Université de Montréal

Taxonomy of Datasets in Graph Learning : A Data-Driven Approach to Improve GNN Benchmarking

par

Semih Cantürk

Département de mathématiques et de statistique
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

1er décembre 2022

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Taxonomy of Datasets in Graph Learning : A Data-Driven Approach to Improve GNN Benchmarking

présenté par

Semih Cantürk

a été évalué par un jury composé des personnes suivantes :

Irina Rish
(président-rapporteur)

Guy Wolf
(directeur de recherche)

Eugene Belilovsky
(membre du jury)

Résumé

L'apprentissage profond sur les graphes a atteint des niveaux de succès sans précédent ces dernières années grâce aux réseaux de neurones de graphes (GNN), des architectures de réseaux de neurones spécialisées qui ont sans équivoque surpassé les approches antérieurs d'apprentissage définies sur des graphes. Les GNN étendent le succès des réseaux de neurones aux données structurées en graphes en tenant compte de leur géométrie intrinsèque. Bien que des recherches approfondies aient été effectuées sur le développement de GNN avec des performances supérieures à celles des modèles références d'apprentissage de représentation graphique, les procédures d'analyse comparative actuelles sont insuffisantes pour fournir des évaluations justes et efficaces des modèles GNN. Le problème peut-être le plus répandu et en même temps le moins compris en ce qui concerne l'analyse comparative des graphiques est la *couverture de domaine* : malgré le nombre croissant d'ensembles de données graphiques disponibles, la plupart d'entre eux ne fournissent pas d'informations supplémentaires et au contraire renforcent les biais potentiellement nuisibles dans le développement d'un modèle GNN. Ce problème provient d'un manque de compréhension en ce qui concerne les aspects d'un modèle donné qui sont sondés par les ensembles de données de graphes. Par exemple, dans quelle mesure testent-ils la capacité d'un modèle à tirer parti de la structure du graphe par rapport aux fonctionnalités des nœuds ? Ici, nous développons une approche fondée sur des principes pour taxonomiser les ensembles de données d'analyse comparative selon un *profil de sensibilité* qui est basé sur la quantité de changement de performance du GNN en raison d'une collection de perturbations graphiques. Notre analyse basée sur les données permet de mieux comprendre quelles caractéristiques des données de référence sont exploitées par les GNN. Par conséquent, notre taxonomie peut aider à la sélection et au développement de repères graphiques adéquats et à une évaluation mieux informée des futures méthodes GNN. Enfin, notre approche et notre implémentation dans le package `GTaxoGym` [1] sont extensibles à plusieurs types de tâches de prédiction de graphes et à des futurs ensembles de données.

1. `https://github.com/G-Taxonomy-Workgroup/GTaxoGym`

Mots-clés : Apprentissage automatique, Apprentissage profond, Apprentissage automatique sur graphes, Réseaux de neurones en graphes, Réseau de neurones artificiels, Benchmarking, Jeux de données, Taxonomie, Théorie des graphes, Traitement du signal graphique

Abstract

Deep learning on graphs has attained unprecedented levels of success in recent years thanks to Graph Neural Networks (GNNs), specialized neural network architectures that have unequivocally surpassed prior graph learning approaches. GNNs extend the success of neural networks to graph-structured data by accounting for their intrinsic geometry. While extensive research has been done on developing GNNs with superior performance according to a collection of graph representation learning benchmarks, current benchmarking procedures are insufficient to provide fair and effective evaluations of GNN models. Perhaps the most prevalent and at the same time least understood problem with respect to graph benchmarking is *domain coverage*: Despite the growing number of available graph datasets, most of them do not provide additional insights and on the contrary reinforce potentially harmful biases in GNN model development. This problem stems from a lack of understanding with respect to what aspects of a given model are probed by graph datasets. For example, to what extent do they test the ability of a model to leverage graph structure vs. node features? Here, we develop a principled approach to taxonomize benchmarking datasets according to a *sensitivity profile* that is based on how much GNN performance changes due to a collection of graph perturbations. Our data-driven analysis provides a deeper understanding of which benchmarking data characteristics are leveraged by GNNs. Consequently, our taxonomy can aid in selection and development of adequate graph benchmarks, and better informed evaluation of future GNN methods. Finally, our approach and implementation in the `GTaxoGym` package [2] are extendable to multiple graph prediction task types and future datasets.

