We first give an overview of the library (Section 6.3.1) and its general interfaces (Section 6.3.2), before presenting lower-level sparse tensor arithmetic details (Section 6.3.3) and additional but advanced features of PyG such as heterogeneous graph learning (Sections 6.3.4 and 6.3.5).

6.3.1 Overview of the Library

PyG provides a fast and flexible framework to simplify implementing and working with GNNs. Overall, it aims to help users to apply relational learning to specific domains and tasks, to help researchers to invent novel methods and compare themselves to related work, and to help to make research more reproducible. It is generally built upon the following design principles:

- **Easy-to-use and unified API:** Users can directly start training a GNN model with just a few lines of code. In particular, PyG aims to be *PyTorch-on-the-rocks*: It utilizes a tensor-centric API and keeps design principles close to vanilla PyTorch. For example, all user facing APIs, *e.g.*, data loading routines, multi-GPU support, data transformations and augmentations, or model instantiations are heavily inspired by PyTorch to keep them as familiar as possible.
- **Comprehensive and flexible:** PyG aims to bundle the state-of-the-art in Graph Representation Learning into a comprehensive and well maintained GNN library. As such, most of the GNN operators and architectures proposed in research have been integrated into the library, and existing graph-based neural network building blocks can either be extended, or combined and applied to various parts of a GNN model, ensuring rich flexibility in GNN design. PyG further provides an abundant set of over 300 common benchmark datasets.
- **Efficient and scalable:** PyG achieves high data throughput by leveraging sparse GPU acceleration, by providing dedicated CUDA kernels and by introducing efficient mini-batch handling for input examples of different size. PyG further supports the implementation of GNNs that can scale to large-scale graphs, containing millions of nodes. Many state-of-the-art scalability approaches have been integrated into PyG, and can benefit from the rich set of GNN operators and models available in PyG.

6.3.1.1 Components. PyG provides a multi-layer framework that enables users to build Graph Neural Network solutions on both low and high levels. It comprises of the following components, *cf.* Figure 6.1:

[1] PyG repository: `https://github.com/rusty1s/pytorch_geometric` (last access: August 25, 2022)
[2] PyG documentation: `https://pytorch-geometric.readthedocs.io` (last access: August 25, 2022)

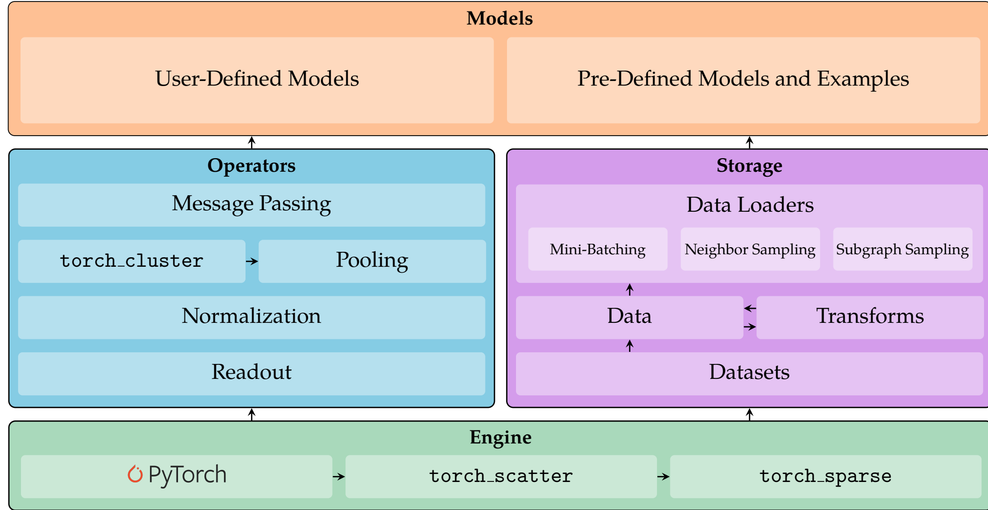


Figure 6.1: Illustrative architecture of the PyTorch Geometric library. GNN operators and graph storage capabilities utilize the PyTorch deep learning framework and in-house additions of efficient CUDA libraries, and can be combined to build and train GNN models with rich flexibility.

- The **PyG engine** utilizes the powerful PyTorch deep learning framework (Paszke *et al.*, 2019), as well as in-house additions of efficient CUDA libraries for operating on sparse data, *e.g.*, `torch-scatter`,[3] `torch-sparse`,[4] and `torch-cluster`.[5]
- The **PyG storage** handles data processing, transformation and loading pipelines. It is capable of handling and processing large-scale graph datasets, and provides effective solutions for heterogeneous graphs. It further provides a variety of sampling solutions, which enable training of GNNs on large-scale graphs.
- The **PyG operators** bundle essential functionalities for implementing Graph Neural Networks. PyG supports important GNN building blocks that can be combined and applied to various parts of a GNN model, ensuring rich flexibility of GNN design and fast design space exploration.
- Finally, PyG provides an abundant set of **GNN models**, and **examples** that showcase the application of GNNs on standard graph benchmark datasets. Due to its flexibility, users can easily build and modify custom GNN models to fit their specific needs.

Table 6.1 lists currently supported PyG operators and models according to category.

In total, PyG supports nearly 50 different GNN message passing layers, which are ready to be deployed upon installation. These layers are implemented based a unified differentiable `MessagePassing` interface that decomposes Message, Update and aggregative functions $\bigoplus$ into customizable user defined functions, *cf.* Sections 3.2.2

[3]`torch-scatter`: `https://github.com/rusty1s/pytorch_scatter` (last access: August 25, 2022)
[4]`torch-sparse`: `https://github.com/rusty1s/pytorch_sparse` (last access: August 25, 2022)
[5]`torch-cluster`: `https://github.com/rusty1s/pytorch_cluster` (last access: August 25, 2022)

Category	**Supported Methods**
Message Passing	Cheby (Defferrard *et al.*, 2016), GraphSAGE (Hamilton *et al.*, 2017), GCN (Kipf & Welling, 2017), GAT (Veličković *et al.*, 2018), GCNII (Chen *et al.*, 2020b), SplineCNN (Fey *et al.*, 2018), MPNN (Gilmer *et al.*, 2017), PointNet (Qi *et al.*, 2017a,b), MoNet (Monti *et al.*, 2017), EdgeCNN (Wang *et al.*, 2019e), S-GCN (Wu *et al.*, 2019a), R-GCN (Schlichtkrull *et al.*, 2018), GIN (Xu *et al.*, 2019c), GIN-E (Hu *et al.*, 2020b), NMF (Duvenaud *et al.*, 2015), DNA (Fey, 2019), Hyper-GCN (Bai *et al.*, 2021), GraphNet (Battaglia *et al.*, 2018), PNA (Corso *et al.*, 2020), Cluster-GCN (Chiang *et al.*, 2019), k-GNN (Morris *et al.*, 2019), SuperGAT (Kim & Oh, 2021), PointCNN (Li *et al.*, 2018b), SparseTransformer (Shi *et al.*, 2020b), GatedGCN (Li *et al.*, 2016b), ResGatedGCN (Bresson & Laurent, 2017), ARMA (Bianchi *et al.*, 2019), EC (Simonovsky & Komodakis, 2017), Signed-GCN (Derr *et al.*, 2018), AGNN (Thekumparampil *et al.*, 2018), CG (Xie & Grossman, 2018), TAG (Du *et al.*, 2017), HGT (Hu *et al.*, 2020c), FeaStNet (Verma *et al.*, 2018), PPFNet (Deng *et al.*, 2018), GravNet (Qasim *et al.*, 2019), PDN (Rozemberczki *et al.*, 2021a), EG (Tailor *et al.*, 2021), LeCNN (Ranjan *et al.*, 2020), PAN (Ma *et al.*, 2021), DeepGCN (Li *et al.*, 2019a), FiLM (Brockschmidt, 2020), GATII (Brody *et al.*, 2021), FA (Bo *et al.*, 2021), GeneralGNN (You *et al.*, 2020), PointTransformer (Zhao *et al.*, 2020), AttentiveFP (Xiong *et al.*, 2021), GeniePath (Liu *et al.*, 2019b)
Pooling	Top-k (Cangea *et al.*, 2018), DiffPool (Ying *et al.*, 2018b), MinCUT (Bianchi *et al.*, 2020), Graclus (Dhillon *et al.*, 2007), VoxelGrid (Simonovsky & Komodakis, 2017), SAG (Lee *et al.*, 2019), EdgePool (Diehl *et al.*, 2019), ASAP (Ranjan *et al.*, 2020), PAN (Ma *et al.*, 2021), MemPool (Khasahmadi *et al.*, 2020), GMT (Baek *et al.*, 2021), GlobalAttention (Li *et al.*, 2016b), Set2Set (Vinyals *et al.*, 2016), SortPool (Zhang *et al.*, 2018), FPS (Qi *et al.*, 2017b)
Models	DeepWalk (Perozzi *et al.*, 2014), Node2Vec (Grover & Leskovec, 2016), MetaPath2Vec (Dong *et al.*, 2017a), JK (Xu *et al.*, 2018), APPNP (Klicpera *et al.*, 2019a), (V)GAE (Kipf & Welling, 2016), ARG(V)A (Pan *et al.*, 2018), S-GAE (Salha *et al.*, 2020), WL (Weisfeiler & Lehman, 1968), LP (Zhu *et al.*, 2003), ReNet (Jin *et al.*, 2020), TGN (Rossi *et al.*, 2020), RECT (Wang *et al.*, 2021), SEAL (Zhang & Chen, 2018), SchNet (Schütt *et al.*, 2017), DimeNet (Klicpera *et al.*, 2020b), DGI (Veličković *et al.*, 2019), Graph U-Net (Gao & Ji, 2019)
Utilities	GNNExplainer (Ying *et al.*, 2019), DropEdge (Rong *et al.*, 2020b), GDC (Klicpera *et al.*, 2019b), GraphNorm (Cai *et al.*, 2020), GraphSizeNorm (Dwivedi *et al.*, 2020), PairNorm (Zhao & Akoglu, 2020), DiffGroupNorm (Zhou *et al.*, 2020), TreeDecomp (Jin *et al.*, 2018), LDP (Cai & Wang, 2018), C&S (Huang *et al.*, 2021), Gini (Henderson *et al.*, 2021)
Scalability	GraphSAGE (Hamilton *et al.*, 2017), Cluster-GCN (Chiang *et al.*, 2019), GraphSAINT (Zeng *et al.*, 2020b), S-GCN (Wu *et al.*, 2019a), SIGN (Frasca *et al.*, 2020), ShaDow (Zeng *et al.*, 2020a), HGT (Hu *et al.*, 2020c), GAS (Fey *et al.*, 2021)

Table 6.1: Overview of all currently supported methods in PyTorch Geometric.

and 6.3.2. The provided operators mostly differ in the input they expect or the way the Message computation is performed. For example, some operators utilize attention or can incorporate one-dimensional or multi-dimensional edge features, while others are especially designed for learning expressive node representations that are able to distinguish structural graph properties. In addition, some operators are designed for operating on geometric input data, such as point clouds or meshes.

PyG also supports graph-level outputs as opposed to node-level outputs by providing a variety of Readout functions such as global add, mean or max pooling, *cf.* Section 3.2.3. It additionally offers more sophisticated methods as well, such as, *e.g.*, set-to-set pooling (Vinyals *et al.*, 2016) or sort-based pooling (Zhang *et al.*, 2018). To further extract hierarchical information and to allow for deeper GNN models, various hierarchical pooling approaches can be applied in a spatial or data-dependent manner, either being non-trainable or trainable. In general, pooling operators follow a unified and modular framework as well, in which **(1)** nodes are first selected to map to one (or more) "supernodes", **(2)** supernodes are reduced to singletons, **(3)** and reduced nodes are then newly connected to each other (Grattarola *et al.*, 2021).

In addition to these low-level operators, we provide high-level implementations and models of, *e.g.*, maximizing mutual information (Veličković *et al.*, 2019), auto-encoding graphs (Kipf & Welling, 2016; Pan *et al.*, 2018), aggregating Jumping Knowledge (Xu *et al.*, 2018), or predicting temporal events in knowledge graphs (Jin *et al.*, 2020). Additional examples are provided for unsupervised and self-supervised learning on graphs, few/zero-shot learning, pre-training, and explainability. Overall, our predefined GNN models incorporate multiple message passing layers, and are ready-to-be-used to make accurate predictions on graphs. Unlike simple stacking of GNN layers, these models may provide dedicated solutions for specific tasks, such as for the task of learning on molecular graphs, and may involve important pre-processing steps, additional learnable parameters, or graph coarsening procedures.

PyG further provides a rich set of neural network operators that are commonly used in many GNN models. They follow an extensible design: It is easy to apply these operators and graph utilities to existing GNN layers to further enhance model performance. Lastly, PyG has integrated a diverse set of scalability techniques stemming from the fields of node-wise, layer-wise and subgraph-wise sampling, *cf.* Section 5.2. In general, these techniques follow a unified interface, and can be thus directly used as plug-and-play to scale any PyG model to larger graphs. All scalability techniques are implemented via low-level C++ routines to make them as efficient as possible.

We provide a consistent data format and an easy-to-use interface for the creation and processing of datasets, both for large datasets and for datasets that can be kept in memory during training. In order to create new datasets, users just need to read or download their given raw data and convert it to the PyG data format in the respective `process` method. In addition, datasets can be modified by the usage of `transforms`, which take in individual graphs and return a transformed version, *e.g.*, for data augmentation, for enhancing node features with synthetic structural graph properties (Cai & Wang, 2018), to automatically generate graphs from point clouds or to sample point clouds from meshes (Fey & Lenssen, 2019).

Notably, PyG already integrates a rich set of over 300 common benchmark datasets often found in literature. All datasets are automatically downloaded and processed on first instantiation. For example, we provide the complete set of graphs from the TU-

```
import torch
from torch_geometric.nn import GCNConv
from torch_geometric.datasets import Planetoid

class GCN(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels):
        super().__init__()
        self.conv1 = GCNConv(in_channels, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, out_channels)

    def forward(self, x, edge_index):
        # x: Node feature matrix of shape [num_nodes, in_channels]
        # edge_index: Graph connectivity matrix of shape [2, num_edges]
        h1 = self.conv1(x, edge_index).relu()
        h2 = self.conv2(h1, edge_index)
        return h2

model = GCN(dataset.num_features, 16, dataset.num_classes).cuda()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

dataset = Planetoid(name="Cora", transform=T.NormalizeFeatures())
data = dataset[0].cuda()

pred = model(data.x, data.edge_index)
loss = F.cross_entropy(pred[data.train_mask], data.y[data.train_mask])

loss.backward()
optimizer.step()
```

Listing 1: High-level creation of a GCN model within PyG. The model is trained in a semi-supervised fashion to classify academic papers in a citation graph.

Dataset, Open Graph Benchmark (OGB) and MoleculeNet benchmark suites (Morris *et al.*, 2020a; Hu *et al.*, 2020a; Wu *et al.*, 2018), commonly used citation graphs (Sen *et al.*, 2008; Yang *et al.*, 2016; Bojchevski & Günnemann, 2018), or various mesh and point cloud datasets (Bogo *et al.*, 2014; Wu *et al.*, 2015; Chang *et al.*, 2015; Ranjan *et al.*, 2018; Guerrero *et al.*, 2018).

6.3.1.2 High-level Usage. We highlight the ease of creating a GNN model in PyG and training it in a semi-supervised fashion in order to classify academic papers in a citation graph, *cf.* Listing 1. For this, we create a simple two-layer GCN model (Kipf & Welling, 2017) by making use of the pre-defined `GCNConv`, and instantiate it by specifying the number of input, hidden and output features. Here, the GCN model's forward computation expects a node feature matrix and a graph connectivity matrix as input, *cf.* Section 6.3.2. We further move all model state to the GPU and pass its parameters to the Adam optimizer (Kingma & Ba, 2015). Afterwards, we load the Cora dataset from the `Planetoid` benchmark suite (Yang *et al.*, 2016), normalize its node features via a custom transformation (`NormalizeFeatures`), and move its data to the GPU as well. We now optimize model parameters using standard PyTorch training: **(1)** We run

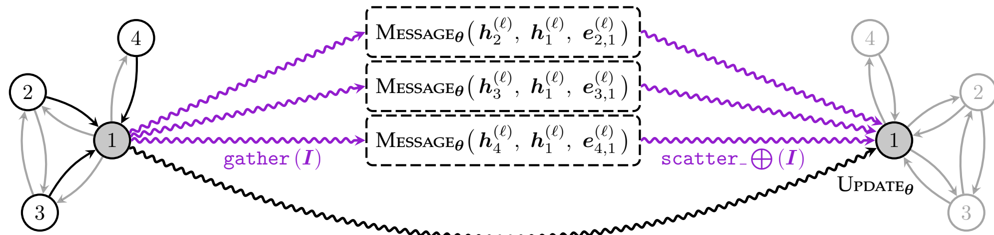


Figure 6.2: Computation scheme of a GNN layer in PyG. PyG leverages `gather` and `scatter` methods based on edge indices $\boldsymbol{I}$ for performing alternating computations in node-parallel and edge-parallel space (Fey & Lenssen, 2019).

the forward computation on the full graph, **(2)** compute the negative log-likelihood loss based on the ground-truth training node labels, **(3)** accumulate the gradients of model parameters via backpropagation, and **(4)** update parameters via a variant of stochastic gradient descent.

6.3.2 A Unified Interface for Graph Neural Networks

PyG represents a graph $\mathcal{G} = (\boldsymbol{X}, (\boldsymbol{I}, \boldsymbol{E}))$ by a node feature matrix $\boldsymbol{X} \in \mathbb{R}^{N \times F}$ of N nodes holding F-dimensional node features, and a *sparse* adjacency tuple $(\boldsymbol{I}, \boldsymbol{E})$ of E edges, where $\boldsymbol{I} \in \mathbb{N}^{2 \times E}$ encodes edge indices in COOrdinate (COO) format and $\boldsymbol{E} \in \mathbb{R}^{E \times D}$ (optionally) holds D-dimensional edge features.

6.3.2.1 Message Passing. PyG provides the user with a general `MessagePassing` interface to allow for rapid and clean prototyping of new research ideas. In particular, the `MessagePassing` interface allows to define arbitrary message passing GNN layers as defined in Equation (3.5) by decomposing Message, Update and aggregative functions $\bigoplus$ into customizable user defined functions. In order to use, users only need to define the methods `message`, and `update`, as well as choosing an appropriate aggregation scheme $\bigoplus \subseteq \{\texttt{add}, \texttt{mean}, \texttt{min}, \texttt{max}, \texttt{mul}, \texttt{std}\}$.

Although working on irregular structured input, this scheme can be heavily accelerated by the GPU, *cf.* Figure 6.2. In general, the message passing GNN framework of Equation (3.5) provides two dimensions of parallelization: the edge-parallel space for computing the Message function, *i.e.* computing a message *per* edge, and the node-parallel space to compute the Update function, *i.e.* updating the representation *per* node, which are performed alternating in a general GNN pipeline. We can easily switch between both spaces via efficient pseudo-parallel `scatter_`$\bigoplus$ and `gather` operations. In particular, a `gather` operation performs a parallel read using E processing cores on the GPU, where each core operates on a single edge $e \in \{1, \ldots, E\}$ that reads the features of source and destination nodes $\boldsymbol{X}[\boldsymbol{I}[1, e], :]$ and $\boldsymbol{X}[\boldsymbol{I}[2, e], :]$ from memory, respectively. Similarly, a `scatter_`$\bigoplus$ operation performs a parallel aggregation using E processing cores based on an atomic reduce operation $\bigoplus$, *e.g.*, taking the sum or the maximum. Given edge-level messages $\boldsymbol{M} \in \mathbb{R}^{E \times F}$ and a neutrally initialized output matrix $\boldsymbol{X}' \in \mathbb{R}^{N \times F}$ as input, each core $e \in \{1, \ldots, E\}$ combines the message $\boldsymbol{M}[j, :]$ with the value at $\boldsymbol{X}'[\boldsymbol{I}[2, j], :]$, thus performing a parallel neighborhood ag-

gregation. Notably, multiple cores can access the same node-level index concurrently, which requires that the aggregation is performed using atomic operations. We refer to `scatter_add` or `scatter_max` to denote scatter operations using sum or maximum atomic operations, respectively. For `scatter_mean`, we decompose computation into two `scatter_add` aggregations, which sum up and count all values pointing to the same destination index, respectively, followed by a node-level normalization. In total, the full computation scheme has a parallel time complexity of $\mathcal{O}(1)$ using $\mathcal{O}(E)$ processing cores, assuming that `scatter_`$\bigoplus$ has constant time complexity (Fey *et al.*, 2018). Noteworthy, `scatter_`$\bigoplus$ is a non-deterministic operation, since the order of parallel operations to the same index is undetermined. For floating-point values, this results in a source of variance in the result. More sophisticated GPU aggregation procedures counteract non-determinism by requiring *sorted inputs*, *i.e.* indices pointing to the same destination are grouped contiguously in memory, *cf.* Section 6.3.3. However, we argue that such induced non-determinism is negligible (and may potentially even boost performance), as the order of neighbors is already undetermined to begin with. Our implementation of parallel scatter operations using custom reductions is open-sourced on GitHub[6], comes pre-build for all major OS/Python/PyTorch/CUDA combinations, and supports broadcasting capabilities, varying data types, traceability, and both CPU/GPU computation with corresponding backward implementations.

In contrast to alternative implementations via sparse matrix multiplications, the usage of gather and scatter operations proves to be advantageous for low-degree graphs and non-contiguous indices (Fey & Lenssen, 2019). Furthermore, it results in the most general implementation of message passing since it easily allows for the integration of central node and multi-dimensional edge features during edge-parallel message computation. However, it will necessarily *materialize* all messages in the edge-parallel space, leading to a total memory consumption of $\mathcal{O}(E)$, which might be impractical for some applications. We will look into advanced optimizations in order to reduce the memory footprint of message passing in Section 6.3.3.

The PyG `MessagePassing` interface provides an intuitive way to write generic GNN layer implementations by abstracting away the low-level details of internal scatter and gather usages. As an example, Listing 2 illustrates the implementation of the edge-convolutional layer (Wang *et al.*, 2019e)

$$\boldsymbol{h}_v^{(\ell)} = \max_{w \in \mathcal{N}(j)} \mathrm{MLP}_{\boldsymbol{\theta}}^{(\ell)} \left(\left[\boldsymbol{h}_v^{(\ell-1)}, \boldsymbol{h}_w^{(\ell-1)} - \boldsymbol{h}_v^{(\ell-1)} \right] \right). \tag{6.1}$$

Here, we first specify which aggregation scheme to use (`aggr="max"`). In the forward computation graph of the layer, we then start to perform message passing via `propagate()`, which takes care of automatically gathering node-level features to the edge-parallel space, computing edge-level messages in parallel inside `message()`, and automatically scattering them back to the node-parallel space via `scatter_max`.

6.3.2.2 Mini-batch Handling. The PyG framework supports mini-batch creation for operating both on many small and single giant graphs. When operating on a dataset that holds multiple graph instances, the creation of mini-batches is crucial for letting the training of a deep learning model scale to huge amounts of data. Instead of processing examples one-by-one, a mini-batch groups a set of examples into a unified representation where it can efficiently be processed in parallel. In the image or language

[6] `torch-scatter`: `https://github.com/rusty1s/pytorch_scatter` (last access: August 25, 2022)

```
import torch
from torch.nn import Sequential, Linear, ReLU, BatchNorm1d
from torch_geometric.nn import MessagePassing

class EdgeConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super().__init__(aggr="max")  # "Max" aggregation.

        self.MLP = Sequential(
            Linear(2 * in_channels, out_channels),
            BatchNorm1d(out_channels),
            ReLU(),
            Linear(out_channels, out_channels),
        )

    def forward(self, h, edge_index):
        # h: Node feature matrix of shape [num_nodes, in_channels]
        # edge_index: Graph connectivity matrix of shape [2, num_edges]
        h = self.propagate(edge_index, h=h)
        return h  # shape [num_nodes, out_channels]

    def message(self, h_v, h_w):
        # h_v: Destination node features of shape [num_edges, in_channels]
        # h_w: Source node features of shape [num_edges, in_channels]
        msg = torch.cat([h_v, h_w - h_v], dim=1)
        msg = self.MLP(msg)
        return msg  # shape [num_edges, out_channels]
```

Listing 2: PyG message passing implementation of the edge-convolutional layer (Wang *et al.*, 2019e) **via custom message and aggregation definitions.** Node-level features are automatically gathered to edge-parallel space by appending `*_v` and `*_w` suffices to the original attribute name, denoting destination and source node features, respectively.

domain, this procedure is typically achieved by rescaling or padding each example to equally-sized shapes, and group them in a dedicated *batch dimension* afterwards. However, in the graph domain, these two approaches are either not feasible or waste huge amounts of GPU memory, as graphs may highly vary in their number of nodes and edges across examples. PyG opts for another approach to achieve parallelization across a number of examples. In general, our mini-batching strategy behaves similar to a *nested tensor* implementation[7] with off-the-shelf support for sparse matrices, which is, however, not yet available in any deep learning framework up to this date. Here, adjacency matrices are grouped by merging them into a single (sparse) block-diagonal adjacency matrix (representing a giant graph that holds multiple isolated subgraphs), and concatenating node-level feature matrices and labels in the node dimension, *cf.* Figure 6.3. This procedure provides crucial advantages over alternative batching formulations:

[7] `nestedtensor`: `https://github.com/pytorch/nestedtensor` (last access: August 25, 2022)

$\mathcal{G}_1 = (\boldsymbol{X}_1, (\boldsymbol{I}_1, \boldsymbol{E}_1))$

$\mathcal{G}_2 = (\boldsymbol{X}_2, (\boldsymbol{I}_2, \boldsymbol{E}_2))$

$$\mathrm{GNN}_{\boldsymbol{\theta}}\left(\begin{bmatrix} \boldsymbol{X}_1 \\ \boldsymbol{X}_2 \end{bmatrix}, \begin{bmatrix} (\boldsymbol{I}_1, \boldsymbol{E}_1) & \\ & (\boldsymbol{I}_2, \boldsymbol{E}_2) \end{bmatrix} \right) = \begin{bmatrix} \boldsymbol{H}_1 \\ \boldsymbol{H}_2 \end{bmatrix}$$

Figure 6.3: Illustrative mini-batch creation of multiple graphs in PyG. Adjacency matrices are merged into a single (sparse) block-diagonal adjacency matrix, and node-level features are concatenated in the node dimension. Message passing GNNs can then be applied without modification, since no messages will be exchanged between disconnected subgraphs.

- Message passing GNN operators can be applied without modification, since no messages will be exchanged between disconnected subgraphs.
- There is no memory or computational overhead since adjacency matrices are saved in a sparse fashion holding only non-zero enries.
- It naturally supports examples of varying size.

PyG takes care of batching multiple graph instances into a single giant graph within the `DataLoader` class such that edge indices are automatically incremented by the cumulated number of nodes of all previously processed graphs in the mini-batch. In addition, it will generate a node-level assignment vector

$$\boldsymbol{b} = [0, \ldots, 0, 1, \ldots, 1, \ldots, B-1, \ldots, B-1] \in \mathbb{N}^N \tag{6.2}$$

that maps each node to its respective graph in the mini-batch, with B denoting the batch size. This assignment vector ensures that nodes in different graphs are well distinguishable from each other, and finds its use-cases in, *e.g.*, global readout operators or hierarchical pooling models. Pooling operators can then again make usage of `scatter_`$\bigoplus$ operations to aggregate node features according to $\boldsymbol{b}$.

PyG further provides unified interfaces to scale GNN training to single giant graphs, such that GNN models can be easily validated on small-scale graphs, and then applied to large-scale graphs with minimal requirements of user intervention. In particular, PyG aims to *disentangle* the underlying GNN implementation from the utilized scalability technique (following the design principles of our GAS framework (Fey *et al.*, 2021)), which is not the case in alternative deep graph libraries (Wang *et al.*, 2019b). Listing 3 illustrates the process of scaling arbitrary GNN models via PyG. Here, commonly encountered CUDA out-of-memory errors on large-scale graphs are avoided by sampling or partitioning the graph into mini-batches, and optimizing model parameters in a training loop. The large number of scalability techniques available in PyG (Hamilton *et al.*, 2017; Chiang *et al.*, 2019; Zeng *et al.*, 2020b) lets the user easily detect the most suitable technique for the given down-stream task.

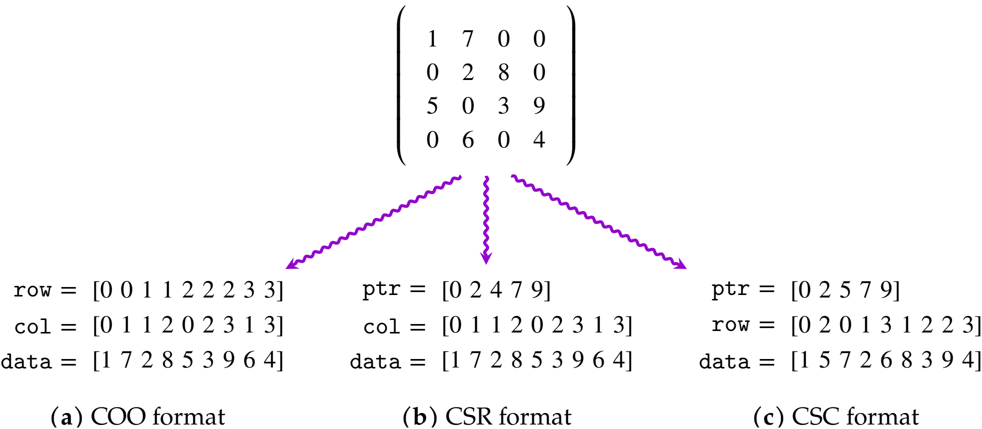
```python
def train(data):  # Full-batch training
    pred = model(data.x, data.edge_index)
    loss = F.cross_entropy(pred, data.y)
    # >>> RuntimeError: CUDA out-of-memory

# Choose a scalability technique:
loader = NeighborLoader(data, num_neighbors=[10, 10], batch_size=128)
loader = ClusterLoader(data, num_parts=1280, batch_size=128)
loader = GraphSAINTLoader(data, batch_size=128)

def train(loader):  # Mini-batch training
    for data in loader:
        pred = model(data.x, data.edge_index)
        loss = F.cross_entropy(pred, data.y)
```

Listing 3: Illustration of applying scalability techniques to any PyG GNN model, requiring only minimal user intervention. CUDA out-of-memory errors of full-batch training are avoided by sampling or partitioning the graph into mini-batches, and optimizing model parameters in a training loop.

6.3.3 Efficient Sparse Tensor Arithmetic

PyG leverages custom sparse tensor arithmetics (both on CPU and GPU) whenever such functionality can not be achieved by a sequence of vectorized operations in an efficient and high-throughput manner. This usually relates to many things in a given Graph Neural Network pipeline, ranging from breadth-first sampling (neighborhood sampling) and depth-first sampling (random walk sampling), to general sparse aggregation procedures such as `scatter`, `segment` or *general sparse matrix multiplications* (*G-SpMMs*), to efficient pooling and grouping algorithms. For all these use-cases, PyG has out-sourced their respective implementations to one of its three extension packages (*cf.* Section 6.3.1), and follows best-practices regarding broadcasting capabilities and varying data type support (*e.g.*, half-precision), and implements both forward and backward implementations whenever applicable.

In this chapter, we will take a closer look at how PyG achieves high GPU throughput via the CUDA parallel computing architecture (Lindholm *et al.*, 2008; Nickolls *et al.*, 2008). A CUDA program consists of a host program that runs on the CPU, sets up the data and transfers it to and from the GPU, and a *kernel* that executes the main processing task on the GPU itself (Filippone *et al.*, 2017). In particular, GPUs utilize a *Single Instruction Multiple Data* (*SIMD*) architecture, such that the same set of instructions is scheduled across a massive collection of *parallel threads*. These threads are organized into *blocks*, in which they can share on-chip low-latency memory and synchronize with hardware barrier instructions (Bell & Garland, 2009). Each thread in a block is also assigned to a *warp* (32 threads per warp), in which a single instruction is executed at a time across all its threads. Although *execution divergence* of threads inside the same warp is possible, it is substantially more efficient for threads in a warp to follow the same execution path (Bell & Garland, 2009). Furthermore, threads inside a warp can heavily benefit from coalesced memory accesses, such that each thread reads from or writes to the same region in GPU memory, thus avoiding *memory di-*



(**a**) COO format (**b**) CSR format (**c**) CSC format

Figure 6.4: Various sparse matrix representations for a simple example matrix.

vergence. Non-coalesced memory accesses reduces bandwidth efficiency and lead to memory-bound kernel execution, and hence should be avoided at all costs (Bell & Garland, 2009). As a result, the main challenge for sparse tensor arithmetic on GPUs stems from the mismatch between its SIMD architecture and the irregular data access pattern of sparse matrices, requiring effective solutions that successfully prevent both execution and memory divergence. While most of the techniques presented in the following are inherited from previous works, they have been successfully implemented in a general and unified way to fit into the context of Graph Representation Learning within the PyTorch Geometric library.

6.3.3.1 Sparse Matrix Storage Formats. A sparse matrix denotes a matrix with a sufficient number of zero valued entries that it pays to take advantage of them (Davis, 2007), *i.e.* $E \ll N \times N$, *e.g.*, by avoiding explicit storage and faster processing. In fact, adjacency matrices of graphs are typically sparse by nature, and hence require sparse compute processing as well. While dense matrices are ultimately stored in a one-dimensional array (using column-major or row-major ordering) and use a linear mapping to efficiently map index pairs to unique memory locations, such a mapping is destroyed in sparse matrices at the cost of minimizing storage requirements (Filippone *et al.*, 2017). Hence, many sparse matrix storage formats have been invented over the years, each one with its own trade-off regarding storage requirements, usability and processing capabilities, *cf.* Figure 6.4.

The *COOrdinate* (*COO*) format utilizes the most simple storage scheme, defined by three distinct vectors holding the row indices, column indices and the explicit values of non-zero valued entries, respectively. As such, it is able to reduce storage costs from $N \times N$ to $3E$. The COO format is typically used to construct sparse matrices, as there is no underlying assumption on its data layout, *e.g.*, new non-zero entries can simply be added by appending them to each of the three vectors. However, such missing assumption limits its applicability for sparse matrix processing, *e.g.*, finding one of its elements requires $\mathcal{O}(E)$ look-ups at worst. Notably, full-matrix GPU processing with COO format can still be performed efficiently via atomic operations (*cf.* Section 6.3.2), and is as such the default format in the PyG library, especially due to its ease-of-use.

Algorithm 7 `scatter_`$\bigoplus$ kernel (COO reduction)

Require: Messages $\boldsymbol{M} \in \mathbb{R}^{E \times F}$, Column indices $\boldsymbol{c} \in \mathbb{N}^{E}$, Output $\boldsymbol{X}' \in \mathbb{R}^{N \times F}$
1: $e \leftarrow$ `threadIdx / ` F
2: $f \leftarrow$ `threadIdx %` F
3: `atomic_`$\bigoplus(\boldsymbol{X}'[\boldsymbol{c}[e], f], \boldsymbol{M}[e, f])$ — Coalesced read and write

Algorithm 8 `segment_`$\bigoplus$ kernel (CSC/CSR reduction)

Require: Messages $\boldsymbol{M} \in \mathbb{R}^{E \times F}$, Boundary indices $\boldsymbol{p} \in \mathbb{N}^{N+1}$, Output $\boldsymbol{X}' \in \mathbb{R}^{N \times F}$
1: $n \leftarrow$ `threadIdx / ` F
2: $f \leftarrow$ `threadIdx %` F
3: $v \leftarrow \mathrm{id}$ — Initialize with the identity of $\bigoplus$
4: **for** $e \leftarrow \boldsymbol{p}[n]$ **to** $\boldsymbol{p}[n+1]$ **do**
5: $\quad v \leftarrow v \bigoplus \boldsymbol{M}[e, f]$ — Coalesced read
6: **end for**
7: $\boldsymbol{X}'[n, f] \leftarrow v$ — Coalesced write

The *Compressed Sparse Row* (*CSR*) and *Compressed Sparse Column* (*CSC*) storages mitigate slow indexing by relying on row-major and column-major ordering of indices, respectively. Their names are based on the fact that index information is compressed in such a way that only boundary indices of rows and columns are stored, respectively, which reduces storage costs further to $2E + N + 1$. The fast access to boundary offsets and row-major/column-major ordering requirements yield the advantage of *constant* time look-ups of direct neighbors, which makes them the most popular formats for sparse matrix processing. However, modifying CSR or CSC matrices in a later stage can get expensive, as elements need to be inserted at dedicated indices, and boundaries have to be modified accordingly.

COO, CSR and CSC denote "general-purpose" data formats, while more specialized formats for dedicated sparse matrix layouts are available as well, *i.e.* the *diagonal* (*DIA*), *ELLPACK*, *Jagged Diagonals* (*JAD*) or *block-based* (*BSR*) formats (Filippone *et al.*, 2017). However, they are only applicable in certain scenarios, *e.g.*, they require nearly uniform degree distributions or dense sub-matrix separation. Therefore, they are not further considered in the remainder of this section.

6.3.3.2 Parallel Reductions. Different reduction schemes of edge-level messages are applicable in a GNN pipeline, dependent on the given sparse matrix storage layout. For COO matrices, an edge-level message matrix $\boldsymbol{M} \in \mathbb{R}^{E \times F}$ can be reduced to a node-level output matrix by utilizing the `scatter_`$\bigoplus$ kernel, *cf.* Section 6.3.2. Here, `scatter_`$\bigoplus$ leverages $E \times F$ threads and performs reduction via atomic operations as illustrated in the pseudo CUDA kernel implementation in Algorithm 7. Notably, in case the number of features F is divisible by 32 (the number of threads in a warp), `scatter_`$\bigoplus$ utilizes coalesced memory accesses both for $\boldsymbol{M}$ and the output matrix $\boldsymbol{X}'$, thus completely avoiding memory divergence.

To counteract the non-determinism of `scatter_`$\bigoplus$, parallel reductions can also be performed via the CSC/CSR format, leading to the formulation of a `segment_`$\bigoplus$ kernel, *cf.* Algorithm 8. `segment_`$\bigoplus$ can leverage the fact that the elements to reduce are grouped contiguously in memory, as denoted by the boundary vector $\boldsymbol{p} \in \mathbb{N}^{N+1}$.

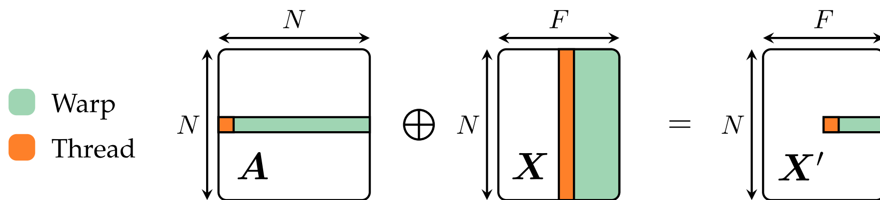


Figure 6.5: Memory access pattern of G-SpMM. Threads inside a warp operate on a single row in $\boldsymbol{A}$ and a feature chunk in $\boldsymbol{X}$ to write a chunk of a single row in $\boldsymbol{X}'$. All accesses are performed in a coalesced manner.

While there exists a broad range of literature on performing segment reductions of vectors by utilizing efficient *intra-warp communication* (Blelloch *et al.*, 1993, 1994; Sengupta *et al.*, 2007; Bell & Garland, 2009), these kernels are not applicable in a multi-dimensional reduction scenario due to the row-major storage format of the edge-level message matrix $\boldsymbol{M}$. Furthermore, we found the usage of shared memory/synchronization barriers unnecessary and overly complicated to implement an efficient `segment_`$\bigoplus$ kernel. As such, our kernel simply leverages $N \times F$ threads such that each thread processes exactly one row. Due to the deterministic ordering of column indices, global determinism is achieved as well. Importantly, both execution and memory divergence is fully eliminated in case the number of features F is divisible by 32, as each thread inside a warp will operate on the same row and therefore takes the same execution path. We can utilize the `segment_`$\bigoplus$ kernel on either CSC and CSR formats to denote source node aggregations and destination node aggregations, respectively.

6.3.3.3 Memory-Efficient Aggregations via G-SpMM. While the unified `gather-scatter` formulation of PyG generalizes to a lot of useful GNN instantiations (Section 6.3.2), it has the disadvantage of explicitly materializing edge-level message vectors. Overall, this results in a total memory footprint of $\mathcal{O}(E)$, which might be impractical for applications on large or dense graphs. Luckily, not all GNNs need to necessarily materialize edge-level tensors, but can instead be implemented by *kernel fusion*, reducing memory requirements to $\mathcal{O}(N)$. For example, when Message simply returns the source node's features and $\bigoplus$ is defined to be the sum operator, the overall computation is equivalent to a sparse matrix multiplication (Kipf & Welling, 2017; Hamilton *et al.*, 2017; Veličković *et al.*, 2018; Xu *et al.*, 2019c; Wang *et al.*, 2019b). As a general rule of thumb, this holds true for GNNs that do not make use of central node features or multi-dimensional edge features during message propagation, resulting in a simplification of message passing that makes small trades in accuracy for a large boost in efficiency (Wu *et al.*, 2019a; Tailor *et al.*, 2021).

PyG allows for fused message passing computation in a memory-efficient manner via its `SparseTensor` class and its `message_and_aggregate` directive. The `SparseTensor` class can represent sparse matrices seamlessly in either COO/CSR/CSC storage format by utilizing caching mechanics, and integrates a simple interface for *general sparse matrix multiplications* (*G-SpMMs*) of the form $\boldsymbol{AX}$ (Huang *et al.*, 2020), in which matrix multiplications are generalized such that they allow for customizable reduction operators $\bigoplus \subseteq \{\texttt{add}, \texttt{mean}, \texttt{min}, \texttt{max}, \texttt{mul}\}$. Our G-SpMM kernel is implemented by following the design principles of Yang *et al.* (2018), since the vendor-shipped cuSPARSE library (Naumov *et al.*, 2010) does not provide support for general sparse ma-

Algorithm 9 G-SpMM kernel (Yang *et al.*, 2018)

Require: Row boundary indices $\boldsymbol{p} \in \mathbb{N}^{N+1}$, Column indices $\boldsymbol{c} \in \mathbb{N}^{E}$, Weights $\boldsymbol{w} \in \mathbb{R}^{E}$, Matrix $\boldsymbol{X} \in \mathbb{R}^{N \times F}$, Output $\boldsymbol{X}' \in \mathbb{R}^{N \times F}$

```
 1: n ← threadIdx.x / 32                                          ▷ The row index
 2: ℓ ← threadIdx.x % 32                  ▷ The lane index inside the thread's warp
 3: f ← threadIdx.y * 32                   ▷ The column offset of the feature chunk
 4: v ← id                                      ▷ Initialize with the identity of ⨁
 5: cols[32], weights[32] ← ∅
 6: for e ← p[n] to p[n + 1] by 32 do
 7:     c ← c[e + ℓ]                                               ▷ Coalesced read
 8:     w ← w[e + ℓ]                                               ▷ Coalesced read
 9:     for i ← 1 to 32 do     ▷ Broadcast columns/weights to each thread in the warp
10:         cols[i] ← shuffle(c, i)
11:         weights[i] ← shuffle(w, i)
12:     end for
13:     for i ← 1 to 32 do
14:         v ← v ⨁ (weights[i] * X[cols[i], f + ℓ])                ▷ Coalesced read
15:     end for
16: end for
17: X'[n, f + ℓ] ← v                                               ▷ Coalesced write
```

trix multiplications. It expects CSR sparse matrices and row-major ordering of dense matrices as input. For backpropagation, we simply transpose the sparse matrix (using its cached CSC representation), and perform G-SpMM once again.

Here, each row in $\boldsymbol{A}$ is assigned to a warp, which processes a row-wise chunk of 32 features in $\boldsymbol{X}$, *cf.* Figure 6.5 and Algorithm 9. This leads to a total of $32N \cdot \lceil F/32 \rceil$ threads, executed in two dimensions: the row dimension $(32N)$, and the feature dimension $(\lceil F/32 \rceil)$. For a given row in $\boldsymbol{A}$ and a feature chunk in $\boldsymbol{X}$, a warp first accesses its column indices and values in $\boldsymbol{A}$ in a coalesced manner, which are then shared across all threads in a warp via intra-warp communication. Afterwards, each warp iterates over its locally shared column indices, and performs a row-major coalesced read into the feature chunk in $\boldsymbol{X}$. Each thread aggregates its intermediate values according to $\bigoplus$, and finally writes its output to $\boldsymbol{X}'$ in a coalesced manner. Notably, this memory access pattern achieves coalesced accesses for both input matrices and output matrices, and is crucial for reaching excellent performance (Yang *et al.*, 2018). Its key component is a round of 32 broadcasts (using the `shuffle` warp intrinsic) by each thread to inform all other threads in the warp. This is required as otherwise each thread would be responsible for loading its own row, which would result in uncoalesced accesses in $\boldsymbol{A}$ (Yang *et al.*, 2018).

6.3.3.4 Random Walk Sampling. Node embedding techniques (Perozzi *et al.*, 2014; Grover & Leskovec, 2016) rely on fast random walk samplers to embed nodes into low-dimensional vectorial representations, *cf.* Section 2.2. In case the full graph structure fits into GPU memory, we can leverage a dedicated CUDA kernel to achieve maximal parallelism. Given a mini-batch of nodes $\boldsymbol{b} \in \mathbb{N}^{B}$ as starting nodes, we parallelize across $\boldsymbol{b}$ and achieve full coalesced memory accesses by sequentially writing randomly sampled neighbors into an output matrix $\boldsymbol{W} \in \mathbb{N}^{L \times B}$, where L denotes the length

Algorithm 10 Random walk kernel

Require: Row boundary indices $\boldsymbol{p} \in \mathbb{N}^{N+1}$, Column indices $\boldsymbol{c} \in \mathbb{N}^{E}$, Starting nodes $\boldsymbol{b} \in \mathbb{N}^{B}$, Output $\boldsymbol{W} \in \mathbb{N}^{L \times B}$

1: $n \leftarrow \boldsymbol{b}[\texttt{threadIdx}]$ — Coalesced read
2: **for** $\ell \leftarrow 1$ **to** L **do**
3: $\quad r \sim \mathcal{U}(0, 1)$
4: $\quad n \leftarrow \boldsymbol{c}[\boldsymbol{p}[n] + \lfloor r \cdot (\boldsymbol{p}[n+1] - \boldsymbol{p}[n]) \rfloor]$
5: $\quad \boldsymbol{W}[\ell, \texttt{threadIdx}] \leftarrow n$ — Coalesced write
6: **end for**

of the random walk, *cf.* Algorithm 10. Notably, CSR format is required to allow for random sampling of neighbors in $\mathcal{O}(1)$.

In the same spirit, we have integrated a biased breadth-first or depth-first sampler via a *rejection sampling* strategy (Abraham, 2020) for implementing the Node2Vec model (Grover & Leskovec, 2016). However, while the general parallel computation scheme across a mini-batch of nodes still holds, it might lead to unavoidable execution divergence due to the nature of rejection sampling.

6.3.3.5 Parallel Graph Clustering. Efficient graph clustering and coarsening strategies are required to allow GNNs to learn multi-scale representations, *cf.* Section 4.3. Defferrard *et al.* (2016) propose to utilize the Graclus coarsening strategy (Karypis & Kumar, 1998; Dhillon *et al.*, 2007), which reduces the size of a graph by a factor of two at each level, and therefore offers precise control on the coarsening and pooling sizes. Graclus is a greedy clustering algorithm, which matches every unmatched node v with one of its unmatched neighbors w such that an objective is maximized, *e.g.*, the local normalized cut $A_{v,w}(1/|\mathcal{N}(v)| + 1/|\mathcal{N}(w)|)$ (Defferrard *et al.*, 2016). The matching phase is repeated until all nodes have been successfully matched or until there no longer exists an unmatched neighbor for any node. While Defferrard *et al.* (2016) propose to compute the clustering phase in a CPU pre-processing step, we found this to be undesirable as the neural network will only be able to explore a single matching, although the Graclus strategy is inherently random, *e.g.*, it depends on the processing order of nodes. However, performing Graclus clustering in an end-to-end fashion requires an efficient GPU implementation, which is challenging, in particular due to the serial nature of the algorithm.