Keywords: Machine learning, Deep learning, Graph representation learning, Graph neural networks, Neural networks, Benchmarking, Datasets, Taxonomy, Graph theory, Graph signal processing

2. `https://github.com/G-Taxonomy-Workgroup/GTaxoGym`

Contents

List of Tables

List of Figures

List of acronyms and abbreviations

ANN	Artificial Neural Network
BA	Barabási-Albert graph generation model
BoW	Bag-of-words
CNN	Convolutional Neural Network
DAG	Directed Acyclic Graph
DCNN	Diffusion-Convolutional Neural Network Atwood and Towsley [3]
DFT	Discrete Fourier Transform
DTFT	Discrete-Time Fourier Transform
DTMC	Discrete-Time Markov Chains
ER	Erdös-Rényi graph generation model
FNN	Feedforward Neural Network

GCN	Graph Convolutional Network
GIN	Graph Isomorphism Network Xu et al. [159]
GNN	Graph Neural Network
GRL	Graph Representation Learning
GSP	Graph Signal Processing
KKT conditions	Karush–Kuhn–Tucker conditions
LGCN	Learnable Graph Convolutional Network Gao et al. [45]
MPNN	Message-Passing Neural Network
MLP	Multilayer Perceptron
NLP	Natural Language Processing
NPI	Nondeterministic Polynomial (NP) time-Intermediate
PE	Positional Encoding
QP	Quadratic Programming

RBF	Radial Basis Function
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Networks
RvNN	Recursive Neural Network
SBM	Stochastic Block (graph generation) Model
SOTA	State-of-the-art
SVM	Support Vector Machine
WL	Weisfeiler-Lehman algorithms/tests
WS	Watz-Strogatz graph generation model

Acknowledgements

It has certainly been an interesting MSc journey, to say the least. Of the last two years, I ended up being able to share my work environment with my colleagues and fellow researchers for only about two weeks, the rest fallen victim to COVID restrictions, logistical nightmares and a seven-hour time difference. Yet here we are, thanks to the many people whose help and company I've sought along the way. This acknowledgements will inevitably be incomplete, and I apologize in advance to any who I may have ignored in these brief words.

This work is based on a research project two years in the making, so I would like to start by thanking those without whom this thesis would not exist. First, many thanks to my MSc supervisor Guy Wolf, who picked out my email from his probably very large pile and offered me a position in his lab. Thanks for your advice, abundant supply of research directions and ideas, and the freedom you gave me to pursue my interests simultaneously. Here's to an even better PhD!

Many thanks to Ladislav Rampášek and Renming Liu, who have acted both as mentors, colleagues and idea-generation machines of intimidating prowess; this thesis would not be possible without your efforts. Working with you has been a privilege, and I sincerely hope our paths cross many times over as I still have so much to learn from you.

One upside of being physically separate from my work environment meant I got to spent more time with my family and friends, possibly to the point of vexing them at times. To my parents Tubiş and Ceco, and my brother Dero: I'm grateful that you've been here for and with me throughout this journey. Thank you to my grandparents Leyla and Tunç, for thinking of me when I don't. To my partner Gül, who has been an advisor, sounding board and a phenomenal support all along. To my friends, who have taken up the responsibility of reminding me there are other joys to life than work. To Valerie, who has taken care of me in Montréal like I'm her own grandson. Thank you all for the unconditional love and support, and also for putting up with me; I love you all. Last but not least, many thanks to my dear Minu (Figure 2.1a), my silent companion through all of this, to whom I dedicate this work to even though she doesn't care.

Many thanks to my colleagues at Zetane, who are not just amazing coworkers, but ones that are smart, curious and supportive. Special thanks to Guillaume, Patrick and Jon, who

not just accepted but encouraged my decision to pursue my academic ambitions alongside Zetane. Your support has not gone unnoticed, and I appreciate it immensely.

Finally, this work has been made possible thanks to the valuable contributions of my coauthors, who I'd also like to thank here: Frederik Wenkel, Sarah McGuire, Elena Wang, Anna Little, Leslie O'Bray, Michael Perlmutter, Bastian Rieck and Matthew Hirn.