PyG offers a dedicated CUDA kernel to perform Graclus clustering based on the design principles introduced by Fagginger Auer & Bisseling (2011), which explicitly avoids the need to consider nodes one-by-one. The CUDA kernel is based on a “propose” and “respond” strategy, which are run in an alternating fashion until all nodes have been matched successfully, *cf.* Algorithms 11 and 12. For this, it keeps track of node colorings (■, ■) for unmatched nodes, which are used to propose to unmatched neighbors of different color as potential matches. Nodes to which a proposal has been made can then accept the proposal of a single neighbor in a second stage. At the beginning, all nodes are initialized as either ■ or ■ nodes at random. The “propose” kernel parallelizes over all unmatched ■ nodes and propose to an unmatched ■ neighbor with the highest edge weight. If all neighbors have already been successfully matched, it will denote itself as a singleton cluster. The “respond” kernel then responds to the proposal by parallelizing over all unmatched ■ nodes and matches

Algorithm 11 GRACLUS propose kernel (Fagginger Auer & Bisseling, 2011)

Require: Row boundary indices $\boldsymbol{p} \in \mathbb{N}^{N+1}$, Column indices $\boldsymbol{c} \in \mathbb{N}^{E}$, Weights $\boldsymbol{w} \in \mathbb{R}^{E}$, Proposal $\boldsymbol{\sigma} \in \mathbb{N}^{N}$, Output $\boldsymbol{\pi} \in (\{🟦, 🟧\} \cup \mathbb{N})^{N}$

1: $n \leftarrow$ `threadIdx`
2: **if** $\boldsymbol{\pi}[n] =$ 🟦 **then** — 🟦 nodes propose to 🟧 neighbors
3: $\quad w_{\max} \leftarrow 0$
4: $\quad$ `dead` $\leftarrow$ `true`
5: $\quad$ **for** $e \leftarrow \boldsymbol{p}[n]$ **to** $\boldsymbol{p}[n+1]$ **do**
6: $\quad\quad m \leftarrow \boldsymbol{c}[e]$
7: $\quad\quad$ **if** $\boldsymbol{\pi}[m] =$ 🟧 **and** $\boldsymbol{w}[e] > w_{\max}$ **then**
8: $\quad\quad\quad \boldsymbol{\sigma}[n] \leftarrow m$ — Propose to 🟧 neighbor with highest weight
9: $\quad\quad\quad w_{\max} = \boldsymbol{w}[e]$
10: $\quad\quad$ **end if**
11: $\quad\quad$ **if** $\boldsymbol{\pi}[m] \in \{$🟦, 🟧$\}$ **then** `dead` $\leftarrow$ `false` **end if**
12: $\quad$ **end for**
13: $\quad$ **if** `dead` **then** $\boldsymbol{\pi}[n] = n$ **end if** — All neighbors are matched $\rightarrow$ singleton
14: **end if**

Algorithm 12 GRACLUS respond kernel (Fagginger Auer & Bisseling, 2011)

Require: Row boundary indices $\boldsymbol{p} \in \mathbb{N}^{N+1}$, Column indices $\boldsymbol{c} \in \mathbb{N}^{E}$, Weights $\boldsymbol{w} \in \mathbb{R}^{E}$, Proposal $\boldsymbol{\sigma} \in \mathbb{N}^{N}$, Output $\boldsymbol{\pi} \in (\{🟦, 🟧\} \cup \mathbb{N})^{N}$

1: $n \leftarrow$ `threadIdx`
2: **if** $\boldsymbol{\pi}[n] =$ 🟧 **then** — 🟧 nodes respond to 🟦 neighbors
3: $\quad m_{\text{best}} \leftarrow \emptyset$
4: $\quad w_{\max} \leftarrow 0$
5: $\quad$ `dead` $\leftarrow$ `true`
6: $\quad$ **for** $e \leftarrow \boldsymbol{p}[n]$ **to** $\boldsymbol{p}[n+1]$ **do**
7: $\quad\quad m \leftarrow \boldsymbol{c}[e]$
8: $\quad\quad$ **if** $\boldsymbol{\pi}[m] =$ 🟦 **and** $\boldsymbol{\sigma}[m] = n$ **and** $\boldsymbol{w}[e] > w_{\max}$ **then**
9: $\quad\quad\quad m_{\text{best}} \leftarrow m$ — Select best 🟦 neighbor that proposed to node n
10: $\quad\quad\quad w_{\max} = \boldsymbol{w}[e]$
11: $\quad\quad$ **end if**
12: $\quad\quad$ **if** $\boldsymbol{\pi}[m] \in \{$🟦, 🟧$\}$ **then** `dead` $\leftarrow$ `false` **end if**
13: $\quad$ **end for**
14: $\quad$ **if** $m_{\text{best}} \neq \emptyset$ **then**
15: $\quad\quad \boldsymbol{\pi}[n] = \min(n, m_{\text{best}})$
16: $\quad\quad \boldsymbol{\pi}[m_{\text{best}}] = \min(n, m_{\text{best}})$
17: $\quad$ **end if**
18: $\quad$ **if** `dead` **then** $\boldsymbol{\pi}[n] = n$ **end if** — All neighbors are matched $\rightarrow$ singleton
19: **end if**

itself to a 🟦 neighbor that has previously proposed to this specific node, based on the highest edge weight among all proposals. After each iteration, all unmatched nodes will once again receive a 🟦 or 🟧 color at random, and the kernel finishes computation if it can no longer assign any colors. As such, this kernel can naturally parallelize greedy GRACLUS matching, in which conflicts in matching are resolved by only allowing matches between nodes of different color.

6.3.4 Heterogeneous Graph Learning

A large set of real-world datasets are stored as heterogeneous graphs (Section 2.1), motivating the introduction of specialized functionality for them in PyTorch Geometric. For example, most graphs in the area of recommendation, such as social graphs, are heterogeneous, as they store information about different types of entities and their different types of relations.

Heterogeneous graphs $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T}_V, \mathcal{T}_E)$ come with different types of information attached to nodes and edges. Thus, a single node or edge feature tensor can not hold all node or edge features of the whole graph, due to differences in type and dimensionality. Instead, a set of types Σ_V and Σ_E is specified for nodes and edges, respectively, each holding its own data tensors. Specifically, PyTorch Geometric identifies node types Σ_V via single strings and edge types Σ_E via a triplet `(source_node_type, edge_type, destination_node_type)` of strings: the edge type identifier and the two node types between which the edge type exists.

As a consequence of the different data structure, the message passing formulation changes accordingly, allowing the computation of MESSAGE and UPDATE functions conditioned on node and edge type (Schlichtkrull *et al.*, 2018). To let a GNN learn node and edge type dependent representations, one can utilize the *same* GNN operator $f_{\boldsymbol{\theta}}^{(\ell)}$ such that only its parametrization $\boldsymbol{\theta}$ changes according to type, *cf.* Section 3.2.4 and Equation (3.20):

$$\boldsymbol{h}_v^{(\ell)} = \sum_{r \in \Sigma_E} f_{\boldsymbol{\theta}_r}^{(\ell)}\big(\boldsymbol{h}_v^{(\ell-1)}, \{\!\{\boldsymbol{h}_w^{(\ell-1)} : w \in \mathcal{N}_r(v)\}\!\}\big). \tag{6.3}$$

Hence, heterogeneous graph learning partitions GNN execution into individual (but potentially parallelizable) message passing flows in *bipartite graphs*, followed by a destination node type-wise aggregation.

In contrast to the application of GNN layers in homogeneous graphs (Section 6.3.1), heterogeneous graph learning involves keeping track of individual feature dimensionalities and storing node-level/edge-level data for different types in temporary dictionaries. This makes heterogeneous GNN implementations messy and overly complicated. PyG offers two major features to circumvent the aforementioned problems:

- **Lazy initialization:** Since the number of input features and thus the size of tensors varies between different types, PyG provides lazy initialization functionality for *all* its supported GNN operators, in which model parameters are initialized only *after* the first call of forward propagation, dependent on the given shapes it has encountered. This allows us to avoid calculating and keeping track of all tensor sizes of the computation graph. Parameters are initialized lazily by passing `-1` to its expected input sizes.

- **Model transformation:** PyG provides the functionality to *automatically* convert any PyG homogeneous GNN model to a model that now expects a heterogeneous graph as input. Specifically, the process takes an existing GNN model and duplicates its MESSAGE and UPDATE functions to work on each edge type and node type individually, respectively. As such, recent advances in Graph Representation Learning can quickly be applied to heterogeneous graphs as well, *e.g.*, utilizing skip-connections, Jumping Knowledge or specialized normalization techniques.

```python
import torch
from torch_geometric.nn import SAGEConv, to_hetero

class GNN(torch.nn.Module):
    def __init__(self, hidden_channels, out_channels):
        super().__init__()
        # Bipartite message passing with lazy initialization:
        self.conv1 = SAGEConv((-1, -1), hidden_channels)
        self.conv2 = SAGEConv((-1, -1), out_channels)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index).relu()
        x = self.conv2(x, edge_index).relu()
        return x

model = GNN(hidden_channels=64, out_channels=dataset.num_classes)

# Automatic conversion to a heterogeneous GNN:
model = to_hetero(model, data.metadata(), aggr="sum")

# The heterogeneous model now expects dictionaries
# of node features and graph structure as input:
pred = model(data.x_dict, data.edge_index_dict)
```

Listing 4: Automatic conversion of a homogeneous PyG GNN model to a heterogeneous one via the `to_hetero` routine. Message passing layers are expected to perform message passing in bipartite graphs, and use lazy initialization to account for varying input feature dimensionalities across node types.

Internally, the process of model transformation makes use of the `torch.fx`[8] package, a toolkit for transforming PyTorch module instances, consisting of a symbolic tracer, an intermediate and customizable computation graph representation, and a Python code generator. Our heterogeneous model transformation routine then takes in a homogeneous GNN model and a metadata description (Σ_V, Σ_E) of the heterogeneous graph, traces its original computation, and transforms it such that it now expects heterogeneous graphs in the form of node type and edge type dictionaries as input, *cf.* Listing 4. Figure 6.6 further illustrates an example of an original homogeneous GNN computation graph and its newly obtained heterogeneous computation graph after transformation. Overall, model transformation involves the following steps:

1. It first tracks for each module and operator in the computation graph whether it performs node-parallel or edge-parallel computations.

2. It transforms and duplicates calls to modules and operators such that they operate on single node type or edge type instances. Special treatment is necessary for calls to `MessagePassing` modules, since they expect both node-level and edge-level data as input, and produce node-level outputs. Here, the inputs to a `MessagePassing` module are modified to take a tuple of source node and

[8] `torch.fx`: `https://pytorch.org/docs/stable/fx.html` (last access: August 25, 2022)

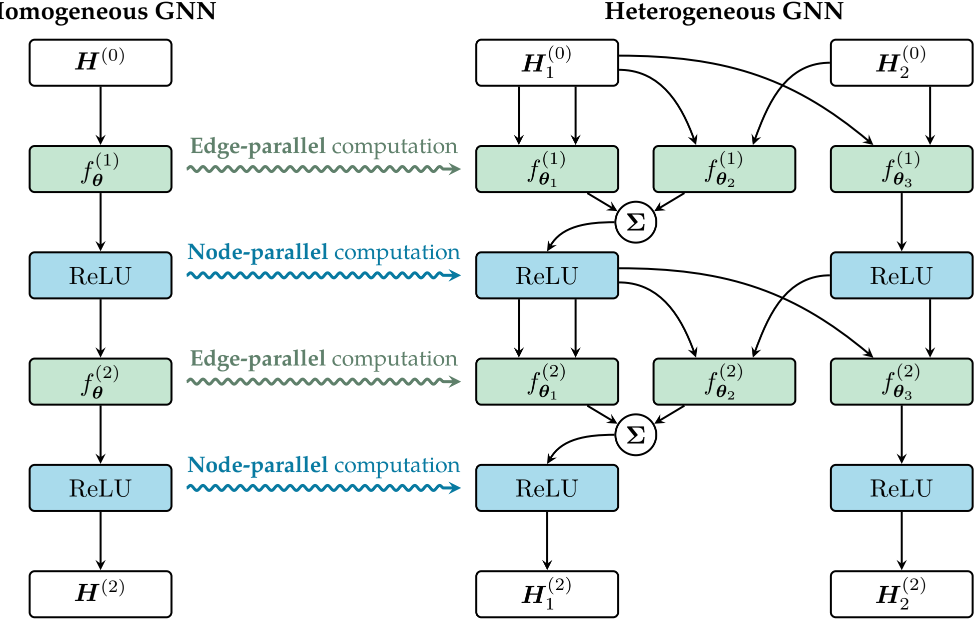


Figure 6.6: Comparison between a computation graph of a homogeneous GNN and its heterogeneous computation graph after model transformation, expecting two node types and three edge types as input. All modules and operators are first categorized into node-parallel or edge-parallel computation blocks, and replicated for each node type and edge type, respectively. For `MessagePassing` modules, computation is modified to perform bipartite message passing via disjunct source and destination node feature representations. Afterwards, intermediate outputs pointing to the same destination node type are aggregated.

destination node features for node-level data as input, and are called for each edge type. Afterwards, an aggregation is performed to transform edge type-wise outputs to node-level ones based on destination node type.

3. It erases all unused nodes in the computation graph.

4. It iterates over each children module and duplicates it for each node type or edge type, dependent on whether the given module performs node type-parallel or edge-parallel computation. Parameters for each new module are reset.

Similar functionality is available for creating heterogeneous Graph Neural Networks that utilize basis decomposition, *cf.* Section 3.2.4. Here, outputs of Message are re-weighted based on learnable basis and relation-specific parameters, implemented via a Message post-hook in the `MessagePassing` interface. Notably, with this scheme, any PyG GNN model can be converted into a heterogeneous graph learning scenario, independent of its original complexity.

6.3.5 Additional Features

Besides its general support for efficient and flexible homogeneous and heterogeneous graph learning, PyG provides further features to ease and enhance the training and inference stages of a general GNN pipeline:

- **TorchScript support:** *TorchScript* is a way to create serializable and optimizable models from PyTorch code, *i.e.* any TorchScript program can be saved from a Python process and loaded in a process where there is no Python dependency, such as in a standalone C++ program (Paszke *et al.*, 2019). This makes it possible to train models in an experimentation-centric setting directly in Python, while exporting trained models to a production environment where Python programs may be disadvantageous, *e.g.*, for performance and multi-threading reasons. TorchScript achieves this by statically analyzing a given Python module, and converting it into an intermediate representation in which it can be further optimized (*e.g.*, via kernel fusion) and deployed at scale, without the necessity of switching frameworks and without the risk of model divergence.

  Since PyG follows a Pythonic and experimentation-centric interface to allow for the definition of almost any GNN operator, it exploits a number of features that are not available in a regular TorchScript program. As such, PyG achieves TorchScript serialization by dynamically re-building user-specified `MessagePassing` implementations, in which it will statically infer all variable types of MESSAGE and UPDATE functions *ahead* of time, and turn it into a specialized implementation that makes use of those definitions by default instead of inferring them during execution only.

  Specifically, we used PyG's TorchScript support to deploy GNN models at scale at CERN for the task of high energy particle reconstruction (presented at the PyTorch Developer Day 2020[9]). Here, PyG's TorchScript integration avoids system inefficencies and helps to keep up with the compute demand of processing exabytes of highly structured data per year, with thousands of particles per event to identify.

- **GraphGym manager:** PyG integrates the *GraphGym* manager (You *et al.*, 2020), a platform for designing and evaluating GNNs with ease. Specifically, GraphGym lets users reproduce GNN experiments, is able to launch and analyze thousands of different GNN configurations, and is customizable by registering new modules to a GNN learning pipeline. It provides a highly modularized pipeline for data loading and splitting routines, GNN implementations, tasks (node-level, edge-level or graph-level) and evaluation protocols, fully described by a configuration file. On top of it, it provides a scalable experiment management, *i.e.* to launch thousands of GNN experiments in parallel and to auto-generate experiment analyses across random seeds and different hyperparameter settings.

- **PyTorch ecosystem integration:** The framework specificity of PyG allows it to fit nicely into the PyTorch ecosystem[10]. For example, PyG provides out-of-the-box integration with PyTorch Lightning (Falcon *et al.*, 2019) for distributed training or mixed-precision training. In particular, the seamless combination of both

[9] Lindsey Gray and Matthias Fey: *Graph Convolutional Operators in the PyTorch JIT*: `https://www.youtube.com/watch?v=4swsv0LzL_A` (last access: August 25, 2022)

[10] PyTorch ecosystem: `https://pytorch.org/ecosystem` (latest access: August 25, 2022)

libraries heavily simplifies the tasks of scaling GNN models up, *i.e.* only the number of GPUs and the accelerator type have to be defined to go from single-GPU to multi-GPU training. Furthermore, PyG provides seamless integration with Captum (Kokhlikyan *et al.*, 2019), a library for interpreting and explaining PyTorch models, to expand explainability analysis to GNN models as well.

- **Inference optimizations:** Furthermore, PyG has started to adopt specialized GNN inference optimization strategies, *e.g.*, via a feature decomposition strategy (Zhou *et al.*, 2021). Here, message and aggregation computation is performed on column-major feature chunks by decomposing the dimension of feature vectors. This scheme leads to improved data re-use of feature vectors through better cache statistics while not harming model performance of common GNNs.

Utilizing these advanced features can greatly accelerate training and inference stages of a GNN pipeline, both for academic and industry use-cases.

6.4 PyGAS: Auto-Scaling Graph Neural Networks

The *PyTorch Geometric AutoScale* (*PyGAS*) framework denotes the practical realization of our GNNAutoScale (GAS) approach and allows to *auto-scale* any GNN model to large-scale graphs while maintaining the properties and power of the original GNN (Fey *et al.*, 2021), *i.e.* it can naturally operate on all edges included in the local neighborhood around each node without the necessity of sub-sampling or non-trainable propagations, *cf.* Section 5.3. PyGAS condenses GAS and its theoretical findings into a unified tool, making it easy to convert common and custom GNN models into their scalable variants while requiring orders of magnitude less GPU memory. It is build upon PyTorch (Paszke *et al.*, 2019) and the PyTorch Geometric library (Fey & Lenssen, 2019), and therefore immediately inherits the application of recent advancements in Graph Representation Learning and makes them applicable for large-scale graph learning, *cf.* Section 6.3.

PyGAS scales the training and inference stages of a GNN by pruning its original computation graph, and substituting out-of-mini-batch information from a detached history storage located in CPU memory. Overall, PyGAS provides an easy-to-use interface to convert common and custom GNN models from PyTorch Geometric into their scalable variants. In addition, it provides a fully deterministic test bed for evaluating custom GNN models on larger graphs. PyGAS and its complete benchmark suite are fully open-sourced on GitHub[11].

To highlight its ease-of-use, we showcase the necessary changes to convert a common GCN architecture (Kipf & Welling, 2017) implemented in PyG to its corresponding scalable version, *cf.* Figure 6.7. In particular, the GCN model now inherits from `ScalableGNN`, which takes care of creating all historical embeddings of all layers (accessible via `self.histories`) and provides an efficient history access pattern via the `push_and_pull()` routine. The additional `n_id` input argument is used to map the local node indices of a mini-batch to the global node indices, which allows for pushing and pulling *local* node embeddings to and from the *global* history storage. Notably, model

[11] PyGAS: `https://github.com/rusty1s/pyg_autoscale` (last access: August 25, 2022)

```python
class GCN(Module):
  def __init__(self, dim, num_layers):
    super().__init__()

    self.convs = ModuleList()
    for _ in range(num_layers):
      conv = GCNConv(dim, dim)
      self.convs.append(conv)

  def forward(self, x, edge_index):
    for conv in self.convs[:-1]:
      x = conv(x, edge_index).relu()


    return self.convs[-1](x, edge_index)
```

(**a**) PyG implementation

```python
class GCN(ScalableGNN):
  def __init__(self, num_nodes, dim, num_layers):
    super().__init__(num_nodes, dim, num_layers)

    self.convs = ModuleList()
    for _ in range(num_layers):
      conv = GCNConv(dim, dim)
      self.convs.append(conv)

  def forward(self, x, edge_index, n_id):
    for i, conv, in enumerate(self.convs[:-1]):
      x = conv(x, edge_index).relu()
      history = self.histories[i]
      x = self.push_and_pull(history, x, n_id)
    return self.convs[-1](x, edge_index)
```

(**b**) PyGAS implementation

Figure 6.7: Comparison between a (a) PyG full-batch GCN and (b) a PyGAS mini-batch GCN implementation. Only minimal changes are required to auto-scale GCN (or any other GNN model) to large graphs (Fey *et al.*, 2021).

transformation into `ScalableGNN` variants and support for heterogeneous graphs is applicable as well, but is left for future work.

Our GAS framework accesses histories frequently to account for any data outside of the current mini-batch, *i.e.* in every GNN layer. As such, naively reading from and writing to histories can quickly cause a major I/O bottleneck, as actual GPU work has to wait until memory transfers from and to the GPU are complete, leading to the GPU being idle most of the time. PyGAS identifies two major causes of this phenomenon: (**1**) the serial execution of memory transfers and GNN computation, and (**2**) the frequent random accesses of node features in the global CPU history storage.

PyGAS resolves the first issue by optimizing pulling from and pushing to histories via *non-blocking* device transfers, *cf.* Figure 6.8. Specifically, it immediately starts pulling historical embeddings for each layer asynchronously at the beginning of each optimization step, which ensures that GPUs do not run idle while waiting for memory transfers to complete. In particular, a separate worker thread gathers historical information into one of multiple pinned CPU memory buffers (denoted by Pull), from where it can be transfered to the GPU via the usage of CUDA streams without blocking any CPU or GPU execution. Synchronization is then performed by synchronizing the respective CUDA stream just before inputting the transferred data into the respective GNN layer. The same strategy is applied for pushing information to the history. Considering that the device transfer of a set of historical features $\bar{\boldsymbol{H}}^{(\ell-1)}$ is faster than the execution of the GNN layer $f_{\boldsymbol{\theta}}^{(\ell)}$, this scheme does not lead to *any* runtime overhead when leveraging historical embeddings and can approximately be twice as fast as its serial non-overlapping counterpart. Notably, PyGAS implements this non-blocking transfer scheme with custom C++/CUDA code to avoid Python's global interpreter lock (Fey *et al.*, 2021).

In order to minimize the inter-connectivity between mini-batches, our GAS framework makes use of graph clustering techniques, *e.g.*, Metis (Karypis & Kumar, 1998; Dhillon *et al.*, 2007), to partition the graph into clusters in a pre-processing step. Notably, this technique does not only reduce the number of out-of-mini-batch nodes

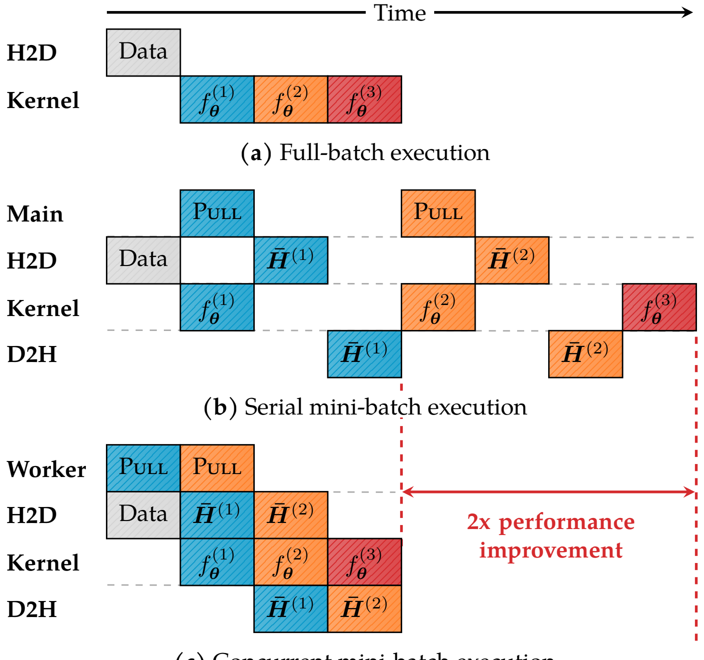


Figure 6.8: Illustrative runtime performances of a serial and concurrent mini-batch execution in comparison to a full-batch GNN execution. In the full-batch approach (**a**), all necessary data is first transferred to the device via the Host2Device (H2D) engine, before GNN layers are executed in serial inside the kernel engine. As depicted in (**b**), a serial mini-batch execution suffers from an I/O bottleneck, in particular because each kernel engine has to wait for memory transfers to complete. The concurrent mini-batch execution (**c**) avoids this problem by leveraging an additional worker thread and overlapping data transfers, leading to two times performance improvements in comparison to a serial execution, which is on par with the standard full-batch approach (Fey *et al.*, 2021).

(which reduces memory transfer costs in return), but also guarantees that nodes of the same mini-batch are hold in a contiguous range in the CPU history storage. As a result, pushing newly estimated information of mini-batch nodes now leads to coalesced memory transfers, which significantly speeds up updates of historical embeddings and mitigates the second issue to some extend.

6.5 Evaluation

In this section, we perform an extensive benchmark analysis across a wide range of GNN operators and datasets to showcase the generality and reproducibility of the PyG framework (Section 6.5.1). We further evaluate the efficiency of GNN implementations, both for varying sparse tensor storage formats provided by PyG, as well as regarding mini-batch processing within our PyGAS extension (Section 6.5.2).

Method	Cora		CiteSeer		PubMed	
	Fixed	Random	Fixed	Random	Fixed	Random
Cheby	81.4±0.7	77.8±2.2	70.2±1.0	67.7±1.7	78.4±0.4	75.8±2.2
GCN	81.5±0.6	79.4±1.9	71.1±0.7	68.1±1.7	79.0±0.6	77.4±2.4
GAT	83.1±0.4	81.0±1.4	70.8±0.5	69.2±1.9	79.0±0.3	77.5±2.3
SGC	81.7±0.1	80.2±1.6	71.3±0.2	68.7±1.6	78.9±0.1	76.5±2.4
ARMA	82.8±0.6	80.7±1.4	**72.3**±1.1	68.9±1.6	78.8±0.3	77.7±2.6
APPNP	**83.3**±0.5	**82.2**±1.5	71.8±0.5	**70.0**±1.4	**80.1**±0.2	**79.4**±2.2

Table 6.2: Mean accuracy and standard deviations of node classification within PyG for both fixed and random splits (Fey & Lenssen, 2019)**.**

6.5.1 A Uniform GNN Benchmark Analysis

We evaluate the correctness of a subset of the implemented methods in PyG by performing a comprehensive comparative study in uniform evaluation scenarios (Fey & Lenssen, 2019). For all experiments, we tried to follow the hyperparameter setup of the respective papers as closely as possible. The individual experimental setups can be derived and all experiments can be replicated from the code provided at our GitHub repository.[12] Notably, new GNNs can be easily plugged into the benchmark, allowing for a fair comparison across a wide range of common benchmark datasets.