Introduction

Machine learning on graphs, commonly referred to as graph representation learning (GRL), has seen rapid development in recent years [59]. Originally inspired by the success of convolutional neural networks in regular Euclidean domains, thanks to their ability to leverage data-intrinsic geometries, classical graph neural network (GNN) models [27, 79, 148] extend those principles to irregular graph domain. Further advances in the field have led to a wide selection of complex and powerful GNN architectures. Some models are provably more expressive than others [159, 99], can leverage multi-resolution views of graphs [96], or can account for implicit symmetries in graph data [15]. Comprehensive surveys of graph neural networks can be found in Bronstein et al. [14], Wu et al. [156], Zhou et al. [165].

Most graph-structured data encode information in two parts: *graph structures* and *node features*. The structure of each graph represents relationships (i.e., edges) between different nodes, while the node features that accompany this structure represent quantities of interest at each individual node. For example, in citation networks, nodes represent papers and edges represent citations between the papers. On such networks, node features often capture the presence or absence of certain keywords in each paper, encoded in binary feature vectors. In graphs modeling social networks, each node represents a user, and the corresponding node features often include user statistics like gender, age, or binary encodings of personal interests.

Intuitively, the power of GNNs lies in relating local node-feature information to global graph structure information, typically achieved by applying a cascade of feature *aggregation* and *transformation* steps. In aggregation steps, information is exchanged between neighboring nodes, while transformation steps apply a (multi-layer) perceptron to feature vectors of each node individually. Such architectures are commonly referred to as *Message Passing Neural Networks (MPNN)* [51].

Historically, GNN methods have been evaluated on a small collection of datasets [100], many of which originated from the development of graph kernels. The limited quantity, size and variety of these datasets have rendered them insufficient to serve as distinguishing benchmarks [34, 106]. Therefore, recent work has focused on compiling a set of large(r) benchmarking datasets across diverse graph domains [34, 68]. Despite these efforts and the

introduction of new datasets, it is still not well understood what aspects of a dataset most influence the performance of GNNs. Which is more important, the geometric structure of the graph or node features? Are long-range interactions crucial, or are short-range interactions sufficient for most tasks?

This lack of understanding in dataset properties make it difficult to determine subsets of graph datasets that are able to statistically separate GNN model performance despite the steady increase in the number of available datasets. This phenomenon stems from redundancies in the properties tested for even when an array of datasets are used, as seemingly different datasets may be employing similar pathways for the propagation of information, leading to quickly diminishing marginal returns as the number of datasets used for benchmarking increases. In turn, the small subset of datasets used for benchmarking have stayed the same for the most part, and have resulted in impaired benchmarking practices in the field of graph learning [106].

The goal of this work is to propose and apply a framework through which we can methodologically define and evaluate the characteristics of benchmarking datasets in terms of their reliance on particular types of information encoding and propagation. Our framework is built around testing empirical transformation sensitivity of graph datasets to gauge *how* task-related information is encoded in them. We believe that our methodology and the resulting taxonomy can alleviate the problems mentioned above by acting both as a tool to understand existing and future graph datasets and models better, and as a guide to aid in the selection of benchmarking datasets that sufficiently express the variation and complexity of real-world derived graph data.

The rest of the paper is structured as follows: In the first chapter, we introduce relevant concepts in machine learning and graph theory. We follow this a historical perspective of machine learning on graphs, where we delve deeper into several branches of work that have inspired GNNs, the current dominant paradigm in geometric deep learning.

In the second chapter, we survey an array of GNN models from a unified view of *neural message passing*: Starting from the original GNN algorithm [53, 126], we motivate the different approaches that draw from spectral and spatial convolutions, attention mechanisms and graph isomorphism and serve as common benchmarks today. Understanding how distinct GNNs operate is fundamental in order to motivate benchmarking: Even though this work benchmarks graph datasets, the end goal is to arrive at a reliable framework to benchmark and compare GNN models.

This brings us to the third chapter, which aims for a comprehensive analysis of current issues on benchmarking, both in machine learning in general and graph learning specifically. In this chapter, we also motivate the benchmarking process itself as an essential building block of machine learning research, and explain how our dataset taxonomy relates with prior work on graph benchmarking.

The fourth chapter presents the novel research that forms the core of this work. We first argue the need for a dataset benchmarking framework in graph learning, and specifically discuss how our work addresses the pain points in graph benchmarking we have discussed in the previous chapter. We then present our method in detail, and discuss the empirical findings of our research. We conclude the thesis with a summary of our findings, and provide suggestions for future research.

The core research of this thesis, mostly comprising chapter four, has been accepted to the Learning on Graphs (LoG) 2022 conference for a spotlight presentation as a standalone paper under the title "Taxonomy of Benchmarks in Graph Representation Learning", and is to be published in the Proceedings of Machine Learning Research (PMLR) series. As a main author of the paper, my specific contributions to this paper cover problem formulation, design and implementation of our taxonomy framework and experimental pipeline, collation of our results and of course the writing of the article. Naturally, each step also relied on significant contributions from my co-main authors Renming Liu and Ladislav Rampášek as well as our other co-authors; I would therefore like to acknowledge and thank them again before we start.

Chapter 1

Geometric Deep Learning

1.1. An Introduction to Deep Learning

Machine learning aims to build algorithms that "learn" patterns in data through mathematical models without explicit programming. Although established as a standalone field for merely decades, it has close ties and overlaps with statistics and optimization; most of the mathematical tools we use for machine learning predate the field itself by many years: The earliest form of regression analysis in the form of the least-squares method dates to the beginning of 19th century with Legendre and Gauss.

Machine learning incorporates a large variety of learning paradigms, and even more numerous application domains. Most popular learning models can be broadly classified into several groups: regression analysis, decision tree-based learning, support-vector machines (SVM) and deep learning (i.e. deep neural networks). These models are unified from an optimization perspective as the model learning process is equivalent to minimizing a loss function. The immense body of work on machine learning belies its relatively recent establishment as a research field, and we certainly cannot provide a justified coverage within the space of a few pages. More importantly, most of the novel research in this work concerns the learning paradigm of deep learning, and a specific subgroup of deep learning models designed for graph data, namely graph neural networks (GNN). Therefore, we will reserve this introductory section to neural networks and deep learning; we will also introduce SVMs later on and discuss their influence in the development of GNNs.

1.1.1. The perceptron: Building blocks of neural networks

Deep learning is a class of machine learning methods that are based on artificial neural networks (ANN). ANNs are built by stacking layers of artificial neurons; the first artificial neuron was proposed by McCulloch and Pitts [93] in 1943 and implemented by Rosenblatt [117] as a basic linear classifier called the *perceptron* (Figure 1.1). The perceptron is a simple

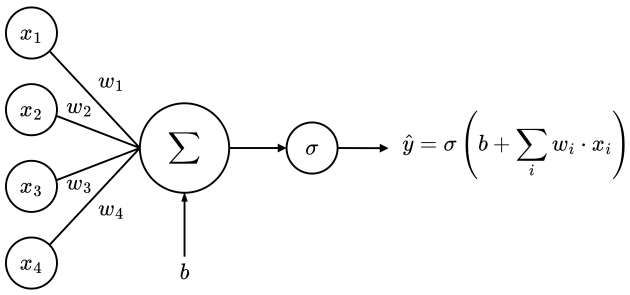


Figure 1.1 – The operations in a perceptron.

computational unit that represents a weighted sum of inputs $\mathbf{x}$ and a bias term b put through a nonlinear activation function σ:

$$f(\mathbf{x}) = \sigma\left(b + \sum_i w_i x_i\right) \tag{1.1.1}$$

We can represent the weighted sum as $\mathbf{w} \cdot \mathbf{x}$ in vectorized form. The original perceptron used the Heaviside step function, meaning any value larger than 0 is mapped to 1 and 0 otherwise, resulting in the following classifier:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases} \tag{1.1.2}$$

Training the perceptron was done via an iterative update rule. Nevertheless, the perceptron is a weak classifier on its own; Minsky and Papert [97] showed that a perceptron cannot learn the XOR function in 1969. Later on, it was realized that the strength of the perceptron was in numbers; stacking multiple layers of perceptrons increased their representational capacity to the point that they are *universal approximators*, meaning they can approximate any continuous function that maps real inputs to real outputs [67]. These stacked layers of perceptrons are called feedforward neural networks (FNN), Figure 1.2 demonstrates how hidden layers are built via stacking neurons. The term FNN is often used interchangeably with multilayer perceptron (MLP), which indicates an FNN where every neuron in one layer is connected to all neurons of the subsequent layer.