6.5.1.1 Semi-supervised Node Classification. We start by performing semi-supervised node classification (*cf.* Table 6.2) on the small-scale citation graphs Cora, CiteSeer and PubMed (Sen *et al.*, 2008; Yang *et al.*, 2016) by reporting average accuracies of **(a)** 100 runs for the fixed training/validation/test split from Kipf & Welling (2017), and **(b)** 100 runs of randomly initialized training/validation/test splits as suggested by Shchur *et al.* (2018), in which we additionally ensure uniform class distribution on the training split. We evaluate on six GNN models that significantly advanced the field of Graph Representation Learning: Cheby (Defferrard *et al.*, 2016), GCN (Kipf & Welling, 2017), GAT (Veličković *et al.*, 2018), SGC (Wu *et al.*, 2019a), ARMA (Bianchi *et al.*, 2019) and APPNP (Klicpera *et al.*, 2019a). Nearly all experiments show a high reproducibility of the results reported in the respective papers. However, test performance is worse for all models when using random data splits. Among all experiments, the APPNP operator generally performs best, while the ARMA, SGC, GCN and GAT operators follow closely behind (Fey & Lenssen, 2019).

6.5.1.2 Graph Classification. Next, we report the average accuracy of 10-fold cross validation for the task of graph classification on a number of common benchmark datasets (Morris *et al.*, 2020a) (*cf.* Table 6.3), in which we randomly sample a training fold to serve as a validation set. This in contrast to the evaluation procedure of related works, which only report validation performance. However, this leads to overly optimistic performances of model predictions and does not represent a fair and realistic evaluation scenario. We evaluate the performances of **(a)** flat GNN architectures (via GCN (Kipf & Welling, 2017), SAGE (Hamilton *et al.*, 2017), GIN-0 and GIN-ϵ (Xu *et al.*,

[12]PyG benchmark: `https://github.com/rusty1s/pytorch_geometric/tree/master/benchmark` (last access: August 25, 2022)

	Method	MUTAG	PROTEINS	COLLAB	IMDB-BINARY	REDDIT-BINARY
Flat	GCN	74.6±7.7	73.1±3.8	**80.6**±2.1	72.6±4.5	89.3±3.3
	SAGE	74.9±8.7	**73.8**±3.6	79.7±1.7	72.4±3.6	89.1±1.9
	GIN-0	**85.7**±7.7	72.1±5.1	79.3±2.7	**72.8**±4.5	89.6±2.6
	GIN-ϵ	83.4±7.5	72.6±4.9	79.8±2.4	72.1±5.1	**90.3**±3.0
Hier.	Graclus	77.1±7.2	73.0±4.1	79.6±2.0	72.2±4.2	88.8±3.2
	top$_k$	76.3±7.5	72.7±4.1	**79.7**±2.2	72.5±4.6	87.6±2.4
	DiffPool	**85.0**±10.3	**75.1**±3.5	78.9±2.3	**72.6**±3.9	**92.1**±2.6
Global	SAGE w/o JK	73.7±7.8	72.7±3.6	79.6±2.4	72.1±4.4	87.9±1.9
	GlobalAttention	74.6±8.0	72.5±4.5	**79.6**±2.2	72.3±3.8	87.4±2.5
	Set2Set	73.7±6.9	**73.6**±3.7	79.6±2.3	72.2±4.2	**89.6**±2.4
	SortPool	**77.3**±8.9	72.4±4.1	77.7±3.1	**72.4**±3.8	74.9±6.7

Table 6.3: Accuracy of graph classification within PyG (Fey & Lenssen, 2019)**.**

2019c)), **(b)** hierarchical GNN architectures (via Graclus (Dhillon *et al.*, 2007), top$_k$ (Cangea *et al.*, 2018) and DiffPool (Ying *et al.*, 2018b)), as well as the performance of **(c)** more sophisticated global readout strategies (via global attention (Li *et al.*, 2016b), Set2Set (Vinyals *et al.*, 2016) and SortPool (Zhang *et al.*, 2018)). For all datasets, we only make use of discrete node features. In case they are not given, we use one-hot encodings of node degrees as feature input. For all experiments on flat GNN architectures, we use the global mean operator to obtain graph-level outputs. Inspired by the Jumping Knowledge framework (Xu *et al.*, 2018), we compute graph-level outputs after each convolutional layer and combine them via concatenation. For evaluating the (global) pooling operators, we use the GraphSAGE operator as our baseline. We omit Jumping Knowledge when comparing global pooling operators, and hence report an additional baseline based on global mean pooling without JK (SAGE w/o JK). For each dataset, we tune **(a)** the number of hidden units $\in \{16, 32, 64, 128\}$ and **(2)** the number of layers $\in \{2, 3, 4, 5\}$ with respect to the validation set (Fey & Lenssen, 2019).

Due to standardized evaluations and network architectures, not all results are aligned with their official reported values. For example, except for DiffPool (Ying *et al.*, 2018b), (global) pooling operators do not perform as beneficially as expected to their respective (flat) counterparts, especially when baselines are enhanced by Jumping Knowledge. This was further verified in an independent subsequent study by Mesquita *et al.* (2020). However, the potential of more sophisticated approaches may not be well-reflected on these simple benchmark tasks (Cai & Wang, 2018). Among the flat GNN approaches, the GIN layer generally achieves the best results, in particular due to its expressivity in reasoning about graph structures, *cf.* Section 3.4.

6.5.1.3 Point Cloud Classification. We further evaluate various point cloud methods on the ModelNet10 dataset (Wu *et al.*, 2015) where we uniformly sample 1,024 points from mesh surfaces based on face area (*cf.* Table 6.4). Specifically, we evaluate MPNN (Simonovsky & Komodakis, 2017), PointNet++ (Qi *et al.*, 2017b), EdgeCNN (Wang *et al.*, 2019e), SplineCNN (Fey *et al.*, 2018) and PointCNN (Li *et al.*, 2018b). As hierarchical pooling layers, we use the iterative farthest point sampling algorithm

Method	**ModelNet10**
MPNN (Simonovsky & Komodakis, 2017)	92.07
PointNet++ (Qi *et al.*, 2017b)	92.51
EdgeCNN (Wang *et al.*, 2019e)	92.62
SplineCNN (Fey *et al.*, 2018)	92.65
PointCNN (Li *et al.*, 2018b)	**93.28**

Table 6.4: Accuracy of point cloud classification within PyG (Fey & Lenssen, 2019)**.**

Dataset	**Method**	**DGL v0.2** Degree Bucketing	**DGL v0.3** `gather&scatter`	**PyG**
Cora	GCN	4.19	0.32	**0.25**
	GAT	6.31	5.36	**0.80**
CiteSeer	GCN	3.78	0.34	**0.30**
	GAT	5.61	4.91	**0.88**
PubMed	GCN	12.91	0.36	**0.32**
	GAT	18.69	13.76	**2.42**
MUTAG	R-GCN	18.81	2.40	**2.14**

Table 6.5: Training runtimes in seconds on small-scale benchmark datasets. The `gather&scatter` approach of PyG outperforms the degree bucketing approach of DGL by a wide margin, and is slightly faster given similar implementations due to the higher Python overhead costs in DGL's wrapper codes (Fey & Lenssen, 2019).

followed by a new graph generation procedure based on a larger query ball (PointNet++, MPNN and SplineCNN) or based on a fixed number of k-nearest neighbors (EdgeCNN and PointCNN). We took care of using approximately the same number of parameters for each model. Notably, all approaches perform nearly identically with PointCNN taking a slight lead. We attribute this to the fact that all operators are based on similar principles and might have the same expressive power for the given task (Fey & Lenssen, 2019).

6.5.2 Efficiency of GNN Design

We now evaluate the efficiency of GNN implementations across different models and datasets, as well as for varying sparse tensor storage formats provided by PyG. Furthermore, we evaluate the efficiency of mini-batch processing within our PyGAS extension.

6.5.2.1 Training runtimes. We conduct several experiments on a number of dataset-model pairs to report the runtime of a whole training procedure for 200 epochs obtained on a single NVIDIA GTX 1080 Ti (*cf.* Table 6.5). In particular, we evaluate the efficiency of GCN (Kipf & Welling, 2017), GAT (Veličković *et al.*, 2018) and a relational GCN variant (R-GCN) (Schlichtkrull *et al.*, 2018). As it shows, PyG is very fast despite

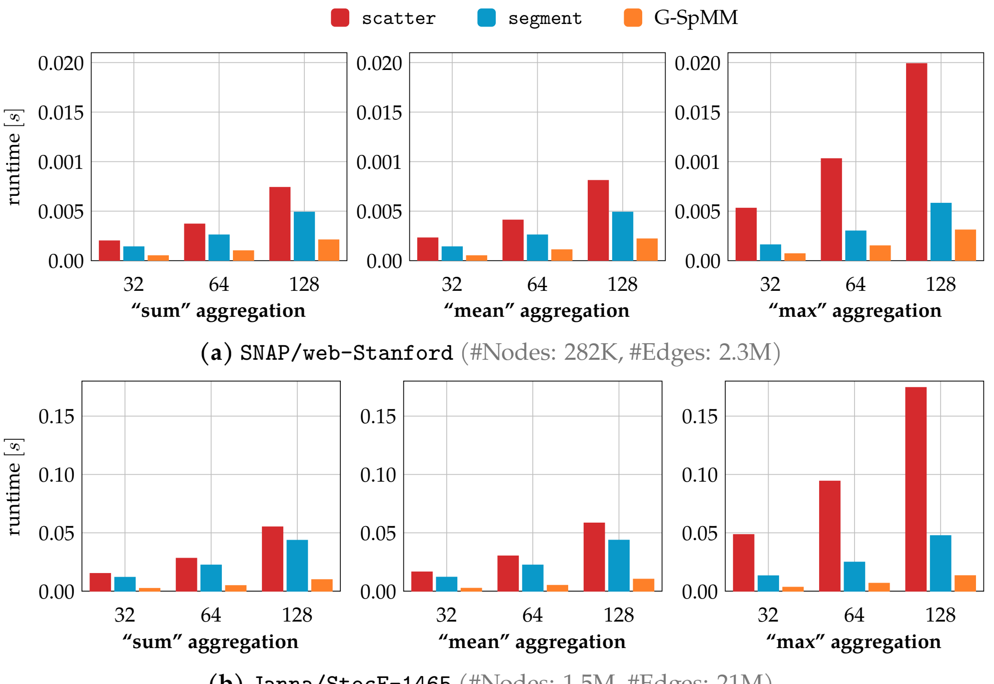
working on sparse data. Compared to the *degree bucketing* approach of DGL (Wang *et al.*, 2019b), PyG trains models up to 40 times faster. The degree bucketing approach of DGL decomposes the neighbor aggregation phase of GNNs into buckets of same node degree. This makes the aggregation procedure $\bigoplus$ of GNNs fully customizable due to its dense nature, *e.g.*, it can easily utilize Long Short-Term Memorys (LSTMs) as aggregators, but has the disadvantage that it needs to process buckets of different node degree sequentially rather than in parallel. In practice, we found that the generality of degree bucketing does not outweigh the cost of non-parallel processing. Although runtimes are much more comparable in later releases of DGL that inherit the `gather&scatter` approach of PyG, PyG is still slightly faster due to the higher Python overhead costs in DGL's wrapper codes (Wang *et al.*, 2019b; Fey & Lenssen, 2019). Additional speed ups could be achieved for GAT by providing our own optimized sparse softmax CUDA kernel, which further improve runtimes by up to 7 times. An independent study in Wu *et al.* (2021) has confirmed the slightly superior runtime statistics of PyG on graph-level tasks as well.

6.5.2.2 Sparse Storage Format Analysis. We now evaluate the efficiency of message aggregation using the `scatter`, `segment` or G-SpMM algorithms on two different datasets stemming from the SuiteSparse Matrix Collection (Kolodziej *et al.*, 2019), namely `SNAP/web-Stanford` and `Janna/StocF-1465`. The `SNAP/web-Stanford` graph is a web network with around 282K nodes and an average node degree of ≈ 8.2. The `Janna/StocF-1465` graph is obtained from a fluid-dynamical problem, and contains around 1.5M nodes with an average node degree of ≈ 14.3. For `scatter` and `segment` algorithms, we benchmark both the gathering of node features into edge-parallel space and performing the final reduction, while our G-SpMM algorithm fuses both gathering and aggregation into a single step. Evaluation is performed by varying the sizes $\in \{32, 64, 128\}$ of node features and selecting different aggregation operators $\bigoplus \in \{\texttt{sum}, \texttt{mean}, \texttt{max}\}$. Runtimes are obtained on a single NVIDIA TITAN RTX with 24GB of GPU memory, *cf.* Figure 6.9.

Notably, the G-SpMM algorithm outperforms both `scatter` and `segment` algorithms by a wide margin on all combinations of feature sizes, aggregation procedures and datasets. In addition, the `segment` algorithm is consistently faster than `scatter` as well, which can be explained by the fact that destination nodes indices are grouped contiguously in memory, and as a result, reduction can be performed inside each thread rather than outside of it (via atomic operations). The difference in observed runtimes between `segment` and G-SpMM denote the overhead in creating intermediate edge-level tensors, which is not necessary for g-SpMM. As a result, fusing message and aggregation computation proves to be highly beneficial in practice, and can lead to huge efficiency improvements whenever the underlying GNN allows to do so.

Furthermore, we observe large differences in efficiency for `scatter` across different reduction operators, in particular when using "max" aggregation. This can be explained by the fact that `atomic_add` is available as a dedicated function on most modern GPUs with reasonably high compute capability, while `atomic_max` is self-implemented via the usage of `atomic_cas` (compare and swap).[13]

[13] `https://docs.nvidia.com/cuda/cuda-c-programming-guide` (last access: August 25, 2022)



(a) SNAP/web-Stanford (#Nodes: 282K, #Edges: 2.3M)

(b) Janna/StocF-1465 (#Nodes: 1.5M, #Edges: 21M)

Figure 6.9: GPU runtime comparison of different message aggregation algorithms, namely `scatter`, `segment` and G-SpMM. Benchmarking is performed for various feature dimensionalities, reduction operators, and datasets. Notably, fused message aggregation via G-SpMM outperforms both `scatter` and `segment` by a wide margin. `scatter_max` performs the worst due to lack of dedicated hardware support in its atomic operation.

6.5.2.3 Mini-batch Efficiency within PyGAS. We now analyze how our PyGAS extension enables large-scale GNN training due to fast mini-batch execution (Fey *et al.*, 2021). Specifically, we are interested in how our concurrent memory transfer scheme (*cf.* Section 6.4) reduces the overhead induced by accessing historical embeddings from the global CPU storage. For this, we evaluate runtimes of a 4-layer GIN model on synthetic graph data, which allows fine-grained control over the ratio between inter- and intra-connected nodes, *cf.* Figure 6.10. Here, a given mini-batch consists of exactly 4,000 nodes which are randomly intra-connected to 60 other nodes. We vary the number of inter-connections (connections to nodes outside of the batch) by adding out-of-batch nodes that are randomly inter-connected to 60 nodes inside the batch. Notably, the naive serial memory transfer increases runtimes up to 350%, which indicates that frequent history accesses can cause major I/O bottlenecks. In contrast, our concurrent access pattern incurs *almost no* I/O overhead *at all*, and the overhead in execution time is solely explained by the computational overhead of aggregating far more messages during message propagation. Note that in most real-world scenarios, the additional aggregation of history data may only increase runtimes up to 25%, since most real-world datasets contain inter-/intra-connectivity ratios between 0.1 and 2.5 (Fey *et al.*, 2021). Further, the additional overhead of computing Metis partitions in the pre-processing stage is negligible and is quickly mitigated by faster training times:

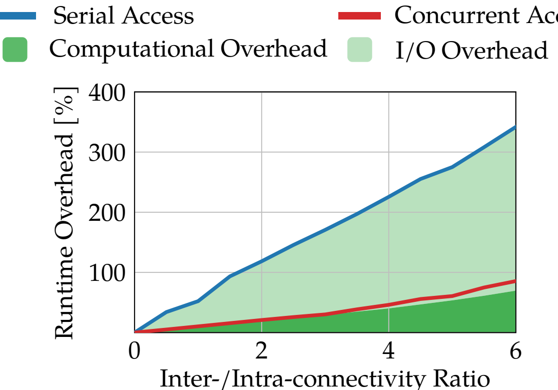


Figure 6.10: The runtime overhead of PyGAS in relation to the inter-/intra-connectivity ratio of mini-batches, both for serial and concurrent history access patterns. The overall runtime overhead is further separated into computational overhead (overhead of aggregating additional messages) and I/O overhead (overhead of pulling from and pushing to histories). The concurrent memory transfer scheme of PyGAS reduces I/O overhead caused by histories by a wide margin (Fey *et al.*, 2021).

Dataset	**Runtime [s]**		**Memory [MB]**	
	GTTF	PyGAS	GTTF	PyGAS
Cora	0.077	**0.006**	18.01	**2.13**
PubMed	0.071	**0.006**	28.79	**2.19**
PPI	0.976	**0.007**	134.86	**12.37**
Flickr	1.178	**0.007**	325.97	**16.32**

Table 6.6: Runtime and memory consumption for processing a mini-batch in a 4-layer GCN model with GTTF and PyGAS. PyGAS is both faster and consumes less memory than GTTF (Fey *et al.*, 2021).

Computing the partitioning of a graph with 2M nodes takes only about 20–50 seconds (depending on the number of clusters) (Fey *et al.*, 2021).

Next, we compare runtimes and memory consumption of PyGAS to the recent GTTF proposal (Markowitz *et al.*, 2021), which utilizes a fast neighbor sampling strategy based on tensor functionals. For this, we make use of a 4-layer GCN model with equal mini-batch and receptive field sizes. As shown in Table 6.6, PyGAS is both faster and consumes less memory than GTTF. Although GTTF utilizes a fast vectorized sampling procedure, its underlying recursive neighborhood construction still scales *exponentially* with GNN depth, which explains the observable differences in runtime and memory consumption (Fey *et al.*, 2021).

7

Conclusion and Future Work

This thesis proposed approaches for applying deep learning techniques to graph-structured data based on the recently emerging framework of neural message passing, and investigated their applicability both from a theoretical and practical point of view. Overall, we introduced solutions by generalizing concepts of traditional neural building blocks as well as by aiming to overcome existing shortcomings and inherent weaknesses of traditional message passing Graph Neural Networks. Furthermore, we looked into ways to eliminate scalability issues inherent to message passing, and proposed effective solutions regarding their efficient and flexible realization.

Importantly, Graph Neural Networks (GNNs) play a crucial role both for the future of graph machine learning in particular and for the future of AI in general. For example, they advance graph machine learning by no longer being forced to rely on hand-crafted features for graph-based predictive tasks. Instead, GNNs can naturally integrate rich feature information attached to nodes and edges for reasoning about structural graph properties as part of an end-to-end representation learning pipeline. They also advance the state of AI in general, as GNNs provide a general and broadly applicable framework for reasoning about the structures and compositions in a highly inter-connected world. Importantly, deep learning models no longer need to take fixed computation graphs for granted, but can actually define computation dynamically as part of the input, given in the form of a graph. On one hand, this allows us to view GNNs as a general and much more broadly applicable class of deep learning models that elegantly group and generalize concepts on fixed-sized domains, *e.g.*, Convolutional Neural Networks, Recurrent Neural Networks or Transformers. On the other hand, GNNs have the capability to act as a router between specialized models trained on individual tasks in isolation, *i.e.* they are inherently able to route and propagate low-level information in order to make higher-level decisions — a key concept for achieving a much more general and powerful kind of artificial intelligence.

In what follows, we summarize our contributions towards these goals in Section 7.1 and outline interesting directions for future work in Section 7.2.

7.1 Conclusion

This thesis was structured into four major components, tackling upon the relationship, methodological development, scalability and realization aspects of the overarching topic of this thesis — Graph Neural Networks via neural message passing.

We introduced and studied the central blueprint of GNNs in the form of a trainable and differentiable message passing scheme. Contributions were made in relating the general concepts of GNNs to well-known deep learning techniques and graph isomorphism heuristics. In particular, we have introduced the Spline-Based Convolutional Neural Networks (SplineCNNs), a novel GNN architecture based on an anisotropic MESSAGE formulation via a continuous B-spline kernel definition, which allow for a fast execution due to the local support property of B-spline basis functions. Importantly, SplineCNNs resemble the traditional definition of Convolutional Neural Networks for discrete data, while directly being applicable on more diverse and general domains as well, *e.g.*, for learning on either simple or embedded graphs. As such, we showed that the concepts and properties of traditional neural network building blocks can be successfully transferred applied to graph-structured data as well (answering **Research Question 1**). In particular, we evaluated SplineCNNs on the task of node classification, superpixels graph classification and shape correspondence, in which it reached state-of-the-art performance on all of these.

Furthermore, we have related the expressive power of Graph Neural Networks to the Weisfeiler-Lehman (WL) graph isomorphism heuristic, and showed their equivalent power in reasoning about and distinguishing non-isomorphic (sub-)graphs. Based on our findings, we proposed a generalization of GNNs named k-GNNs, which can take higher-order graph structures at multiple scales into account, leading to provably more powerful Graph Neural Networks. Empirically we showed that k-GNNs provide a general blueprint that consistently strengthen the performance of GNNs on graph-level tasks. In addition, we identified additional shortcomings and inherent weaknesses of general message passing GNNs, and proposed several novel methodological advancements w.r.t. GNN design (answering **Research Question 2**). Contributions to these problems were three-fold, depending on the given task and domain:

- Specifically, we introduced the Dynamic Neighborhood Aggregation (DNA) procedure, a novel and specialized neighborhood aggregation scheme to overcome the over-smoothing problem in deep GNNs. As a result, DNA is able to capture localized representations stemming from long-range dependencies. Our solution utilizes a selective and node-adaptive aggregation formulation of neighbors of potentially differing locality, guided by attention. In contrast to related approaches, DNA allows a GNN to control its own spread-out, possibly aggregating more global information in one branch, and falling back to more local information in others. Overall, integrating DNA into GNN model design is generally able to boost model performance, especially in heterophily graphs.
- Furthermore, we introduced the Hierarchical Inter-Message Passing (HIMP) architecture to allow for more expressive GNNs in the task of molecular graph learning. HIMP is able to exchange information between different hierarchies and higher-order sub-structures of molecules, *e.g.*, between rings or bonds. In particular, we argued that related graph coarsening strategies are not sufficient

to accurately capture meaningful cluster assignments due to their potentially higher-order nature. As such, HIMP can naturally overcome the restrictions of traditional GNNs, strengthening their performance with minimal computational overhead in return. In our evaluation, we showed that HIMP can improve traditional GNN model performance up to approximately 60%.

- Lastly, significant contributions were made to advance the state-of-the-art in data-driven deep graph matching algorithms. Here, our proposed Deep Graph Matching Consensus (DGMC) framework utilizes a two-stage neural architecture, whose second stage is able to resolve the ambiguities and adjust any false matchings made in the first stage, induced by the locality of message passing. For this, DGMC aims to reach a data-driven neighborhood consensus in local neighborhoods by injecting global node colorings to the message passing formulation in a purely local and sparsity-aware fashion. As we have shown, such a refinement of initial correspondences is crucial to tackle the task of deep graph matching with high precision, leading to significant improvements across a wide range of tasks and GNN instantiations.

Furthermore, we proposed methodological advancements regarding the scalability aspect of GNNs, motivated by the shortcomings of related scalability approaches that are either restricted to (**1**) shallow GNNs, (**2**) shallow subgraphs, or (**3**) provably less powerful GNN variants. Our GNNAutoScale (GAS) framework prunes entire sub-trees of the computation graph by utilizing historical node embeddings acquired from prior training iterations, leading to constant GPU memory consumption in respect to input node size. In particular, this allows for deeply stacked GNNs while accounting for all available neighborhood information in every layer. We have shown both theoretically and practically that GAS is able to maintain the expressive power of the original GNN (answering **Research Question 3**). Importantly, the GAS mini-batch training technique can be applied to *any* GNN backbone, ultimately allowing the application of deep and expressive GNNs on large-scale graphs. Empirically, we have shown the efficiency of our GAS approach and the practical benefits of evaluating larger models on larger scale. Furthermore, significant contributions were made to the availability of large-scale graph benchmark datasets and standardized GNN evaluation techniques, as condensed in the Open Graph Benchmark (OGB) suite. OGB overcomes the shortcomings of existing graph benchmark datasets (*e.g.*, small-scale, data leakage, non-standardized splitting strategies and evaluation protocols) by providing a rich set of diverse and realistic graph datasets, grouped into different task categories, application domains and scales. In particular, OGB aims to provide meaningful splitting schemes, *e.g.*, based on temporal information or scaffolds. Empirically, we have identified major challenges on all available datasets in an extensive benchmark analysis, using representative graph-based machine learning models that utilize a diverse range of GNN scalability techniques.

Finally, we introduced important concepts to efficiently realize GNNs in a unified and flexible way. Our main contribution towards this goal is the development of PyTorch Geometric (PyG), a well-known Graph Neural Network library, built upon PyTorch. PyG bundles the state-of-the-art in Graph Representation Learning in a unified and comprehensive package, and achieves flexibility in GNN design on both low levels, *e.g.*, via a general `MessagePassing` interface, and high levels, *e.g.*, by composing GNN models via pre-defined and ready-to-use layers and operators. As we have shown, PyG achieves high data throughput by leveraging sparse GPU acceleration, dedi-

cated CUDA kernels, and by introducing effective mini-batching techniques. Furthermore, it provides a general testbed for evaluating GNN variants in a fair and comparable fashion, achieving high reproducibility of officially reported model performances (answering **Research Question 4**). In addition, we introduced the PyGAS extension, which allows to convert common and custom GNN models from PyG into their scalable variants by following our GAS framework. Importantly, PyGAS is able to efficiently access and transfer historical embeddings from CPU storage via a novel asynchronous memory transfer strategy. We have shown that this technique heavily improves upon the runtimes of our GAS approach. In fact, we have been able to completely eliminate the overhead induced by frequent history accesses and device transfers, making GAS one the fastest scalability approaches for mini-batch processed GNNs up to this date.

7.2 Future Work

The area of designing novel architectures and inventing novel optimization strategies for graph-based machine learning is very broad, leaving rich potential for future research. This section will outline potential directions for most of the tools described in this thesis.

The theoretical understanding of the capabilities of GNNs has been mainly investigated from the perspective of classifying their expressive power in distinguishing certain (sub-)graph structures, but the mechanisms of the underlying learning process are still largely unexplored. In particular, the theoretical expressiveness of GNNs might actually not be the deciding factor for GNN performance, especially since the WL is already well able to distinguish all non-isomorphic graphs on most of the benchmark datasets available today (Morris *et al.*, 2021a). As such, arguing that provably more powerful GNN architectures will automatically result in better model performance need to be taken with a grain of salt. In complete contrast, there exist GNN variants that mostly ignore graph structure completely and only care about local feature propagation, which nonetheless perform well in practice. As such, it is desirable to understand the concrete interplay between model generalization capabilities, the underlying expressive power of the utilized GNN and the overall importance of feature propagation mechanisms. One potential fruitful future direction is to explore the difference in learned representations arising from different expressive GNN variants through the lens of explainable AI, in which we would expect higher order variants to be able to explore rich subgraph patterns that cannot be detected by vanilla GNNs alone, but otherwise lead to similar explanations.

Furthermore, various enhancements to our task-specific models are feasible, *e.g.*, by extending them or applying them in different domains. For example, as shown in related work (Stärk *et al.*, 2021), working on and learning from the 3D molecular structure is generally preferable for the task of molecular property prediction. However, this information is often infeasible to compute as it requires expensive DFT-based geometry optimization. As such, one possible direction to explore is how our HIMP approach can be utilized to learn the 3D molecular structure directly from their 2D molecular graphs, *e.g.*, by using methods from self-supervised learning. Notably, special care must be taken in designing rotation-invariant loss formulations. Furthermore, our DGMC approach for the task of graph matching can be extended in order

to relax the one-to-one mapping constraint, as it currently expects that every node in the source graph can be exactly matched to one node in the target graph. It may be possible to bypass the matching of certain nodes, *e.g.*, by introducing dummy nodes to the target graph. In addition, DGMC can also be extended to the task of multiple object tracking, in which we would like to find correspondences between objects *over* time. Here, the refinement stage needs to be modified in order to reach a neighborhood consensus across potentially multiple timestamps.

Our GAS framework provides a crucial starting point in maintaining the properties of the underlying GNN while being able to scale GNNs up to giant graphs, which we like to extend further in future works. In particular, extensions to multi-GPU or distributed training scenarios are desirable, in which synchronization of historical accesses need to be performed on a per GPU/machine basis. Training GAS in a distributed fashion requires the distribution of histories as well, in which the most effective memory management strategy remains to be explored, *i.e.* histories can either be distributed across machines per layer or per mini-batch. In both cases, remote process communication of historical embeddings needs to be achieved concurrently. Furthermore, we would like to extend GAS to support both temporal and heterogeneous graphs as well. In particular, for learning on temporal graphs, historical embeddings need to be obtained per layer and timestamp, in which storage capacities can become a major bottleneck depending on the chosen temporal window size. As such, quantization (Ding *et al.*, 2021), auto-encoding procedures or disk-accessible histories are necessary to avoid CPU memory requirements from becoming too large.

Lastly, our PyTorch Geometric library will continuously expand its scope to provide immediate access to recent advancements in Graph Representation Learning, aiming to accelerate future research progress and to make latest GNN trends instantaneously deployable in industrial applications. On low levels, future work will resolve around the design of more memory-efficient edge-level message passing computations through a general kernel fusion interface, similar to the design of the KeOps library (Feydy *et al.*, 2020). On high levels, we aim to achieve a scalability-agnostic GNN interface, building upon the initial work of our GAS framework. That is, users should be able to write their GNN models and training procedures as if they are expecting full-batch graphs as input, and PyG will automatically take care of applying the appropriate changes to scale GNN execution to any graph size. To accelerate the immediate integration of new features, we also plan to integrate a Model Hub. With this, users can simply register their customized and pre-trained GNN models to make them directly accessible within PyG. Lastly, besides the support of homogeneous graphs and heterogeneous graphs, we aim to extend the scope of PyG to temporal graphs (Rozemberczki *et al.*, 2021b) and knowledge graphs (Ren *et al.*, 2021) as well.

Overall, Graph Neural Networks also yield the potential to revolutionize machine learning on (relational) tabular data. While the majority of data scientists and machine learning practitioners use relational data in their work, there is still significant data extraction and feature engineering efforts required to “flatten” data points into independent and identically distributed examples (Cvitkovic, 2020). In contrast, GNNs can reason about the relational nature of the underlying data entirely on their own. As such, they may well be able to rapidly pioneer a new era of machine learning models that data scientists will frequently rely on in their day-to-day work.

Bibliography

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. `https://www.tensorflow.org`, 2015. (last access: August 25, 2022).

R. Abboud, I. I. Ceylan, M. Grohe, and T. Lukasiewicz. The surprising power of graph neural networks with random node initialization. *CoRR*, abs/2010.01179, 2020.

L. Abraham. Accelerating node2vec with rejection sampling. `https://louisabraham.github.io/articles/node2vec-sampling`, 2020. (last access: August 25, 2022).

S. Abu-El-Haija, B. Perozzi, R. Al-Rfou, and A. Alemi. Watch your step: Learning node embeddings via graph attention. In *NeurIPS*, 2018.

L. Adamic and E. Adar. Friends and neighbors on the web. *Social Networks*, 25(3), 2003.

R. P. Adams and R. S. Zemel. Ranking via sinkhorn propagation. *CoRR*, abs/1106.1925, 2011.

R. Addanki, P. W. Battaglia, D. Budden, A. Deac, J. Godwin, T. Keck, W. L. Sibon Li, A. Sanchez-Gonzalez, J. Stott, S. Thakoor, and P. Veličković. Large-scale graph representation learning with very deep GNNs and self-supervision. *CoRR*, abs/2107.09422, 2021.

Y. Aflalo, A. Bronstein, and R. Kimmel. On convex relaxation of graph isomorphism. *Proceedings of the National Academy of Sciences*, 112(10), 2015.

A. Agarwal and A. V. Mangal. Visual relationship detection using scene graphs: A survey. *CoRR*, abs/2005.08045, 2020.

M. Albooyeh, D. Bertolini, and S. Ravanbaksh. Incidence networks for geometric deep learning. *CoRR*, abs/1905.11460, 2019.

Alibaba. Euler. `https://github.com/alibaba/euler`, 2019. (last access: August 25, 2022).

M. Allamanis. The adverse effects of code duplication in machine learning models of code. In *ACM SIGPLAN*, 2019.

M. Allamanis, H. Peng, and C. Sutton. A convolutional attention network for extreme summarization of source code. In *ICML*, 2016.

M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. *CoRR*, abs/1711.00740, 2017.

M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

U. Alon and E. Yahav. On the bottleneck of graph neural networks and its practical implications. In *ICLR*, 2021.

U. Alon, S. Brody, O. Levy, and E. Yahav. code2seq: Generating sequences from structured representations of code. *CoRR*, abs/1808.01400, 2018.

U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3:1–29, 2019.

A. Angerd, K. Balasubramanian, and M. Annavaram. Distributed training of graph convolutional networks using subgraph approximation. *ICLR submission*, 2020.

K. Anstreicher. Recent advances in the solution of quadratic assignment problems. *Mathematical Programming*, 97, 2003.

V. Arvind, J. Köbler, G. Rattan, and O. Verbitsky. On the power of color refinement. In *International Symposium on Fundamentals of Computation Theory*, 2015.

V. Arvind, F. Fuhlbrück, J. Köbler, and O. Verbitsky. On Weisfeiler-Lehman invariance: Subgraph counts and related graph properties. In *International Symposium on Fundamentals of Computation Theory*, 2019.

A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *SC*, 2014.

L. Babai. Graph isomorphism in quasipolynomial time. In *ACM SIGACT Symposium on Theory of Computing*, 2016.

L. Babai, P. Erdős, and S. M. Selkow. Random graph isomorphism. *SIAM Journal on Computing*, 9(3):628–635, 1980.

J. Baek, M. Kang, and S. J. Hwang. Accurate learning of graph representations with graph multiset pooling. In *ICLR*, 2021.

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

S. Bai, F. Zhang, and P. H. S. Torr. Hypergraph convolution and hypergraph attention. *Pattern Recognition*, 110, 2021.

Y. Bai, H. Ding, Y. Sun, and W. Wang. Convolutional neural networks for fast approximation of graph edit distance. *CoRR*, abs/1809.0440, 2018.

Baidu. PGL: Paddle graph learning. `https://github.com/PaddlePaddle/PGL`, 2019. (last access: August 25, 2022).

J. Bajorath. Integration of virtual and high-throughput screening. *Nature Reviews Drug Discovery*, 1(11):882–894, 2002.

M. Balog, B. van Merriënboer, S. Moitra, Y. Li, and D. Tarlow. Fast training of sparse graph neural networks on dense hardware. *CoRR*, abs/1906.11786, 2019.

J. Barker, R. Marxer, E. Vincent, and S. Watanabe. The third 'CHiME' speech separation and recognition challenge: Dataset, task and baselines. In *ASRU*, 2015.

A. P. Bartók, R. Kondor, and G. Csányi. On representing chemical environments. *Physical Review B*, 87(18), 2013.

M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. *IBGM Research Report RC24704*, 2009.

P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. F. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, Ç. Gülçehre, F. Song, A. J. Ballard, J. Gilmer, G. E. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu. Relational inductive biases, deep learning, and graph networks. *CoRR*, abs/1806.01261, 2018.

M. Bayati, D. F. Gleich, A. Saberi, and Y. Wang. Message-passing algorithms for sparse network alignment. *ACM Transactions on Knowledge Discovery from Data*, 7(1), 2013.

J. Behler and M. Parrinello. Generalized neural-network representation of high-dimensional potential-energiy surfaces. *Physical Review Letters*, 98(1), 2007.

N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. *NVIDIA Technical Report NVR-2008-004*, 2008.

N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC*, 2009.

Y. Bengio, R. Ducharme, and P. Vincent. A neural probabilistic language model. In *NIPS*, 2010.

J. Bento and S. Ioannidis. A family of tractable graph distances. In *SDM*, 2018.

J. Bezanson, A. Edelman, S. Kapinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

K. Bhatia, K. Dahiya, H. Jain, A. Mittal, Y. Prabhu, and M. Varma. The extreme classification repository: Multi-label datasets and code. `http://manikvarma.org/downloads/XC/XMLRepository.html`, 2016. (last access: August 25, 2022).

Y. Bia, H. Ding, S. Bian, T. Chen, Y. Sun, and W. Wang. SimGNN: A neural network approach to fast graph similarity computation. In *WSDM*, 2019.

F. M. Bianchi, D. Grattarola, L. Livi, and C. Alippi. Graph neural networks with convolutional ARMA filters. *CoRR*, abs/1901.01343, 2019.

F. M. Bianchi, D. Grattarola, and C. Alippi. Spectral clustering with graph neural networks for graph pooling. In *ICML*, 2020.

N. Biggs, E. K. Lloyd, and R. J. Wilson. *Graph Theory, 1736-1936*. Oxford University Press, 1986.

G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. *Technical Report CMU-CS-93-173*, 1993.

G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

D. Bo, X. Wang, C. Shi, and H. Shen. Beyond low-frequency information in graph convolutional networks. In *AAAI*, 2021.

F. Bogo, J. Romero, M. Loper, and M. J. Black. FAUST: Dataset and evaluation for 3D mesh registration. In *CVPR*, 2014.

P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5, 2017.

A. Bojchevski and S. Günnemann. Deep gaussian embedding of attributed graphs: Unsupervised inductive learning via ranking. In *ICLR*, 2018.

A. Bojchevski, J. Klicpera, B. Perozzi, A. Kapoor, M. Blais, B. Ròzemberczki, M. Lukasik, and S. Günnemann. Scaling graph neural networks with approximate PageRank. In *KDD*, 2020.

K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: A collaboratively created graph database for structuring human knowledge. In *SIGMOD*, 2008.

A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko. Translating embeddings for modeling multi-relational data. In *NIPS*, 2013.

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

K. M. Borgwardt, C. S. Ong, S. Schönhauer, S. V. N. Vishwanathan, A. J. Smola, and H. P. Kriegel. Protein function prediction via graph kernels. *Bioinformatics*, 21(1): 47–56, 2005.

D. Boscaini, J. Masci, E. Rodolà, and M. Bronstein. Learning shape correspondence with anisotropic convolutional neural networks. In *NIPS*, 2016.

L. Bottoue and O. Bousquet. The tradeoffs of large scale learning. In *NIPS*, 2007.

S. Bougleux, L. Brun, V. Carletti, P. Foggia, B. Gaüzère, and M. Vento. Graph edit distance as a quadratic assignment problem. *Pattern Recognition Letters*, 87, 2017.

L. Bourdev and J. Malik. Poselets: Body part detectors trained using 3D human pose annotations. In *ICCV*, 2009.

G. Bouritsas, F. Frasca, S. Zafeiriou, and M. M. Bronstein. Improving graph neural network expressivity via subgraph isomorphism counting. In *ICML-W*, 2020.

J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: Composable transformations of Python+NumPy programs. `http://github.com/google/jax`, 2018. (last access: August 25, 2022).

X. Bresson and T. Laurent. Residual gated graph convnets. *CoRR*, abs/1711.07553, 2017.

A. Brock, J. Donahue, and K. Simonyan. Large-scale GAN training for high fidelity natural image synthesis. In *ICLR*, 2019.

M. Brockschmidt. GNN-FiLM: Graph neural networks with feature-wise linear modulation. In *ICML*, 2020.

S. Brody, U. Alon, and E. Yahav. How attentive are graph attention networks? *CoRR*, abs/2105.14491, 2021.

H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters*, 18(8), 1997.

T. S. Caetano, J. J. McAuley, L. Cheng, Q. V. Le, and A. J. Smola. Learning graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(6), 2009.

C. Cai and Y. Wang. A simple yet effective baseline for non-attribute graph classification. *CoRR*, abs/1811.03508, 2018.

J. Cai, M. Fürer, and N. Immerman. An optimal lower bound on the number of variables for graph identifications. *Combinatorica*, 12(4):389–410, 1992.

T. Cai, S. Luo, K. Xu, D. He, T. Y. Liu, and L. Wang. GraphNorm: A principled approach to accelerating graph neural network training. *CoRR*, abs/2009.03294, 2020.

C. Cangea, P. Veličković, N. Jovanović, T. N. Kipf, and P. Liò. Towards sparse hierarchical graph classifiers. In *NeurIPS-W*, 2018.

W. Cao and J. Xue. Recent progress in organic photovoltaics: Device architecture and optical design. *Energy Environmental Science*, 7(7):2123–2144, 2014.

Y. Cao, Z. Liu, C. Li, Z. Liu, J. Li, and T. Chua. Multi-channel graph neural network for entity alignment. In *ACL*, 2019.

Q. Cappart, D. Chételat, E. Khalil, A. Lodi, C. Morris, and P. Veličković. Combinatorial optimization and reasoning with graph neural networks. *CoRR*, abs/2102.09544, 2021.

I. Chami, S. Abu-El-Haija, B. Perozzi, C. Ré, and K. Murphy. Machine learning on graphs: A model and comprehensive taxonomy. *CoRR*, abs/2005.03675, 2020.

A. X. Chang, T. Funkhouser, L. J. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, J. Xiao, L. Yi, and F. Yu. ShapeNet: An information-rich 3D model repository. *CoRR*, abs/1512.03012, 2015.

C. C. Chang and C. J. Lin. LibSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(27):1–27, 2011.

X. Chang, P. Ren, P. Xu, Z. Li, X. Chen, and A. Hauptmann. Scene graphs: A survey of generations and applications. *CoRR*, abs/2104.01111, 2021.

C. Chen, W. Ye, Y. Zuo, C. Zheng, and S. P. Ong. Graph networks as a universal machine learning framework for molecules and crystals. *Chemistry of Materials*, 32 (9):3564–3572, 2019a.

H. Chen, B. Perozzi, Y. Hu, and S. Skiena. HARP: Hierarchical representation learning for networks. In *AAAI*, 2018a.

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

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

M. Chen, Z. Wei, B. Ding, Y. Li, Y. Yuan, X. Du, and J. R. Wen. Scalable graph neural networks via bidirectional propagation. In *NeurIPS*, 2020a.

M. Chen, Z. Wei, Z. Huang, B. Ding, and Y. Li. Simple and deep graph convolutional networks. In *ICML*, 2020b.

X. Chen, H. Huo, J. Huan, and J. S. Vitter. An efficient algorithm for graph edit distance computation. *Knowledge-Based Systems*, 163, 2019b.

W. L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C. J. Hsieh. Cluster-GCN: An efficient algorithm for training deep and large graph convolutional networks. In *KDD*, 2019.

K. Cho, B. van Merrienboar, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, 2014.

M. Cho, K. Alahari, and J. Ponce. Learning graphs to match. In *ICCV*, 2013.

F. Chollet. Xception: Deep learning with depthwise separable convolutions. In *CVPR*, 2017.

F. Chollet et al. Keras. `https://keras.io`, 2015. (last access: August 25, 2022).

C. B. Choy, J. Gwak, S. Savarese, and M. Chandraker. Universal correspondence network. In *NIPS*, 2016.

J. Clayden, N. Greeves, S. Warren, and P. Wothers. *Organic Chemistry*. Oxford University Press, 2001.

T. S. Cohen and M. Welling. Group equivariant convolutional networks. In *ICML*, 2016.

W. Cong, R. Forsati, M. Kandemir, and M. Mahdavi. Minimal variance sampling with provable guarantees for fast training of graph neural networks. In *KDD*, 2020.

Gene Ontology Consortium. The gene ontology resource: 20 years and still GOing strong. *Nucleic Acids Research*, 47(1):330–338, 2018.

G. Corso, L. Cavalleri, D. Beaini, P. Liò, and P. Veličković. Principal neighbourhood aggregation for graph nets. In *NeurIPS*, 2020.

T. Cour, P. Srinivasan, and J. Shi. Balanced graph matching. In *NIPS*, 2006.

L. Cowen, T. Ideker, B. J. Raphael, and R. Sharan. Network propagation: A universal amplifier of genetic associations. *Nature Reviews Genetics*, 18(9), 2017.

CSIRO-Data61. StellarGraph machine learning library. `https://github.com/stellargraph/stellargraph`, 2018. (last access: August 25, 2022).

M. Cvitkovic. Supervised learning on relational databases with graph neural networks. *CoRR*, abs/2002.02046, 2020.

S. Dalton, L. Olson, and N. Bell. Optimizing sparse matrix—matrix multiplication for the GPU. *ACM Transactions on Mathematical Software*, 41(25), 2015.

G. Dasoulas, L. Dos Santos, K. Scaman, and A. Virmaux. Coloring graph neural networks for node disambiguation. In *IJCAI*, 2020.

A. P. Davis, C. J. Grondin, R. J. Johnson, D. Sciaky, R. McMorran, J. Wiegers, T. C. Wiegers, and C. J. Mattingly. The comparative toxicogenomics database. *Nucleic Acids Research*, 47(1):948–954, 2019.

T. Davis. Wilkinson's sparse matrix definition. *NA Digest*, 7(12):379–401, 2007.

P. de Haan, T. S. Cohen, and M. Welling. Natural graph networks. In *NeurIPS*, 2020.

D. De Juan, F. Pazos, and A. Valencia. Emerging methods in protein co-evolution. *Nature Reviews Genetics*, 14(4):249–261, 2013.

J. Dean. Introducing pathways: A next-generation AI architecture. `https://blog-google.cdn.ampproject.org/c/s/blog.google/technology/ai/introducing-pathways-next-generation-ai-architecture/amp`, 2021. (last access: August 25, 2022).

A. S. Debnath, R. L. Lopez de Compadre, G. Debnath, A. J. Shusterman, and C. Hansch. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbitral energies and hydrophobicity. *Jorunal of Medicinal Chemistry*, 34(2):786–797, 1991.

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

M. H. DeGroot and M. J. Schervish. *Probability and Statistics*. Addison-Wesley, 2012.

H. Deng, T. Birdal, and S. Ilic. PPFNet: Global context aware local features for robust 3D point matching. In *CVPR*, 2018.

J. Deng, W. Dong, R. Socher, L. J. Li, K. Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *CVPR*, 2009.

T. Derr, Y. Ma, and J. Tang. Signed graph convolutional networks. In *ICDM*, 2018.

T. Derr, H. Karimi, X. Liu, J. Xu, and J. Tang. Deep adversarial network alignment. *CoRR*, abs/1902.10307, 2019.

J. Devlin, M. W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

I. S. Dhillon, Y. Guan, and B. Kulis. Weighted graph cuts without eigenvectors: A multilevel approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(11):1944–1957, 2007.

F. Diehl, T. Brunner, M. T. Le, and A. Knoll. Towards graph pooling by edge contraction. In *ICML Workshop on Learning and Reasoning with Graph-Structured Data*, 2019.

E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *ICLR*, 2020.

M. Ding, K. Kong, J. Li, C. Zhu, J. Dickerson, F. Huang, and T. Goldstein. VQ-GNN: A universal framework to scale-up graph neural networks using vector quantization. In *NeurIPS*, 2021.

Y. Dong, N. V. Chawla, and A. Swami. metapath2vec: Scalable representation learning for heterogeneous networks. In *KDD*, 2017a.

Y. Dong, H. Ma, Z. Shen, and K. Wang. A century of science: Globalization of scientific collaborations, citations, and innovations. In *KDD*, 2017b.

J. Du, S. Zhang, G. Wu, J. M. F. Moura, and S. Kar. Topology adaptive graph convolutional networks. *CoRR*, abs/1710.10370, 2017.

D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguire, R. Gómez-Bombarelli, T. Hirzel, A. Aspuri-Guzik, and R. P. Adams. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*, 2015.

V. P. Dwivedi, C. K. Joshi, T. Laurent, Y. Bengio, and X. Bresson. Benchmarking graph neural networks. *CoRR*, abs/2003.00982, 2020.

D. Easley and J. Kleinberg. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, 2010.

A. Egozi, Y. Keller, and H. Guterman. A probabilistic approach to spectral graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(1), 2013.

P. Erdős and A. Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6, 1959.

F. Errica, M. Podda, D. Bacciu, and A. Micheli. A fair comparison of graph neural networks for graph classification. In *ICLR*, 2020.

M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The Pascal visual object classes (VOC) challenge. In *IJCV*, 2010.

B. O. Fagginger Auer and R. H. Bisseling. A GPU algorithm for greedy graph matching. In *Facing the Multicore - Challenge II - Aspects of New Paradigms and Technologies in Parallel Computing*, 2011.

W. Falcon et al. PyTorch Lightning: The lightweight PyTorch wrapper for high-performance AI research. `https://www.pytorchlightning.ai`, 2019. (last access: August 25, 2022).

Y. Feng, H. You, Z. Zhang, R. Ji, and Y. Gao. Hypergraph neural networks. In *AAAI*, 2019.

A. Feragen, N. Kasenbug, J. Petersen, M. D. Bruijne, and K. M. Borgwardt. Scalable kernels for graphs with continuous attributes. In *NIPS*, 2013.

L. G. Ferreira, R. N. Dos Santos, G. Oliva, and A. D. Andricopulo. Molecular docking and structure-based drug design strategies. *Molecules*, 20(7):13384–13421, 2015.

M. Feurer, J. N. van Rijn, A. Kadra, P. Gijsbers, N. Mallik, S. Ravi, A. Müller, J. Vanschoren, and F. Hutter. OpenML-Python: An extensible python API for OpenML. *CoRR*, abs/1911.02490, 2019.

M. Fey. Just jump: Dynamic neighborhood aggregation in graph neural networks. In *ICLR Workshop on Representation Learning on Graphs and Manifolds (RLGM)*, 2019.

M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds (RLGM)*, 2019.

M. Fey, J. E. Lenssen, F. Weichert, and H. Müller. SplineCNN: Fast geometric deep learning with continuous B-spline kernels. In *Computer Vision and Pattern Recognition (CVPR)*, 2018.

M. Fey, J. E. Lenssen, J. Masci, and N. M. Kriege. Deep graph matching consensus. In *International Conference on Learning Representations (ICLR)*, 2020a.

M. Fey, J. G. Yuen, and F. Weichert. Hierarchical inter-message passing for learning on molecular graphs. In *ICML Workshop on Graph Representation Learning and Beyond (GRL+)*, 2020b.