MLPs are composed of an input layer, an output layer, and one or more *hidden layers*: A hidden layer is a stack of neurons that take outputs of other neurons as inputs, and whose outputs are inputs to other neurons in turn. In other words, these neurons do not interact with the inputs or outputs directly, hence the term "hidden". A neural network is termed "deep" if it consists of two or more hidden layers. In matrix form, a hidden takes layer inputs $\mathbf{h}^{(k-1)}$ and performs the following operation to produce outputs $\mathbf{h}^{(k)}$:

$$\mathbf{h}^{(k)} = \alpha\left(\mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}\right) \tag{1.1.3}$$

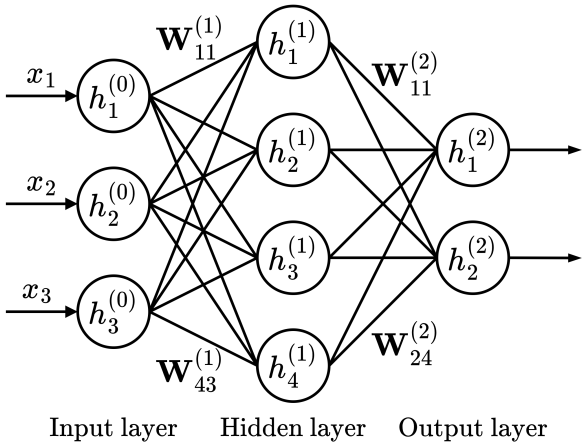


Figure 1.2 – Diagram of a one-hidden-layer MLP, indicating the indices of weights in the weight matrices $\mathbf{W}$ and states at each layer h.

where $\mathbf{W}^{(k)} \in \mathbb{R}^{N^{(k+1)} \times N^{(k)}}$ is a weight matrix, and $N^{(k)}$ represents the number of layer inputs/outputs. The input data $\mathbf{x}$ forms the inputs to the first MLP layer, $\mathbf{x} = \mathbf{h}^{(0)}$.

Typically, MLPs utilize nonlinear activations such as ReLU [44] or sigmoid instead of the Heaviside step function. In particular, output layers may have different activations than the rest of the network. Unlike the perceptron, MLPs can have multiple outputs to represent discrete probability distributions over multiple categories. In such cases, the softmax function is used to normalize the outputs $\hat{\mathbf{y}}$:

$$\hat{\mathbf{y}}_i = \text{softmax}(z)_i = \frac{\exp(\mathbf{z}_i)}{\sum_j \exp(\mathbf{z}_j)}, \quad i = 1, \ldots, n \tag{1.1.4}$$

where $\hat{\mathbf{y}}_i$ indicates the prediction probability for category i.

1.1.2. Training neural networks

Given a set of ground truths $\mathbf{y}$ corresponding to our predictions $\hat{\mathbf{y}}$, we can compute a scalar *loss* through a loss function. The loss is a measure of "error" between the ground truth and prediction, which is objective to minimize through numerical optimization. Developing loss functions to better accommodate a vast range of machine learning tasks is a research subfield of its own; for a recent and comprehensive survey we refer the reader to Wang et al. [150]. For the purposes of this work, the loss function we are concerned with is the *cross-entropy loss*, which is the standard loss for classification tasks in deep learning, not restricted to MLPs:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=0}^{n-1} \mathbf{y}_i \log(\hat{\mathbf{y}}_i) \tag{1.1.5}$$

where n denotes the number of categories (classes) we're predicting amongst. Given a parametrized ML model, the loss can be thought as a function of model parameters $\mathbf{W}$.

This loss is usually not computed per data point in practice, but over a batch of N examples:

$$\mathcal{L}(\mathbf{W}) = \sum_{i=1}^{N} \mathcal{L}_i(\mathbf{W}) \tag{1.1.6}$$

Minimization of the loss function in neural networks is done via gradient descent, a first-order iterative optimization algorithm for differentiable functions. There is an ever-expanding research literature on developing better-performing gradient descent variants (see Ruder [121]), but the fundamental principle remains the same for all. Gradient descent updates a set of parameters (weights) $\mathbf{W}$ iteratively by taking a step (determined by the positive scalar η, the learning rate) in the direction of the negative gradient of the loss function with respect to the parameters:

$$\mathbf{W}' = \mathbf{W} - \eta \cdot \nabla_{\mathbf{W}} \mathcal{L}(\mathbf{W}) \tag{1.1.7}$$