M. Fey, J. E. Lenssen, F. Weichert, and J. Leskovec. GNNAutoScale: Scalable and expressive graph neural networks via historical embeddings. In *International Conference on Machine Learning (ICML)*, 2021.

J. Feydy, J. Glaunès, B. Charlier, and M. M. Bronstein. Fast geometric learning with symbolic matrices. In *NeurIPS*, 2020.

S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo. Sparse matrix-vector multiplication on GPGPUs. *ACM Transactions on Mathematical Software*, 43(4), 2017.

FluxML. GeometricFlux.jl. `https://github.com/FluxML/GeometricFlux.jl`, 2020. (last access: August 25, 2022).

F. Frasca, E. Rossi, D. Eynard, B. Chamberlain, M. M. Bronstein, and F. Monti. SIGN: Scalable inception graph neural networks. In *ICML-W*, 2020.

H. Fröhlich, J. K. Wegner, F. Sieker, and A. Zell. Optimal assignment kernels for attributed molecular graphs. In *ICML*, 2005.

T. Gale, M. Zaharia, C. Young, and E. Elsen. Sparse GPU kernels for deep learning. In *SC*, 2020.

H. Gao and S. Ji. Graph U-Nets. In *ICML*, 2019.

M. R. Garey. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 2002.

V. K. Garg, S. Jegelka, and T. S. Jaakkola. Generalization and representational limits of graph neural networks. *CoRR*, abs/2002.06157, 2020.

T. Gärtner, P. Flach, and S. Wrobel. On graph kernels: Hardness results and efficient alternatives. In *Learning Theory and Kernel Machines*, pp. 129–143. Springer, 2003.

J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *ICML*, 2017.

X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *AISTATS*, 2011.

J. Godwin, T. Keck, P. Battaglia, V. Bapst, T. Kipf, Y. Li, K. Stachenfeld, P. Veličković, and A. Sanchez-Gonzalez. Jraph: A library for graph neural networks in JAX. `https://github.com/deepmind/jraph`, 2020. (last access: August 25, 2022).

S. Gold and A. Rangarajan. A graduated assignment algorithm for graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(4), 1996.

R. Gómez-Bombarelli, J. N. Wei, D. Duvenaud, J. M. Hernández-Lobato, B. Sánchez-Lengeling, D. Sheberla, J. Aguilera-Iparraguirre, T. D. Hirzel, R. P. Adams, and A. Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS Central Science*, 2018.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

M. Gori, M. Maggini, and L. Sarti. Exact and approximate graph matching using random walks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(7), 2005a.

M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *IJCNN*, 2005b.

K. Gouda and M. Hassaan. CSI_GED: An efficient approach for graph edit similarity computation. In *ICDE*, 2016.

H. Gouk, E. Frank, B. Pfahringer, and M. J. Cree. Regularisation of neural networks by enforcing Lipschitz continuity. *CoRR*, abs/1804.04368, 2018.

D. Grattarola and C. Alippi. Graph neural networks in TensorFlow and Keras with Spektral. In *ICML Workshop on Graph Representation Learning and Beyond (GRL+)*, 2020.

D. Grattarola, D. Zambon, F. M. Bianchi, and C. Alippi. Understanding pooling in graph neural networks. *CoRR*, abs/2110.05292, 2021.

J. Griffith and L. Orgel. Ligand-field theory. *Quarterly Reviews, Chemical Society*, 11(4): 381–393, 1957.

M. Grohe. *Descriptive Complexity, Canonisation, and Definable Graph Structure Theory*. Cambridge University Press, 2017.

M. Grohe, K. Kersting, M. Mladenov, and E. Selman. Dimension reduction via colour refinement. In *European Symposium on Algorithms*, 2014.

A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *KDD*, 2016.

A. Grover, A. Zweig, and S. Ermon. Graphite: Iterative generative modeling of graphs. In *ICML*, 2019.

P. Guerrero, Y. Kleiman, M. Ovsjanikov, and N. J. Mitra. PCPNet: Learning local shape properties from raw point clouds. *Computer Graphics Forum*, 37, 2018.

E. Guney. Reproducible drug repurposing: When similarity does not suffice. In *Pacific Symposium on Biocomputing*, 2017.

Q. Guo, F. Zhuang, C. Qin, H. Zhu, X. Xie, H. Xiong, and Q. He. A survey on knowledge graph-based recommender systems. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

B. Ham, M. Cho, C. Schmid, and J. Ponce. Proposal flow. In *CVPR*, 2016.

W. L. Hamilton. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159, 2020.

W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NIPS*, 2017.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

M. Heimann, H. Shen, T. Safavi, and D. Koutra. REGAL: Representation learning-based graph alignment. In *CIKM*, 2018.

T. Helgaker, P. Jorgensen, and J. Olsen. *Molecular Electronic-Structure Theory*. John Wiley & Sons, 2014.

C. Helma, R. D. King, S. Kramer, and A. Srinivasan. The Predictive Toxicology Challenge 2000–2001. *Bioinformatics*, 17(1):107–108, 2001.

R. Henderson, D. A. Clevert, and F. Montanari. Improving molecular graph neural network explainability with orthonormalization and induced sparsity. In *ICML*, 2021.

L. Hermansson, F. D. Joansson, and O. Watanabe. Generalized shortest path kernel on graphs. In *Discovery Science: International Conference*, 2015.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9 (8), 1997.

S. Hoory, N. Linial, and A. Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, 2006.

K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

J. Hu, S. Qian, Q. Fang, Y. Wang, Q. Zhao, H. Zhang, and C. Xu. Efficient graph deep learning in TensorFlow with tf_geometric. In *ACM Multimedia Conference (MM)*, pp. 3775–3778, 2021a.

W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec. Open Graph Benchmark: Datasets for machine learning on graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020a.

W. Hu, B. Liu, J. Gomes, M. Zitnik, P. Liang, V. Pande, and J. Leskovec. Strategies for pre-training graph neural networks. In *ICLR*, 2020b.

W. Hu, M. Fey, H. Ren, M. Nakata, Y. Dong, and J. Leskovec. OGB-LSC: Large-scale challenge for machine learning on graphs. In *NeurIPS: Datasets and Benchmarks Track*, 2021b.

Z. Hu, Y. Dong, K. Wang, and Y. Sun. Heterogeneous graph transformer. In *WWW*, 2020c.

G. Huang, G. Dai, Y. Wang, and H. Yang. GE-SpMM: General-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

Q. Huang, H. He, A. Singh, S. N. Lim, and A. R. Benson. Combining label propagation and simple models out-performs graph neural networks. In *ICLR*, 2021.

W. Huang, T. Zhang, Y. Rong, and J. Huang. Adaptive sampling towards fast graph representation learning. In *NeurIPS*, 2018.

L. A. Hug, B. J. Baker, K. Anantharaman, C. T. Brown, A. J. Probst, C. J. Castelle, C. N. Butterfield, A. W. Hernsdorf, Y. Amano, K. Ise, Y. Suzuki, N. Dudek, D. A. Relman, K. M. Finstad, R. Amundson, B. C. Thomas, and J. F. Banfield. A new view of the tree of life. *Nature Microbiology*, 1(5), 2016.

H. Husain, H. H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019.

M. T. Hussain, O. Selvitopi, A. Buluc, and A. Azad. Communication-avoiding and memory-constrained sparse matrix-matrix multiplication at extreme scale. *CoRR*, abs/2010.08526, 2020.

F. Huszár. How powerful are graph convolutions? `https://www.inference.vc`, 2016. (last access: August 25, 2022).

T. S. Hy, S. Trivedi, H. Pan, B. Anderson, and R. Kondor. Predicting molecular properties with covariant compositional networks. *The Journal of Chemical Physics*, 148 (24), 2018.

N. Immerman and E. Lander. Describing graphs: A first-order approach to graph canonization. In *Complexity Theory Retrospective: In Honor of Juris Hartmanis on the Occasion of His Sixtieth Birthday*, 1990.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

K. Ishiguro, S. Maeda, and M. Koyama. Graph warp module: An auxiliary module for boosting the power of graph neural networks. *CoRR*, abs/1902.01020, 2019.

S. Ivanov, S. Sviridov, and E. Burnaev. Understanding isomorphism bias in graph data sets. *CoRR*, abs/1910.12091, 2019.

H. Jackson-Flux, M. Brockschmidt, M. Stanley, and P. Cameron. Graph neural networks in TF2. `https://github.com/microsoft/tf2-gnn`, 2019. (last access: August 25, 2022).

M. Jaggi. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In *ICML*, 2013.

Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken. Improving the accuracy, scalability, and performance of graph neural networks with ROC. *Proceedings of Machine Learning and Systems*, pp. 187–198, 2020.

W. Jin, R. Barzilay, and T. Jaakkola. Junction tree variational autoencoder for molecular graph generation. In *ICML*, 2018.

W. Jin, K. Yang, R. Barzilay, and T. Jaakkola. Learning multimodal graph-to-graph translation for molecule optimization. In *ICLR*, 2019.

W. Jin, M. Qu, X. Jin, and X. Ren. Recurrent event network: Autoregressive structure inference over temporal knowledge graphs. In *EMNLP*, 2020.

P. B. Jørgensen, K. W. Jacobsen, and M. N. Schmidt. Neural message passing with edge updates for predicting properties of molecules and materials. In *NIPS*, 2018.

C. Joshi. Transformers are graph neural networks. `https://graphdeeplearning.github.io`, 2020. (last access: August 25, 2022).

J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. J. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, 2021.

V. Kann. On the approximability of the maximum common subgraph problem. In *STACS*, 1992.

G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359—-392, 1998.

H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled graphs. In *ICML*, 2003.

N. Keriven and G. Peyré. Universal invariant and equivariant graph neural networks. In *NeurIPS*, 2019.

K. Kersting, M. Mladenov, R. Garnett, and M. Grohe. Power iterated color refinement. In *Conference on Artificial Intelligence (AAAI)*, 2014.

A. H. Khasahmadi, K. Hassani, P. Moradi, L. Lee, and Q. Morris. Memory-based graph networks. In *ICLR*, 2020.

S. Kiefer, P. Schweitzer, and E. Selman. Graphs identified by logics with counting. In *International Symposium on Mathematical Foundations of Computer Science*, 2015.

D. Kim and A. Oh. How to find your friendly neighborhood: Graph attention design with self-supervision. In *ICLR*, 2021.

V. G. Kim, Y. Lipman, and T. Funkhouser. Blended intrinsic maps. *ACM Trans. Graph.*, 30(4):1–12, 2011.

D. P. Kingma and J. L. Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

T. N. Kipf and M. Welling. Variational graph auto-encoders. In *NIPS-W*, 2016.

T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

G. W. Klau. A new graph-based method for pairwise global network alignment. *BMC Bioinformatics*, 10, 2009.

J. Klicpera, A. Bojchevski, and S. Günnemann. Predict then propagate: Graph neural networks meet personalized PageRank. In *ICLR*, 2019a.

J. Klicpera, S. Weißenberger, and S. Günnemann. Diffusion improves graph learning. In *NeurIPS*, 2019b.

J. Klicpera, S. Giri, J. T. Margraf, and S. Günnemann. Fast and uncertainty-aware directional message passing for non-equilibrium molecules. In *NeurIPS Workshop on Machine Learning for Molecules*, 2020a.

J. Klicpera, J. Groß, and S. Günnemann. Directional message passing for molecular graphs. In *ICLR*, 2020b.

B. Knyazev, X. Lin, M. R. Amer, and G. W. Taylor. Image classification with hierarchical multigraph networks. In *BMVC*, 2019a.

B. Knyazev, G. W. Taylor, and M. R. Amer. Understanding attention and generalization in graph neural networks. In *NeurIPS*, 2019b.

N. Kokhlikyan, V. Miglani, M. Martin, E. Wang, J. Reynolds, A. Melnikov, N. Lunova, and O. Reblitz-Richardson. PyTorch Captum. `https://captum.ai`, 2019. (last access: August 25, 2022).

G. Kollias, S. Mohammadi, and A. Grama. Network similarity decomposition (NSD): A fast and scalable approach to network alignment. *IEEE Tranactions on Knowledge and Data Engineering*, 24(12), 2012.

S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, and R. Sandstrom. The SuiteSparse matrix collection website interface. *Journal of Open Source Software*, 4(35), 2019.

N. M. Kriege and P. Mutzel. Subgraph matching kernels for attributed graphs. In *ICML*, 2012.

N. M. Kriege, P. L. Giscard, and R. C. Wilson. On valid optimal assignment kernels and applications to graph classification. In *NIPS*, 2016.

N. M. Kriege, C. Morris, A. Rey, and C. Sohler. A property testing framework for the theoretical expressivity of graph kernels. In *IJCAI*, 2018.

N. M. Kriege, P. L. Giscard, F. Bause, and R. C. Wilson. Computing optimal assignments in linear time for approximate graph matching. In *ICDM*, 2019a.

N. M. Kriege, L. Humbeck, and O. Koch. Chemical similarity and substructure searches. In *Encyclopedia of Bioinformatics and Computational Biology*. Academic Press, 2019b.

N. M. Kriege, F. D. Johansson, and C. Morris. A survey on graph kernels. *Applied Network Science*, 5(1), 2020.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS*, 2012.

M. J. Kusner, B. Paige, and J. M. Hernández-Lobato. Grammar variational autoencoder. In *ICML*, 2017.

G. Lample, A. Conneau, M. Ranzato, L. Denoyer, and H. Jégou. Word translation without parallel data. In *ICLR*, 2018.

G. Landrum. RDKit: Open-source cheminformatics software, 2016.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 1998.

J. Lee, I. Lee, and J. Kang. Self-attention graph pooling. In *ICML*, 2019.

M. Leordeanu and M. Hebert. A spectral technique for correspondence problems using pairwise constraints. In *ICCV*, 2005.

M. Leordeanu, M. Hebert, and R. Sukthankar. An integer projected fixed point method for graph matching and MAP inference. In *NIPS*, 2009.

A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich. PyTorch-BigGraph: A large-scale graph embedding system. *CoRR*, abs/1903.12287, 2019.

J. Lerouge, Z. Abu-Aisheh, R. Raveaux, P. Héroux, and S. Adam. New binary linear programming formulation to compute the graph edit distance. *Pattern Recognition*, 72, 2017.

R. J. LeVeque. Finite difference methods for ordinary and partial differential equations: Steady-state and time-dependent problems. *SIAM*, 98, 2007.

R. Levie, F. Monti, X. Bresson, and M. M. Bronstein. CayleyNets: Graph convolutional neural networks with complex rational spectral filters. *CoRR*, abs/1705.07664, 2017.

G. Li, M. Müller, A. Thabet, and B. Ghanem. DeepGCNs: Can GCNs go as deep as CNNs? In *ICCV*, 2019a.

J. Li, D. Cai, and X. He. Learning graph-level representation for drug discovery. *CoRR*, abs/1709.03741, 2017.

P. Li, Y. Wang, H. Wang, and J. Leskovec. Distance encoding: Design provably more powerful neural networks for graph representation learning. In *NeurIPS*, 2020.

Q. Li, Z. Han, and X. M. Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *AAAI*, 2018a.

W. Li, H. Saidi, H. Sanchez, M. Schäf, and P. Schweitzer. Detecting similar programs via the Weisfeiler-Lehman graph kernel. In *International Conference on Software Reuse*, 2016a.

Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. In *ICLR*, 2016b.

Y. Li, R. Bu, M. Sun, W. Wu, X. Di, and B. Chen. PointCNN: Convolution on $\mathcal{X}$-transformed points. In *NeurIPS*, 2018b.

Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli. Graph matching networks for learning the similarity of graph structured objects. In *ICML*, 2019b.

D. Liben-Nowell and J. M. Kleinberg. The link-prediction problem for social networks. *Journal of the Association for Information Science and Technology*, 58(7):1019–1031, 2007.

T. Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: Common objects in context. In *ECCV*, 2014.

T. Y. Lin, P. Dollar, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *CVPR*, 2017.

E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

O. Litany, T. Remez, E. Rodolà, A. Bronstein, and M. Bronstein. Deep functional maps: Structured prediction for dense shapre correspondence. In *ICCV*, 2017.

M. Liu, Y. Luo, L. Wang, Y. Xie, H. Yuan, S. Gui, H. Yu, Z. Xu, J. Zhang, Y. Liu, K. Yan, H. Liu, C. Fu, B. M. Oztekin, X. Zhang, and S. Ji. DIG: A turnkey library for diving into graph deep learning research. *Journal of Machine Learning Research*, 22(240):1–9, 2021.

Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019a.

Z. Liu, C. Chen, L. Li, J. Zhou, X. Li, L. Song, and Y. Qi. GeniePath: Graph neural networks with adaptive receptive paths. In *AAAI*, 2019b.

S. L. Lohr. *Sampling: Design and Analysis*. Nelson Education, 2009.

A. Loukas. What graph neural networks cannot learn: Depth vs width. In *ICLR*, 2020.

V. Lyzinski, D. E. Fishkind, M. Fiori, J. T. Vogelstein, C. E. Priebe, and G. Sapiro. Graph matching: Relax at your own risk. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(1), 2016.

L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai. NeuGraph: Parallel deep neural network computation on large graphs. In *USENIX Annual Technical Conference*, 2019.

Y. Ma and J. Tang. *Deep Learning on Graphs*. Cambridge University Press, 2020.

Z. Ma, J. Xuan, Y. G. Wang, M. Li, and P. Liò. Path integral based convolution and pooling for graph neural networks. In *NeurIPS*, 2021.

R. Macarron, M. N. Banks, D. Bojanic, D. J. Burns, D. A. Cirovic, T. Garyantes, D. V. S. Green, R. P. Hertzberg, W. P. Janzen, J. W. Paslay, U. Schopfer, and G. Sitta Sittampalam. Impact of high-throughput screening in biomedical research. *Nature Reviews Drug discovery*, 10(3):188–195, 2011.

N. Malod-Dognin, K. Ban, and N. Pržulj. Unified alignment of protein-protein interaction networks. *Scientific Reports*, 7(1):1–11, 2017.

E. Markowitz, K. Balasubramanian, M. Mirtaheri, S. Abu-El-Haija, B. Perozzi, G. Ver Steeg, and A. Galstyan. Graph traversal with tensor functionals: A meta-algorithm for scalable learning. In *ICLR*, 2021.

H. Maron, H. Ben-Hamu, H. Serviansky, and Y. Lipman. Provably powerful graph networks. In *NeurIPS*, 2019a.

H. Maron, H. Ben-Hamu, N. Shamir, and Y. Lipman. Invariant and equivariant graph networks. In *ICLR*, 2019b.

J. Masci, D. Boscaini, M. Bronstein, and P. Vandergheynst. Geodesic convolutional neural networks on riemannian manifolds. In *ICCV*, 2015.

D. W. Matula. Subtree isomorphism in $O(n^{5/2})$. In *Algorithmic Aspects of Combinatorics*, volume 2. Elsevier, 1978.

P. Mernyei and C. Cangea. Wiki-CS: A wikipedia-based benchmark for graph neural networks. In *ICML-W*, 2020.

D. Mesquita, A. H. Souza, and S. Kaski. Rethinking pooling in graph neural networks. In *NeurIPS*, 2020.

Microsoft. ptgnn: A PyTorch GNN library. `https://github.com/microsoft/ptgnn`, 2019. (last access: August 25, 2022).

T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.

R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

B. Min, R. Grishman, L. Wan, C. Wang, and D. Gondek. Distant supervision for relation extraction with an incomplete knowledge base. In *NAACL*, 2013.

G. Montavon, K. Hansen, S. Fazli, M. Rupp, F. Biegler, A. Ziehe, A. Tkatchenko, O. A. von Lilienfeld, and K. R. Müller. Learning invariant representations of molecules for atomization energy prediction. In *NIPS*, 2012.

F. Monti, D. Boscaini, J. Masci, E. Rodolà, J. Svoboda, and M. M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model CNNs. In *CVPR*, 2017.

H. L. Morgan. The generation of a unique machine description for chemical structures - a technique developed at chemical abstracts service. *Journal of Chemical Documentation*, 5(2):107–113, 1965.

C. Morris, N. M. Kriege, K. Kersting, and P. Mutzel. Faster kernel for graphs with continuous attributes via hashing. In *ICDM*, 2016.

C. Morris, K. Kersting, and P. Mutzel. Glocalized Weisfeiler-Lehman kernels: Global-local feature maps of graphs. In *ICDM*, 2017.

C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe. Weisfeiler and Leman go neural: Higher-order graph neural networks. In *Conference on Artificial Intelligence (AAAI)*, 2019.

C. Morris, N. M. Kriege, F. Bause, K. Kersting, P. Mutzel, and M. Neumann. TU-Dataset: a collection of benchmark datasets for learning with graphs. *CoRR*, abs/2007.08663, 2020a.

C. Morris, G. Rattan, and P. Mutzel. Weisfeiler and Leman go sparse: Towards scalable higher-order graph embeddings. In *NeurIPS*, 2020b.

C. Morris, M. Fey, and N. M. Kriege. The power of the Weisfeiler-Leman algorithm for machine learning with graphs. In *International Joint Conference on Artificial Intelligence (IJCAI) - Survey Track*, 2021a.

C. Morris, Y. Lipman, H. Maron, B. Rieck, N. M. Kriege, M. Grohe, M. Fey, and K. Borgwardt. Weisfeiler and Leman go machine learning: The story so far. *CoRR*, abs/2112.09992, 2021b.

R. L. Murphy, B. Srinivasan, V. Rao, and B. Ribeiro. Janossy pooling: Learning deep permutation-invariant functions for variable-size inputs. In *ICLR*, 2019a.

R. L. Murphy, B. Srinivasan, V. Rao, and B. Ribeiro. Relational pooling for graph representations. In *ICML*, 2019b.

M. Nakata. The PubChemQC project: A large chemical database from the first principle calculations. *AIP Conference Proceedings*, 2015.

M. Nakata and T. Shimazaki. PubChemQC project: A large-scale first-principles electronic structure database for data-driven chemistry. *Journal of Chemical Information and Modeling*, 57(6):1300–1308, 2017.

M. Naumov, L. S. Chien, P. Vandermersch, and U. Kapasi. cuSPARSE library: A set of basic linear algebra subroutines for sparse matrices. In *GTC*, 2010.

A. Newell, K. Yang, and J. Deng. Stacked hourglass networks for human pose estimation. In *ECCV*, 2016.

M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45 (2):167–256, 2003.

M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2015.

M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1), 2016.

J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

G. Nikolentzos, P. Meladianos, and M. Vazirgiannis. Matching node embeddings for graph similarity. In *Conference on Artificial Intelligence (AAAI)*, 2017.

H. NT and T. Maehara. Revisiting graph neural networks: All we have is low-pass filters. *CoRR*, abs/1905.09550, 2019.

F. Orsini, P. Frasconi, and L. De Raedt. Graph invariant kernels. In *IJCAI*, 2015.

L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

S. Pan, R. Hu, G. Long, J. Jiang, L. Yao, and C. Zhang. Adversarially regularized graph autoencoder for graph embedding. In *IJCAI*, 2018.

V. Panayotov, G. Chen, D. Povey, and S. Khudanpur. Librispeech: An ASR corpus based on public domain audio books. In *ICASSP*, 2015.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

B. Perozzi, R. Al-Rfou, and S. Skiena. DeepWalk: Online learning of social representations. In *KDD*, 2014.

G. Peyré, M. Cuturi, and J. Solomon. Gromov-Wasserstein averaging of kernel and distance matrices. In *ICML*, 2016.

L. Piegl and W. Tiller. *The NURBS Book*. Springer-Verlag New York, 1997.

J. Piñero, J. M. Ramírez-Anguita, J. Saüch-Pitarch, F. Ronzano, E. Centeno, F. Sanz, and L. I. Furlong. The DisGeNET knowledge platform for disease genomics. *Nucleic Acids Research*, 48(1):845–855, 2020.

S. R. Qasim, J. Kieseler, Y. Iiyama, and M. Pierini. Learning representations of irregular particle-detector geometry with distance-weighted graph networks. *European Physics Journal C*, 79, 2019.

C. R. Qi, H. Su, K. Mo, and L. J. Guibas. PointNet: Deep learning on point sets for 3D classification and segmentation. In *CVPR*, 2017a.

C. R. Qi, L. Yi, H. Su, and L. J. Guibas. PointNet++: Deep hierarchical feature learning on point sets in a metric space. In *NIPS*, 2017b.

J. Qiu, Y. Dong, H. Ma, J. Li, C. Wang, K. Wang, and J Tang. NetSMF: Large-scale network embedding as sparse matrix factorization. In *WWW*, 2019.

Quiver. Quiver: A distributed graph learning library for PyTorch Geometric. `https://github.com/quiver-team/torch-quiver`, 2021. (last access: August 25, 2022).

A. Radhakrishna, A. Shaji, K. Smith, A. Lucchi, P. Fua, and S. Süsstrunk. SLIC superpixels compared to state-of-the-art superpixels methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(11):2274–2282, 2012.

P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. SQuAD: 100000+ questions for machine comprehension of text. *CoRR*, abs/1606.05250, 2016.

R. Ramakrishnan, P. O. Dral, M. Rupp, and O. A. von Lilienfeld. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific Data*, 1(1):1–7, 2014.