The practical challenge in applying gradient descent to neural networks is the computation of the gradient with respect to all parameters. Backpropagation [122, 87] is the standard dynamic programming algorithm for efficient gradient calculation in deep learning models. An elaborate overview of the modern backpropagation algorithm can be found in Goodfellow et al. [52], but we can summarize it as the composition of two phases:

(1) **Forward Phase:** A batch of inputs are fed to the model, which outputs a set of predictions based on the current model parameters. The loss function $\mathcal{L}$ is calculated for the batch using the predictions and corresponding ground truths.

(2) **Backward Phase:** The gradients for all parameters are calculated dynamically in the backward direction, starting from the output layer and ending at the input layer by recursive application of the chain rule for each parameter. A neural network is composed of sequences of differentiable computation units; the derivative calculation for a parameter (represented by edges in/outflowing to the units in network diagrams, see Figure 1.2) relies on this recursive application of the chain rule based on all computational paths that go through that edge. In MLPs, we have multiple paths going through a parameter, meaning we sum the gradients calculated by each path.

This concludes our (very brief) introduction to deep learning. As mentioned, deep learning research covers so many tools even at the most basic level of building blocks (e.g. loss functions, activations, neuron types, logic gates etc.) that any attempt to be more exhaustive would detract us from our main focus of graph learning. We will revisit additional tools in neural networks in later sections where we discuss families of networks that served as the progenitors of graph neural networks. For a comprehensive review into the theoretical foundations of machine learning and deep learning, Shalev-Shwartz and Ben-David [131] and Goodfellow et al. [52] respectively are well-established resources for the interested reader, though there exist plenty of literature that tackle these domains from different perspectives.

1.2. Introduction to Graph Theory

1.2.1. What is a graph?

Before we dive into geometric deep learning, we need to establish some definitions and introduce several concepts that we will encounter numerous times in this thesis. Let us begin by defining the *simple graph*:

Definition 1.2.1 (Simple graph)**.** *A simple graph* $G = (\mathcal{V}, \mathcal{E})$ *is a collection of vertices* $\mathcal{V}$ *and edges* $\mathcal{E}$*.*

$$\mathcal{E} \subset \{\{a, b\} : a, b \in \mathcal{V}; a \neq b\}$$

where $\{a, b\}$ *denotes that order of vertices does not matter, i.e.* $(a, b) = (b, a)$*.*

Vertices of a graph are also called *nodes*, we will use these terms interchangeably. In many cases, we will want to add weights to our edges, resulting in a *weighted graph*:

Definition 1.2.2 (Weighted graph)**.** *A weighted graph* $G = (\mathcal{V}, \mathcal{E}, w)$ *is a collection of vertices* $\mathcal{V}$ *and edges* $\mathcal{E}$ *with weights* $w : \mathcal{E} \to \mathbb{R}$ *such that* $w(a, b) = w(b, a)$*.*

The simple graph is a special weighted graph where each edge has the same weight, e.g. 1. Another extension is adding directionality to the edges, resulting in the *directed graph*:

Definition 1.2.3 (Directed graph)**.** *A directed graph* $G = (\mathcal{V}, \mathcal{E}, w)$ *is a collection of vertices* $\mathcal{V}$ *and edges* $\mathcal{E}$*, (optionally) with weights* w*.*

$$\mathcal{E} \subset \{(a, b) : a, b \in \mathcal{V}; a \neq b\}$$

where the vertex pairs are ordered, i.e. $(a, b) \neq (b, a)$*.*

The concept of a graph can be further extended further. While they are less commonly used, particularly in the scope of geometric deep learning, some examples to consider are:

Mixed graphs: Graphs with both directed and undirected edges.

Pseudographs: Graphs that permit loops, i.e. $\{a, a\} \in \mathcal{E}; a \in \mathcal{V}$

Multigraphs: Graphs that permit multiple edges between vertex pairs.

Note that most of the focus in this work will be on unweighted, undirected graphs. Therefore, unless weights or directionality is relevant to the definitions or computations at hand, we will refer to graphs of the unweighted, undirected type for clarity.