A. Ranjan, T. Bolkart, S. Sanyal, and M. J. Black. Generating 3D faces using convolutional mesh autoencoders. In *ECCV*, 2018.

E. Ranjan, S. Sanyal, and P. P. Talukdar. ASAP: Adaptive structure aware pooling for learning hierarchical graph representations. In *AAAI*, 2020.

M. Rarey and J. S. Dixon. Feature trees: A new molecular similarity measure based on tree matching. *Journal of Computer-aided Molecular Design*, 12(5):471–490, 1998.

N. Reimers and I. Gurevych. Sentence-BERT: Sentence embeddings using siamese BERT-networks. In *EMNLP*, 2019.

H. Ren, H. Dai, B. Dai, X. Chen, D. Zhou, J. Leskovec, and D. Schuurmans. SMORE: Knowledge graph completion and multi-hop reasoning in massive knowledge graphs, 2021.

K. Riesen and H. Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing*, 27(7), 2009.

K. Riesen, M. Ferrer, R. Dornberger, and H. Bunke. Greedy graph edit distance. In *Machine Learning and Data Mining in Pattern Recognition*, 2015a.

K. Riesen, M. Ferrer, A. Fischer, and H. Bunke. Approximation of graph edit distance in quadratic time. In *Graph-Based Representations in Pattern Recognition*, 2015b.

I. Rocco, M. Cimpo, R. Arandjelović, A. Torii, T. Pajdla, and J. Sivic. Neighbourhood consensus networks. In *NeurIPS*, 2018.

D. Rogers and M. Hahn. Extended-connectivity fingerprints. *Journal of Chemical Information and Modeling*, 50(5):742–754, 2010.

D. W. Romero, A. Kuzina, E. J. Bekkers, J. M. Tomczak, and M. Hoogendoorn. CKConv: Continuous kernel convolution for sequential data. *CoRR*, abs/2102.02611, 2021.

Y. Rong, Y. Bian, T. Xu, W. Xie, Y. Wei, W. Huang, and J. Huang. GROVER: Self-supervised message passing transformer on large-scale molecular data. *CoRR*, abs/2007.02835, 2020a.

Y. Rong, W. Huang, T. Xu, and J. Huang. DropEdge: Towards deep graph convolutional networks on node classification. In *ICLR*, 2020b.

O. Ronneberger, P. Fischer, and T. Brox. U-Net: Convolutional networks for biomedical image segmentation. In *MICCAI*, 2015.

F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 1958.

E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein. Temporal graph networks for deep learning on dynamic graphs. In *ICML Workshop on Graph Representation Learning and Beyond (GRL+)*, 2020.

B. Rozemberczki, P. Englert, A. Kapoor, M. Blais, and B. Perozzi. Pathfinder discovery networks for neural message passing. In *WWW*, 2021a.

B. Rozemberczki, P. Scherer, Y. He, G. Panagopoulos, A. Riedel, M. Astefanoaei, O. Kiss, F. Beres, G. Lopez, N. Collignon, and R. Sarkar. PyTorch Geometric Temporal: Spatiotemporal signal processing with neural machine learning models. In *CIKM*, 2021b.

M. Rupp, A. Tkatchenko, K. R. Müller, and O. A. von Lilienfeld. Fast and accurate moldeing of molecular atomization energies with machine learning. *Physical Review Letters*, 108(5), 2012.

G. Salha, R. Hennequin, and M. Vazirgiannis. Simple and effective graph autoencoders with one-hop linear models. In *ICML-PKDD*, 2020.

A. Sanfeliu and K. S. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(3), 1983.

R. Sato, M. Yamada, and H. Kashima. Random features strengthen graph neural networks. *CoRR*, abs/2002.03155, 2020.

V. G. Satorras, E. Hoogeboom, and M. Welling. E(n) equivariant graph neural networks. *CoRR*, abs/2102.09844, 2021.

T. Sattler, B. Leibe, and L. Kobbelt. SCRAMSAC: Improving RANSAC's efficiency with a spatial consistency filter. In *ICCV*, 2009.

K. Scaman and A. Virmuax. Lipschitz regularity of deep neural networks: Analysis and efficient estimation. In *NeurIPS*, 2018.

F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. In *IEEE Transactions on Neural Networks*, 2009.

M. S. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling. Modeling relational data with graph convolutional networks. In *ESWC*, 2018.

C. Schmid and R. Mohr. Local grayvalue invariants for image retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(5), 1997.

S. S. Schoenholz, E. D. Cubuk, D. M. Sussman, E. Kaxiras, and A. J. Lui. A structural approach to relaxation in glassy liquids. *Nature Physics*, 2016.

B. Schölkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Adaptive Computation and Machine Learning. MIT Press, 2002.

K. Schütt, P. J. Kindermans, Huziel E. Sauceda F., S. Chmiela, A. Tkatchenko, and K. R. Müller. SchNet: A continuous-filter convolutional neural network for modeling quantum interactions. In *NIPS*, 2017.

G. Sen, G. Namata, M. Bilgic, and L. Getoor. Collective classification in network data. *AI Magazine*, 29, 2008.

S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Graphics Hardware*, pp. 97–106, 2007.

Y. Seo, A. Loukas, and N. Perraudin. Discriminative structural graph classification. *CoRR*, abs/1905.13422, 2019.

R. Sharan and T. Ideker. Modeling cellular machinery through biological network comparison. *Nature Biotechnology*, 24(4), 2006.

R. Sharan, S. Suthram, R. M. Kelley, T. Kuhn, S. McCuine, P. Uetz, T. Sittler, R. M. Karp, and T. Ideker. Conserved patterns of protein interaction in multiple species. *Proceedings of the National Academy of Sciences*, 102(6):1974–1979, 2005.

J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

O. Shchur, M. Mumme, A. Bojchevski, and S. Günnemann. Pitfalls of graph neural network evaluation. In *NeurIPS-W*, 2018.

N. Shervashidze, S. V. N. Vishwanathan, T. H. Petri, K. Mehlhorn, and K. M. Borgwardt. Efficient graphlet kernels for large graph comparison. In *AISTATS*, 2009.

N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-Lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011.

S. Shi, Q. Wang, and X. Chu. Efficient sparse-dense matrix-matrix multiplication on GPUs using the customized sparse storage format. *CoRR*, abs/2005.14469, 2020a.

Y. Shi, Z. Huang, W. Wang, H. Zhong, S. Feng, and Y. Sun. Masked label prediction: Unified message passing model for semi-supervised classification. *CoRR*, abs/2009.03509, 2020b.

D. I. Shuman, S. K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE Signal Processing Magazine*, 30(3), 2013.

M. Simonovsky and N. Komodakis. Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *CVPR*, 2017.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2014.

R. Singh, J. Xu, and B. Berger. Global alignment of multiple protein interaction networks with application to functional orthology detection. In *National Academy of Sciences*, 2008.

A. Singhal. Introducing the knowledge graph: Things, not strings. *Official Google Blog*, 5:16, 2012.

K. Sinha, S. Sodhani, J. Pineau, and W. L. Hamilton. Evaluating logical generalization in graph neural networks. *CoRR*, abs/2003.06560, 2020.

R. Sinkhorn and P. Knopp. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific Journal of Mathematics*, 21(2), 1967.

J. Sivic and A. Zisserman. Video Google: A text retrieval approach to object matching in videos. In *ICCV*, 2003.

N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15, 2014.

H. Stärk, D. Baeini, G. Corso, P. Tossou, C. Dallago, S. Günnemann, and P. Liò. 3D infomax improves GNNs for molecular property prediction, 2021.

M. Stauffer, T. Tschachtli, A. Fischer, and K. Riesen. A survey on applications of bipartite graph edit distance. In *Graph-Based Representations in Pattern Recognition*, 2017.

Z. Sun, W. Hu, and C. Li. Cross-lingual entity alignment via joint attribute-preserving embedding. In *ISWC*, 2017.

Z. Sun, W. Hu, Q. Zhang, and Y. Qu. Bootstrapping entity alignment with knowledge graph embedding. In *IJCAI*, 2018.

Z. Sun, Z. H. Deng, J. Y. Nie, and J. Tang. RotatE: Knowledge graph embedding by relational rotation in complex space. In *ICLR*, 2019.

P. Swoboda, C. Rother, H. A. Ahljaija, D. Kainmueller, and B. Savchynskyy. A study of lagrangean decompositions and dual ascent solvers for graph matching. In *CVPR*, 2017.

C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *ICLR*, 2014.

D. Szklarczyk, A. Santos, C. von Mering, L. J. Jensen, P. Bork, and M. Kuhn. STITCH 5: Augmenting protein–chemical interaction networks with tissue and affinity data. *Nucleic Acids Research*, 44(1):380–384, 2016.

D. Szklarczyk, A. L. Gable, D. Lyon, A. Junge, S. Wyder, J. Huerta-Cepas, M. Simonovic, N. T. Doncheva, J. H. Morris, P. Bork, et al. STRING v11: Protein–protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets. *Nucleic Acids Research*, 47(1):607–613, 2019.

S. A. Tailor, F. L. Opolka, P. Liò, and N. D. Lane. Adatpive filters and aggregator fusion for efficient graph convolutions. In *MLSys Workshop on Graph Neural Networks and Systems (GNNSys)*, 2021.

J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. LINE: Large-scale information network embedding. In *WWW*, 2015.

K. K. Thekumparampil, C. Wang, S. Oh, and L. Li. Attention-based graph neural network for semi-supervised learning. *CoRR*, abs/1803.03735, 2018.

G. Tinhofer. A note on compact graphs. *Discrete Applied Mathematics*, 30(2), 1991.

M. Togninalli, E. Ghisu, F. Llinares-López, B. Rieck, and K. M. Borgwardt. Wasserstein Weisfeiler-Lehman graph kernels. In *NeurIPS*, 2019.

F. Tombari, S. Salti, and L. Di Stefano. Unique signatures of histograms for local surface description. In *ECCV*, 2010.

A. Tripathy, K. Yelick, and A. Buluc. Reducing communcation in graph neural network training. *CoRR*, abs/2005.03300, 2020.

T. Trouillon, J. Welbl, S. Riedel, É. Gaussier, and G. Bouchard. Complex embeddings for simple link prediction. In *ICML*, 2016.

S. Umeyama. An eigendecomposition approach to weighted graph matching problems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(5), 1988.

O. T. Unke and M. Meuwly. PhysNet: A neural network for predicting energies, force, dipole moments, and partial charges. *Journal of Chemical Theory and Computation*, 15 (6):3678–3693, 2019.

M. Usama and D. E. Chang. Towards robust neural networks with Lipschitz continuity. *CoRR*, abs/1811.09008, 2018.

D. Valsesia, G. Fracastoro, and E. Magli. Don't stack layers in graph neural networks, wire them randomly. *ICLR submission*, 2020.

R. van den Berg, T. N. Kipf, and M. Welling. Graph convolutional matrix completion. *CoRR*, abs/1706.02263, 2017.

L. J. P. van der Maaten and G. E. Hinton. Visualizing high-dimensional data using t-SNE. *Journal of Machine Learning Research*, 5:2579–2605, 2008.

J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo. OpenML: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *NIPS*, 2017.

B. S. Veeling, J. Linmans, J. Winkens, T. S. Cohen, and M. Welling. Rotation equivariant CNNs for digital pathology. In *MICCAI*, 2018.

P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *ICLR*, 2018.

P. Veličković, W. Fedus, W. L. Hamilton, P. Liò, Y. Bengio, and R. D. Hjeml. Deep graph infomax. In *ICLR*, 2019.

M. Vento and P. Foggia. Graph matching techniques for computer vision. *Graph-Based Methods in Computer Vision: Developments and Applications*, 1, 2012.

N. Verma, E. Boyer, and J. Verbeek. FeaStNet: Feature-steered graph convolutions for 3D shape analysis. In *CVPR*, 2018.

C. Vignac, A. Loukas, and P. Frossard. Building powerful and equivariant graph neural networks with structural message-passing. In *NeurIPS*, 2020.

O. Vinyals, S. Bengio, and M. Kudlur. Order matters: Sequence to sequence for sets. In *ICLR*, 2016.

D. Vrandečić and M. Krötzsch. Wikidata: A free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014a.

D. Vrandečić and M. Krötzsch. Wikidata: A free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014b.

C. Wan, Y. Li, N. S. Kim, and Y. Lin. BDS-GCN: Efficient full-graph training of graph convolutional nets with partition-parallelism and boundary sampling. *ICLR submission*, 2020.

A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *CoRR*, abs/1804.07461, 2018a.

F. Wang, N. Xue, Y. Zhang, G. Xia, and M. Pelillo. A functional representation for graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019a.

K. Wang, Z. Shen, C. Huang, C. H. Wu, Y. Dong, and A. Kanakia. Microsoft Academic Graph: When experts are not enough. *Quantitative Science Studies*, 1(1):396–413, 2020.

M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang. Deep Graph Library: A graph-centric, highly-performant package for graph neural networks. *CoRR*, abs/1909.01315, 2019b.

R. Wang, J. Yan, and X. Yang. Learning combinatorial embedding networks for deep graph matching. In *ICCV*, 2019c.

X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, and P. S. Yu. Heterogeneous graph attention network. In *KDD*, 2019d.

Y. Wang and J. M. Solomon. Deep closest point: Learning representations for point cloud registration. In *ICCV*, 2019.

Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the GPU. In *Symposium on Principles and Practice of Parallel Programming (SIGPLAN)*, 2016.

Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon. Dynamic graph CNN for learning on point clouds. *ACM Transactions on Graphics (TOG)*, 2019e.

Z. Wang, Q. Lv, X. Lan, and Y. Zhang. Cross-lingual knowledge graph alignment via graph convolutional networks. In *EMNLP*, 2018b.

Z. Wang, X. Ye, C. Wang, J. Cui, and P. S. Yu. Network embedding with completly-imbalanced labels. *IEEE Transactions on Knowledge and Data Engineering*, 33(11), 2021.

D. Weininger. SMILES, a chemical language and information system: Introduction to methodology and encoding rules. *Journal of Chemical Information and Computer Sciences*, 28(1):31–36, 1988.

B. Weisfeiler. On construction and identification of graphs. In *Lecture Notes in Mathematics*. Springer, 1976.

B. Weisfeiler and A. A. Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9), 1968.

P. J. Werbos. Applications of advances in nonlinear sensitivity analysis. In *System Modeling and Optimization*. Springer, 1982.

M. Winter, D. Mlakar, R. Zayer, H. P. Seidel, and M. Steinberger. Adaptive sparse matrix-matrix multiplication on the GPU. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2019.

D. S. Wishart, Y. D. Feunang, A. C. Guo, E. J. Lo, A. Marcu, J. R. Grant, T. Sajed, D. Johnson, C. Li, Z. Sayeeda, et al. DrugBank 5.0: A major update to the DrugBank database for 2018. *Nucleic Acids Research*, 46(1):1074–1082, 2018.

F. Wu, T. Zhang, A. H. de Souza Jr., C. Fifty, T. Yu, and K. Q. Weinberger. Simplifying graph convolutional networks. In *ICML*, 2019a.

J. Wu, J. Sun, H. Sun, and G. Sun. Performance analysis of graph neural network frameworks. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021.

Y. Wu, X. Liu, Y. Feng, Z. Wang, R. Yan, and D. Zhao. Relation-aware entity alignment for heterogeneous knowledge graphs. In *IJCAI*, 2019b.

Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao. 3D ShapeNets: A deep representation for volumetric shapes. In *CVPR*, 2015.

Z. Wu, B. Ramsundar, E. N. Feinberg, J. Gomes, C. Geniesse, A. S. Pappu, K. Leswing, and V. Pande. MoleculeNet: A benchmark for molecular machine learning. *Chemical Science*, 9(1):513–530, 2018.

T. Xie and J. C. Grossman. Crystal graph convolutional neural networks for an accurate and interpretable prediction of material properties. *Physical Review Letters*, 120, 2018.

Z. Xiong, D. Wang, X. Liu, F. Zhong, X. Wan, X. Li, Z. Li, X. Luo, K. Chen, H. Jiang, and M. Zheng. Pushing the boundaries of molecular representation for drug discovery with the graph attention mechanism. *Journal of Medicinal Chemistry*, 63(16):8749–8760, 2021.

H. Xu, D. Luo, and L. Carin. Scalable Gromov-Wasserstein learning for graph partitioning and matching. *CoRR*, abs/1905.07645, 2019a.

H. Xu, D. Luo, H. Zha, and L. Carin. Gromov-wasserstein learning for graph matching and node embedding. In *ICML*, 2019b.

K. Xu, C. Li, Y. Tian, T. Sonobe, K. Kawarabayashi, and S. Jegelka. Representation learning on graphs with jumping knowledge networks. In *ICML*, 2018.

K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *ICLR*, 2019c.

K. Xu, L. Wang, M. Yu, Y. Feng, Y. Song, Z. Wang, and D. Yu. Cross-lingual knowledge graph alignment via graph matching neural network. In *ACL*, 2019d.

K. Xu, J. Li, M. Zhang, S. S. Du, K. Kawarabayashi, and S. Jegelka. What can neural networks reason about? In *ICLR*, 2020.

K. Xu, M. Zhang, J. Li, S. S. Du, K. Kawarabayashi, and S. Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. In *ICLR*, 2021.

J. Yan, X. C. Yin, W. Lin, C. Deng, H. Zha, and X. Yang. A short survey of recent advances in graph matching. In *ICMR*, 2016.

Y. Yan, M. Hashemi, K. Swersky, Y. Yang, and D. Koutra. Two sides of the same coin: Heterophily and oversmoothing in graph convolutional neural networks. *CoRR*, abs/2102.06462, 2021.

P. Yanardag and S. V. N. Vishwanathan. Deep graph kernels. In *KDD*, 2015.

B. Yang, W. Yih, X. He, J. Gao, and L. Deng. Embedding entities and relations for learning and inference in knowledge bases. In *ICLR*, 2015.

C. Yang, A. Buluc, and J. D. Owens. Design principles for sparse matrix multiplication on the GPU. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2018.

K. Yang, K. Swanson, W. Jin, C. Coley, P. Eiden, H. Gao, A. Guzman-Perez, T. Hopper, B. Kelley, M. Mathea, A. Palmer, V. Settels, T. Jaakkola, K. Jensen, and R. Barzilay. Analyzing learned molecular representations for property prediction. *Journal of Chemical Information and Modeling*, 59(8):3370–3388, 2019.

X. Yang, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining. *CoRR*, abs/1103.2405, 2011.

Z. Yang, W. Cohen, and R. Salakhutdinov. Revisiting semi-supervised learning with graph embeddings. In *ICML*, 2016.

Y. Yao and L. B. Holder. Scalable classification for large dynamic networks. In *IEEE International Conference on Big Data*, 2015.

R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *KDD*, 2018a.

R. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec. Hierarchical graph representation learning with differentiable pooling. In *NeurIPS*, 2018b.

R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec. GNNExplainer: Generating explanations for graph neural networks. In *NeurIPS*, 2019.

J. You, R. Ying, X. Ren, W. L. Hamilton, and J. Leskovec. GraphRNN: Generating realistic graphs with deep auto-regressive models. In *ICML*, 2018.

J. You, R. Ying, and J. Leskovec. Position-aware graph neural networks. In *ICML*, 2019.

J. You, R. Ying, and J. Leskovec. Design space for graph neural networks. In *NeurIPS*, 2020.

D. Younger, S. Berger, D. Baker, and E. Klavins. High-throughput characterization of protein-protein interactions by reprogramming yeast mating. *Proceedings of the National Academy of Sciences*, 114(46):12166–12171, 2017.

L. Yu, J. Shen, J. Li, and A. Lerer. Scalable graph neural networks for heterogeneous graphs. *CoRR*, abs/2011.09679, 2020.

O. Zachariadis, N. Satpute, J. Gómez-Luna, and J. Olivares. Accelerating sparse matrix-matrix multiplication with GPU tensor cores. In *CAEE*, 2020.

M. Zaheer, S. Kottur, S. Ravanbhakhsh, B. Póczos, R. Salakhutdinov, and A. J. Smola. Deep sets. In *NIPS*, 2017.

A. Zanfir and C. Sminchisescu. Deep learning of graph matching. In *CVPR*, 2018.

M. Zaslavskiy, F. Bach, and J. P. Vert. A path following algorithm for the graph matching problem. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(12), 2009.

H. Zeng, M. Zhang, Y. Xia, A. Srivastava, R. Kannan, V. Prasanna, L. Jin, A. Malevich, and R. Chen. Deep graph neural networks with shallow subgraph samplers. *CoRR*, abs/2012.01.380, 2020a.

H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna. GraphSAINT: Graph sampling based inductive learning method. In *ICLR*, 2020b.

C. Zhang, D. Song, C. Huang, A. Swami, and N. V. Chawla. Heterogeneous graph neural network. In *WWW*, 2019a.

H. Zhang, S.and Tong. FINAL: fast attributed network alignment. In *SIGKDD*, 2016.

J. Zhang and S. Y. Philip. Multiple anonymized social networks alignment. In *ICDM*, 2015.

M. Zhang and Y. Chen. Weisfeiler-Lehman neural machine for link prediction. In *KDD*, 2017.

M. Zhang and Y. Chen. Link prediction based on graph neural networks. In *NeurIPS*, 2018.

M. Zhang, Z. Cui, M. Neumann, and Y. Chen. An end-to-end deep learning architecture for graph classification. In *AAAI*, 2018.

M. Zhang, P. Li, Y. Xia, K. Wang, and L. Jin. Revisiting graph neural networks for link prediction. *CoRR*, abs/2010.16103, 2020a.

R. Zhang, Y. Zou, and J. Ma. Hyper-SAGNN: A self-attention based graph neural network for hypergraphs. In *ICLR*, 2020b.

W. Zhang, K. Shu, H. Liu, and Y. Wang. Graph neural networks for user identity linkage. *CoRR*, abs/1903.02174, 2019b.

Y. Zhang, A. Prügel-Bennett, and J. Hare. Learning representations of sets through optimized permutations. In *ICLR*, 2019c.

Z. Zhang and W. S. Lee. Deep graphical feature learning for the feature matching problem. In *ICCV*, 2019.

Z. Zhang, Y. Xiang, L. Wu, B. Xue, and A. Nehorai. KerGM: Kernelized graph matching. In *NeurIPS*, 2019d.

H. Zhao, L. Jiang, J. Jia, P. Torr, and V. Koltun. Point transformer. *CoRR*, abs/2012.09164, 2020.

L. Zhao and L. Akoglu. PairNorm: Tackling oversmoothing in GNNs. In *ICLR*, 2020.

D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis. DistDGL: Distributed graph neural network for training for billion-scale graphs. *CoRR*, abs/2010.05337, 2020a.

D. Zheng, X. Song, C. Ma, Z. Tan, Z. Ye, J. Dong, H. Xiong, Z. Zhang, and G. Karypis. DGL-KE: Training knowledge graph embeddings at scale. *CoRR*, abs/2004.08532, 2020b.

A. Zhou, J. Yang, Y. Gao, T. Quia, Y. Qi, X. Wang, Y. Chen, P. Dai, W. Zhao, and C. Hu. Optimizing memory efficiency of graph neural networks on edge computing platforms. In *RTAS*, 2021.

F. Zhou and F. De la Torre. Factorized graph matching. In *CVPR*, 2016.

K. Zhou, X. Huang, Y. Li, D. Zha, R. Chen, and X. Hu. Towards deeper graph neural networks with differentiable group normalization. In *NeurIPS*, 2020.

J. Zhu, Y. Yan, L. Zhao, M. Heimann, L. Akoglu, and D. Koutra. Beyond homophily in graph neural networks: Current limitations and effective designs. In *NeurIPS*, 2020.

J. Zhu, R. A. Rossi, A. Rao, T. Mai, N. Lipka, N. K. Ahmed, and D. Koutra. Graph neural networks with heterophily. In *AAAI*, 2021a.

J. Y. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *ICCV*, 2017.

Q. Zhu, X. Zhou, J. Wu, J. Tan, and L. Guo. Neighborhood-aware attentional representation for multilingual knowledge graphs. In *IJCAI*, 2019a.

R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou. AliGraph: A comprehensive graph neural network platform. In *KDD*, 2019b.

X. Zhu, Z. Ghahramani, and J. Lafferty. Semi-supervised learning using gaussian fields and harmonic functions. In *ICML*, 2003.

X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *USENIX Symposium on Operating Systems Designand Implementation*, 2016.

Z. Zhu, S. Xu, J. Tang, and M. Qu. GraphVite: A high-performance CPU-GPU hybrid system for node embedding. In *WWW*, 2019c.

Z. Zhu, S. Lui, C. Shi, J. Chen, Z. Zhang, M. Qu, L. P. Xhonneux, M. Xu, X. Yuan, c. Ma, J. Lu, Y. Zhang, R. Lui, and J. Tang. TorchDrug: A powerful and flexible machine learning platform for drug discovery. `https://torchdrug.ai`, 2021b. (last access: August 25, 2022).

M. Zitnik, R. Sosič, M. W. Feldman, and J. Leskovec. Evolution of resilience in protein interactomes across the tree of life. *Proceedings of the National Academy of Sciences*, 116(10):4426–4433, 2019.

D. Zou, Z. Hu, Y. Wang, S. Jiang, Y. Sun, and Q. Gu. Layer-dependent importance sampling for training deep and large graph convolutional networks. In *NeurIPS*, 2019.

X. Zou, Q. Jia, J. Zhang, C. Zhou, H. Yang, and J. Tang. Dimensional reweighting graph convolution networks, 2020.

Abbreviations

AST Abstract Syntax Tree

CNN Convolutional Neural Network

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DFT Density Functional Theory

DGMC Deep Graph Matching Consensus

DNA Dynamic Neighborhood Aggregation

GAS GNNAutoScale

GAT Graph Attention Network

GCN Graph Convolutional Network

GECN Group Equivariant Convolutional Network

GIN Graph Isomorphism Network

GNN Graph Neural Network

GPU Graphics Processing Unit

GRL Graph Representation Learning

GRU Gated Recurrent Unit

HIMP Hierarchical Inter-Message Passing

JK Jumping Knowledge

KG Knowledge Graph

k**-GNN** k-dimensional GNN

k**-WL** k-dimensional Weisfeiler-Lehman

LSTM Long Short-Term Memory

MAG Microsoft Academic Graph

MLP Multi-Layer Perceptron

OGB Open Graph Benchmark

OGB-LSC Open Graph Benchmark Large-Scale Challenge

PNA Principal Neighborhood Aggregation

PyG PyTorch Geometric

PyGAS PyTorch Geometric AutoScale

QAP Quadratic Assignment Problem

ReLU Rectified Linear Unit

RNN Recurrent Neural Network

SplineCNN Spline-Based Convolutional Neural Network

SVD Singular Value Decomposition

SVM Support-Vector Machine

VR-GCN Variance Reduction Graph Convolutional Network

WL Weisfeiler-Lehman

A

Appendix

A.1 OGB Datasets

We introduced the Open Graph Benchmark (OGB) suite (Hu *et al.*, 2020a, 2021b) in Section 5.4. Here, we provide more details of each available dataset in OGB according to task category.

A.1.1 Node Property Prediction Datasets

We provide six datasets in OGB, adopted from diverse application domains, for predicting the properties of individual nodes:

Specifically, `products` is an Amazon products co-purchasing network originally developed by Chiang *et al.* (2019), in which nodes represent products sold on Amazon, and edges between two products indicate that the products were purchased together. The task is to predict the category of a given product. The graph, target labels, and node features are generated by following Chiang *et al.* (2019), where node features are dimensionality-reduced bag-of-words of product descriptions. While we inherit the original graph representation introduced in Chiang *et al.* (2019), we introduce a more challenging and realistic dataset split than conventional random splitting. In particular, we use the *sales ranking* to split nodes into training, validation and test sets. That is, we sort the products according to their sales ranking and use the top 8% for training, next top 2% for validation, and the rest for testing. This split closely matches the real-world application where manual labeling is prioritized to important nodes in the network, and machine learning models are subsequently used to make predictions on less important ones (Hu *et al.*, 2020a). In particular, the sales ranking split emphasizes the importance of out-of-distribution generalization in real-world graph machine learning tasks, as indicated by the t-SNE visualization (van der Maaten & Hinton, 2008) of training, validation and test nodes in Figure A.1. Here, we can observe that test node representations may be inherently different from training node representations, which is not the case in conventional random splits (Hu *et al.*, 2020a).

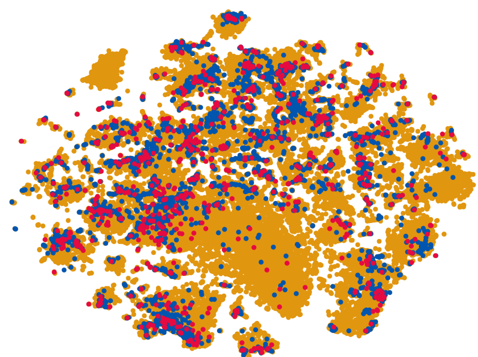


Figure A.1: T-SNE visualization of ■ training, ■ validation and ■ test nodes in the `products` **dataset.** It can be seen that test node representations may be inherently different from training node representations (Hu *et al.*, 2020a).

The `arxiv`, `papers100M`, `mag` and `mag240M` datasets are extracted from the Microsoft Academic Graph (MAG) (Wang *et al.*, 2020), utilizing different scales, tasks, and include both homogeneous and heterogeneous graph information. Specifically, `arxiv` is a paper citation network between all Computer Science ARXIV papers indexed by MAG. Each node is an ARXIV paper and each edge indicates that one paper cites another one. Each paper comes with a 128-dimensional feature vector obtained by averaging their WORD2VEC embeddings (Mikolov *et al.*, 2013) of words in its title and abstract. The task is to predict the subject areas of ARXIV papers, which were manually labeled by the paper's authors and ARXIV moderators. The `papers100M` dataset is created in a similar fashion, but contains all of the 111 million papers indexed by MAG (Wang *et al.*, 2020). Among its node set, approximately 1.5 million of them are ARXIV papers, each of which is manually labeled with one of ARXIV's subject areas. With the volume of scientific publications doubling every 12 years over the past century (Dong *et al.*, 2017b), it is practically important to automatically classify each paper's areas and topics. In contrast to previously-used small-scale citation networks that are split randomly (Sen *et al.*, 2008; Yang *et al.*, 2016), we consider a realistic data split based on the publication dates of papers. This amounts to the general real-world setting in which machine learning models are trained on existing papers which are then used to predict the subject areas of newly-published papers. An accurate automatic predictor of papers' subject categories not only reduces the significant burden of manual labeling, but can also be used to classify the vast number of non-ARXIV papers, thereby allowing better search and organization of academic papers. Specifically, we propose to train on papers published until 2017, validate on those published in 2018, and test on those published since 2019 (Hu *et al.*, 2020a).

On the other hand, both the `mag` and `mag240M` datasets represent the MAG as a heterogeneous rather than homogeneous network, either as a subset or as its full set, respectively. These networks contain additional information than what is solely provided in a homogeneous paper citation network (Hu *et al.*, 2020a, 2021b). In particular, it contains three node types ("paper", "author" and "institution") with different connections between nodes of different type, *e.g.*, an author "writes" a paper or a paper "cites" a paper. This accounts to the real-world setting in which graphs are typically heterogeneous, and emphasizes the development of novel heterogeneous Graph Neu-

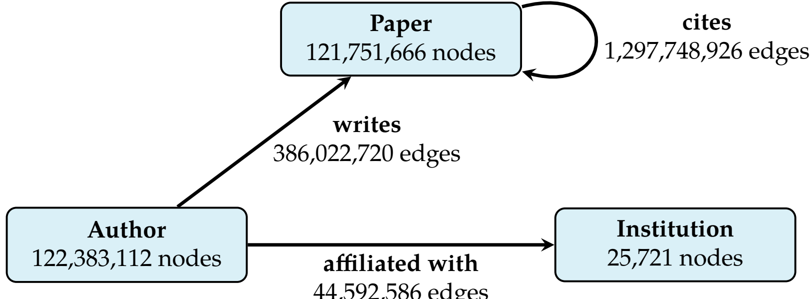


Figure A.2: The schema diagram of the heterogeneous `mag240M` dataset. `mag240M` is a large-scale graph network, consisting of three different node types ("paper", "author", "institution") and three different edge types ("cites", "writes", "affiliated with") (Hu *et al.*, 2021b).

ral Networks (Schlichtkrull *et al.*, 2018; Zhang *et al.*, 2019a; Wang *et al.*, 2019d; Yu *et al.*, 2020; Hu *et al.*, 2020c). For `mag`, the task is to predict the venue (conference or journal) of each paper. This is of practical interest as some manuscript's venue information is unknown or missing in MAG (Hu *et al.*, 2020a). For `mag240M`, the task is once again to automatically annotate a paper's topic, *i.e.* predicting the primary subject area of each arXiv paper. Up to this date, `mag240M` is the largest publicly available dataset, containing around 121M academic papers written by 122M authors who are affiliated with 26K institutions. Among these papers, there exists 1.3B citation links captured by MAG (Hu *et al.*, 2021b). An illustrative overview of this dataset is provided in Figure A.2. In order to learn more fine-grained embeddings, `mag240M` represents titles and abstracts of papers as a 768-dimensional vector generated by a RoBERTa sentence encoder (Liu *et al.*, 2019a; Reimers & Gurevych, 2019). With this, input node features and graph structure occupy around 349GB and 26GB of disk space, respectively. The `mag240M` dataset was part of the Open Graph Benchmark Large-Scale Challenge (OGB-LSC) (Hu *et al.*, 2021b).[1]

The `proteins` dataset denotes a protein-protein association network in which nodes represent proteins, and edges indicate different types of biologically meaningful associations between proteins, *e.g.*, physical interactions, co-expression or homology (Szklarczyk *et al.*, 2019; Consortium, 2018; Hu *et al.*, 2020a). All edges come with 8-dimensional features, where each dimension represents the approximate confidence of a single association type. The task is to predict the presence of protein functions, leading to 112 kinds of labels to predict. For data splitting, we split the protein nodes into training, validation and tests according to the species which the proteins come from. This enables the evaluation of the generalization performance of the model *across* different species. Notably, `proteins` does not provide any input node features, but has edge features on more than 30 million edges. This emphasizes the development of sophisticated Graph Neural Network (GNN) solutions that can utilize edge features rather than node features in a sophisticated way (Hu *et al.*, 2020a).

[1]OGB-LSC website: `https://ogb.stanford.edu/kddcup2021` (last access: August 25, 2022)

A.1.2 Link Property Prediction Datasets

We provide seven link property prediction datasets in OGB, adopted from diverse application domains, including biological, academic datasets as well as Knowledge Graphs (KGs). The different datasets are highly diverse in their graph structure, *cf.* Table 5.3. For example, the biological networks are much denser than the academic networks and the KGs (Hu *et al.*, 2020a).

In particular, we provide the two biological datasets `ppa` and `ddi`, which represent protein-protein association and drug-drug interaction networks, respectively (Szklarczyk *et al.*, 2019; Wishart *et al.*, 2018). The `ppa` dataset is an undirected, unweighted graph, in which nodes represent proteins from 58 different species, and edges indicate biologically meaningful associations between proteins, *e.g.*, physical interactions, co-expression, homology or genomic neighborhood (Szklarczyk *et al.*, 2019; Hu *et al.*, 2020a). Each node contains a 58-dimensional one-hot encoded feature vector that indicates the species that the corresponding protein comes from. The task is to predict new association edges given existing associations. We provide a biological throughput split of the edges into training/validation/test edges. Training edges are protein associations that are measured experimentally by a high-throughput technology (*e.g.*, cost-effective, automated experiments that make large scale repetition feasible (Macarron *et al.*, 2011; Bajorath, 2002; Younger *et al.*, 2017)) or are obtained computationally (*e.g.*, via text-mining). In contrast, validation and test edges contain protein associations that can only be measured by low-throughput, resource-intensive experiments performed in laboratories. In particular, the goal is to predict a particular type of protein association, *e.g.*, physical protein-protein interaction, from other types of protein associations (*e.g.*, co-expression, homology, or genomic neighborhood) that can be more easily measured and are known to correlate with associations that we are interested in. Similarly, the `ddi` dataset also describes an undirected and unweighted graph, representing drug-drug interactions (Wishart *et al.*, 2018). Each node represents an FDA-approved or experimental drug. Edges represent interactions between drugs and can be interpreted as a phenomenon where the joint effect of taking the two drugs together is considerably different from the expected effect in which drugs act independently of each other. The task is to predict unknown drug-drug interactions given information on already known drug-drug interactions, in which true drug interactions should be ranked higher than non-interacting drug pairs (Hu *et al.*, 2020a). We develop a protein-target split, meaning that we split drug edges according to what proteins those drugs target in the body. As a result, the test set consists of drugs that predominantly bind to different proteins from drugs in the train and validation sets. This means that drugs in the test set work differently in the body, and have a rather different biological mechanism of action than drugs in the train and validation sets. The protein-target split thus enables us to evaluate to what extent the models can generate practically useful predictions (Guney, 2017), *i.e.* non-trivial predictions that are not hindered by the assumption that there exist already known and very similar medications available for training (Hu *et al.*, 2020a).

Furthermore, the two academic datasets `collab` and `citation` describe author collaboration and paper citation networks, extracted from MAG (Wang *et al.*, 2020). The `collab` dataset is an undirected graph, representing a subset of the collaboration network between authors. Each node represents an author and edges indicate the collaboration between authors. All nodes come with 128-dimensional features, obtained by averaging the word embeddings of papers that are published by the authors. All

edges are associated with two types of meta-information: the year and the number of collaborations in that year. As such, the graph can be viewed as a dynamic multi-graph as there exists multiple edges between two authors in case they have collaborated in more than one year (Hu *et al.*, 2020a). The task is to predict the author collaboration relationships in a particular year given past collaborations. As the task is inherently temporal, it is natural for models to incorporate the most recent edge information to make prediction. Similarly, `citation` represents a citation network indexed by MAG, where an edge indicates that one paper cites another. The task is to predict missing citations given existing citations. Specifically, for each source paper, two of its references are randomly dropped, and we would like the model to rank the missing two references higher than negative reference candidates. This simulates the practical use-case of where a user is writing a new paper and has already cited several existing papers, but wants to be recommended additional references (Hu *et al.*, 2020a). Both datasets are splitted according to time, in order to simulate a realistic application in collaboration and citation recommendation, respectively. Specifically, we use the relations until 2017 as training edges, those in 2018 as validation edges, and those in 2019 as test edges (Hu *et al.*, 2020a).

In addition, we provide three KGs named `biokg`, `wikikg` and `wikikg90M`, utilizing different tasks and scales (Hu *et al.*, 2020a, 2021b). Knowledge Graphs are known to provide rich structured information about many entities, aiding a variety of knowledge-intensive down-stream applications such as information retrieval, question answering (Singhal, 2012), and recommender systems (Guo *et al.*, 2020). However, these large KGs are known to be far from complete (Min *et al.*, 2013), missing many relational information between entities. Using machine learning methods to automatically impute missing triplets (head, relation, tail) significantly reduces the manual curation of knowledge and provides a more comprehensive KG, which in turn improves the aforementioned down-stream applications. As such, the general task in KGs is to predict new triplets given the already existing triplets.

In particular, the `biokg` dataset represents a KG curated from a large number of biomedical data repositories, containing 5 types of entities: diseases (10,687 nodes), proteins (17,499), drugs (10,533 nodes), side effects (9,969 nodes), and protein functions (45,085 nodes). There are 51 types of directed relations connecting two types of entities, including 39 kinds of drug-drug interactions, 8 kinds of protein-protein interactions, as well as drug-protein, drug-side effect, drug-protein, function-function relations (Hu *et al.*, 2020a). All relations are modeled as directed edges, among which the relations connecting the same entity types (*e.g.*, protein-protein, drug-drug, function-function) are always symmetric, *i.e.* the edges are bi-directional. This dataset is relevant to both biomedical and fundamental machine learning research. On the biomedical side, the dataset allows us to get better insights into human biology and generate predictions that can guide down-stream biomedical research. On the fundamental machine learning side, the dataset presents challenges in handling a noisy, incomplete KG with possible contradictory observations. This is because the `biokg` dataset involves heterogeneous interactions that span from the molecular scale (*e.g.*, protein-protein interactions within a cell) to whole populations (*e.g.*, reports of unwanted side effects experienced by patients in a particular country). Further, triplets in the KG come from sources with a variety of confidence levels, including experimental readouts, human-curated annotations, and automatically extracted metadata. Note that this dataset is the only dataset for which we adapt a random split. While splitting the triplets according to time is an attractive alternative, we note that it is incredibly

challenging to obtain accurate information as to when individual experiments and observations underlying the triplets were made (Hu *et al.*, 2020a).

Furthermore, the wikikg and wikikg90M datasets describe KGs extracted from the Wikidata knowledge base (Vrandečić & Krötzsch, 2014a), and contain sets of triplets capturing the different types of relations between entities in the world, *e.g.*, "*Hinton* $\xrightarrow{citizen\ of}$ *Canada*". Each triplet (head, relation, tail) in these datasets represent an Wikidata claim, where the head and tail denote Wikidata items, and the relation represents the Wikidata predicate. Specifically, we downloaded Wikidata at three different timestamps for training, validation, and testing, respectively. This temporal split simulates the task automatically impute missing triplets that are not yet present in the current KG. Accurate imputation models can then be readily deployed on the Wikidata to improve its coverage (Hu *et al.*, 2021b). While wikikg and wikikg90M are curated from the same data source, they highly differ in scale. In particular, wikikg90M bundles the *entire* Wikidata knowledge base, leading to 87,143,637 entities, 1,315 relations, and 504,220,369 triplets in total. Furthermore, wikikg90M additionally contains rich feature information for both entities and relations (Hu *et al.*, 2021b). Specifically, each entity/relation in Wikidata is associated with a title and a short description, *e.g.*, one entity is associated with the title "Geoffrey Hinton" and the description "computer scientist and psychologist". We provide those text representations via embeddings obtained from a RoBERTa sentence encoder (Liu *et al.*, 2019a; Reimers & Gurevych, 2019). With this, wikikg90M is by far the largest KG available, and the only one containing rich feature information in addition. As such, this requests not only new machine learning models that can make use of rich feature information, but also provides huge challenges in terms of scalability. This has made wikikg90M an excellent candidate to be part of the OGB-LSC (Hu *et al.*, 2021b).

A.1.3 Graph Property Prediction Datasets

We provide five datasets in OGB for predicting the properties of entire graphs or subgraphs, adopted from three distinct application domains.

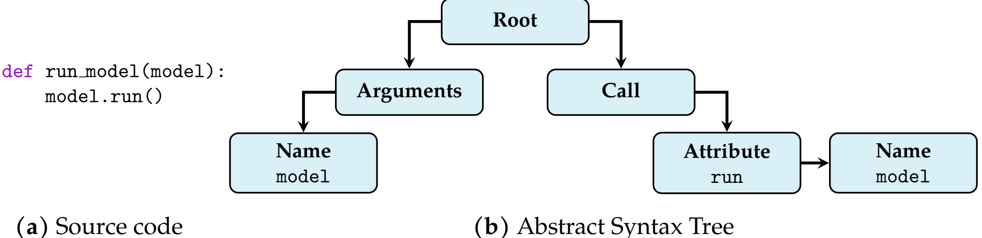
In particular, we provide the three datasets molhiv, molpcba and pcqm4M for tackling the task of molecular property prediction (Hu *et al.*, 2020a, 2021b). As those datasets are provided as pure SMILES strings (Weininger, 1988), we pre-process them via RDKit (Landrum, 2016) into a unified graph representation. Specifically, each graph represents a molecule, in which nodes denote atoms, and edges describe chemical bonds. Input node features are given as a nine-dimensional vector, containing its atomic number, as well as other useful features such as chirality, formal charge, or whether the atom is part of a ring. Input edge features are three-dimensional, containing its bond type, bond stereochemistry as well as the information whether the bond is conjugated. While the above features are not necessarily needed to uniquely identify molecules, they have been shown to boost the performance of graph-based machine learning models (Hu *et al.*, 2020a,b). Furthermore, this standardized feature representation encourages a fair comparison of different models in homogeneous evaluation scenarios. Besides the standardized graph representation, we also propose to leverage a standardized and consistent splitting procedure. In particular, we propose to adapt the *scaffold splitting* procedure for all molecular graph learning tasks, which splits the molecules based on their two-dimensional structural frameworks (Wu *et al.*, 2018; Yang *et al.*, 2019; Hu *et al.*, 2020b; Ishiguro *et al.*, 2019; Rong *et al.*, 2020a). The

scaffold splitting attempts to separate structurally different molecules into different subsets, which provides a more realistic estimate of model performance in prospective experimental settings than conventional random splitting (Hu *et al.*, 2020a).

The `molhiv` and `molpcba` are adopted from MoleculeNet (Wu *et al.*, 2018), which are among the largest ones of the 12 totally provided datasets, *i.e.* containing 41,127 and 437,929 graphs, respectively. Here, the task is to predict certain molecular properties as accurately as possible. Specifically, in `molhiv`, the task is to infer whether a molecule inhibits HIV virus replication or not. The `molpcba` dataset has collected a variety of biological activities of small molecules generated by high-throughput screening (Wang *et al.*, 2018a).

In contrast, the `pcqm4M` molecular graph dataset was constructed by ourselves (Hu *et al.*, 2021b). The `pcqm4M` dataset is a quantum chemistry dataset based on the PubChemQC project (Nakata, 2015; Nakata & Shimazaki, 2017), for which we define the meaningful task of predicting the Density Functional Theory (DFT)-calculated HOMO-LUMO energy gap of molecules, given their molecular graph representation. The HOMO-LUMO gap is one of the most practically-relevant quantum chemical properties of molecules since it is related to reactivity, photoexcitation and charge transport (Griffith & Orgel, 1957). While DFT is a powerful and widely-used quantum physics calculation that can accurately predict various molecular properties (Helgaker *et al.*, 2014), it is very time-consuming to compute. Even for small molecules, it can take up to several hours to obtain. Therefore, using fast and accurate machine learning models to approximate DFT-calculations enables diverse down-stream applications, such as property prediction for organic photovoltaic devices (Cao & Xue, 2014) and structure-based virtual screening for drug discovery (Ferreira *et al.*, 2015). Furthermore, predicting the quantum chemical property only from molecular graphs without their 3D equilibrium structures is practically favorable and more challenging (Ramakrishnan *et al.*, 2014; Gilmer *et al.*, 2017; Schütt *et al.*, 2017; Klicpera *et al.*, 2020b). This is because obtaining 3D equilibrium structures requires DFT-based geometry optimization as well. Besides its novelty, the `pcqm4M` dataset is one of the largest molecular graph learning datasets created. Specifically, it contains 3,803,453 molecules, with 80% of them available to train data-hungry machine learning models. This has made the `pcqm4M` dataset the perfect fit to be part of the OGB-LSC (Hu *et al.*, 2021b).

The `ppa` dataset contains a set of undirected protein association neighborhoods extracted from the protein-protein association networks of 1,581 different species (Szklarczyk *et al.*, 2019; Hu *et al.*, 2020a) that cover 37 broad taxonomic groups, *e.g.*, mammals, bacterial families and archaeans (Hug *et al.*, 2016). Each graph represents the protein association neighborhood of a protein, containing up to 100 proteins from each species. Similar to the `proteins` dataset (Appendix A.1.1), nodes in each protein association graph represent proteins, and edges indicate biologically meaningful associations between proteins. In comparison to the other graph datasets, the biological subgraphs in `ppa` have a much larger number of nodes per graph, as well as much denser and clustered graph structures, resulting in a large average node degree, large average clustering coefficient and large graph diameter (Hu *et al.*, 2020a). The edges are associated with a 7-dimensional feature vector, where each value represents the approximate confidence of a particular type of protein-protein association. The task of `ppa` is to predict the taxonomic group from which the graph originates from. The ability to successfully tackle this problem has implications for understanding the evolution of protein complexes across species (De Juan *et al.*, 2013), the rewiring of



(**a**) Source code (**b**) Abstract Syntax Tree

Figure A.3: An example input graph (b) in the `code` dataset, obtained from Python source code (a). In our AST, the root node always defines the main function definition, and the goal is to predict its tokenized function name, *e.g.*, {`run`, `model`}. Its children contain the input arguments, as well as its individual commands (Hu *et al.*, 2020a).

protein interactions over time (Sharan *et al.*, 2005; Zitnik *et al.*, 2019), the discovery of functional associations between genes even for otherwise rarely-studied organisms (Cowen *et al.*, 2017), and can give us insights into key bioinformatics tasks such as biological network alignment (Malod-Dognin *et al.*, 2017; Hu *et al.*, 2020a).

The `code` dataset is a collection of Abstract Syntax Trees (ASTs) obtained from approximately 450K Python functions, which were extracted from a total of 13,587 different repositories across the most popular projects on GitHub according to the number of stars and forks (Husain *et al.*, 2019; Hu *et al.*, 2020a). Given the input arguments and body of a Python method represented by an AST, the task is to predict its method name as a set of sub-tokens. This task is often referred to as "code summarization" (Allamanis *et al.*, 2016, 2017; Alon *et al.*, 2018, 2019), because the model is trained to find a succinct and precise description (*i.e.* the method name chosen by the developer) for a complete logical unit (*i.e.* the method body). Code summarization is a representative task in the field of machine learning for code not only because of its straightforward adoption in developer tools, but also because it is a proxy measure for assessing how well a model captures code semantics (Allamanis *et al.*, 2018). The graph of a method is directly given by its AST representation, *cf.* Figure A.3. In our AST representation, the root node always corresponds to the main function definition. Its children contain the input arguments, as well as its individual commands. Commands are further associated with meaningful node features that denote their types (from a pool of 97 types) and attributes (such as its variable name given by a fixed-sized vocabulary, or the current depth in the AST). We also mask out attributes of recursive function definitions to avoid data leakage (Hu *et al.*, 2020a). As splitting procedure, we split the dataset *by project*, where the ASTs for the train sets are obtained from GitHub projects that do not appear in the validation and test sets. This split respects the practical scenario of training a model on a large collection of source code, which is then used to predict method names in a separate codebase. The project split stress-tests the model's ability to capture code semantics, and avoids the effects of memorization of specific idiosyncrasies in the training projects (such as the naming conventions and the coding style of a specific developer) (Allamanis, 2019; Hu *et al.*, 2020a).