1.2.2. Representation of data in graphs

Graphs are particularly suitable mathematical tools to represent and leverage *relationships* of objects with each other, in addition to the properties of the individual objects. This is an extension of the representation capabilities of unstructured data (e.g. tabular in the context of machine learning), which cannot account for relationships and relies on the individual objects. This improved representational capacity of graph data makes it a better candidate to analyze many of the complex systems we encounter in the real world.

We have formally defined what a graph *is* in the previous section, but the formal definition of "a set of vertices and edges" is not very useful in itself to model the aforementioned complex systems. Why is that so? Note that we mentioned two types of information encoded in graph data: (a) the properties of the individual objects, and (b) the relationships of objects with each other. A set of vertices and edges only encodes the latter type, we still need to assign properties to the vertices themselves to represent the objects.

We will find it useful to think about graphs with associated functions (also referred to as signals) on graph vertices. A graph signal $\mathbf{x}$ on vertices $\mathcal{V}$ maps each vertex to a real $\mathbb{R}$; $\mathbf{x} : \mathcal{V} \to \mathbb{R}$. More often than not, multiple functions operate on the vertices, encoding a set of properties, commonly referred to as *node features* or *feature set*. One can also see that functions on edges $\mathbf{z} : \mathcal{E} \to \mathbb{R}$ may also exist, similarly encoding edge properties.

In graph data, information is typically encoded in two forms. Firstly, "feature-based" information is encoded on vertices and/or edges as we just covered. In addition, information is implicitly encoded in the graph structure itself, since the existence of an edge between two vertices implies the presence of a certain relationship between the two corresponding objects. We will see that the algorithms that are suitable for graph data will need to leverage these relationships in the coming sections. Let us provide a few real-life examples of graph data for a more intuitive understanding.

— **Molecular representations:** Graphs have been used to represent molecular data in chemistry and biology since early 20th century. In most common uses, vertices represent individual atoms or functional groups while edges represent molecular bonds. Vertex properties, i.e. labels, in such a setting may include atom/group type, molecular weight, free electrons etc., while edge properties may include molecular bond type and/or distance between the vertices.

— **Molecular interactions:** Also within the scope of molecular data, interaction graphs are commonly used in biochemistry to model the interactions of structures. In biological systems, actions of proteins are usually regulated by other proteins, for example. Protein–protein interaction data and cell *interactomes* (set of all molecular interactions in a cell) are therefore suitable candidates for graph representations.

— **Citation networks:** Citation networks are one of the most commonly encountered forms of graph data, due to the ease of generating them from relational databases. Citation networks are drawn from various domains, where nodes may represent articles or authors and an edge shows that one article/author has cited the other.

— **Social networks:** With the increasing popularity and number of social platforms on the web as well as the rapid increase in large-scale data collection and processing in the last two decades, both the availability and the study of social networks have skyrocketed in the 21st century. Graphs are natural candidates for representing

social interactions; vertices commonly represent users, where the space of properties are almost infinite: demographic information, interests, whether they have used certain keywords in their posts etc. Edges represent user relationships, such as follow/friendship status, or interactions between users.

On occasion, we will encounter graphs in which node features indeed do not exist. These types of graphs are mostly found in inductive graph-level tasks where graph properties are predicted based on patterns in graph connectivity; several such datasets are presented in Section 4.3.1. The trivial solution when working with such graphs is to assign an *indicator variable* to each vertex, i.e. indexing the vertices based on an arbitrary ordering, and assigning the one-hot encoding of their indices to each vertex. Other approaches involve deriving node features from structure, such as computing the vertex degree, clustering coefficients and such statistics and assigning a concatenation of these node-level statistics to each node. Both approaches are also covered in our array of datasets in Section 4.3.1.

1.2.3. Learning tasks on graphs

Machine learning tasks are traditionally categorized into several groups based on the learning paradigm: *Supervised* learning aims to predict a target label for a given data point, while *unsupervised* learning focuses on discovering patterns on data without target labels. In addition to supervised tasks, many graph learning tasks exist in a "twilight zone": In transductive node/edge prediction tasks, the standard configuration is to have a single graph with only a portion of nodes possessing target labels; the goal is to predict labels for the rest of the nodes. These tasks in which a single prediction (in our case, a node over the whole graph) involves both labeled and unlabeled components is referred to as *semi-supervised*.

Let us now introduce a broad categorization of learning tasks on graphs, which will assist us in our inspection of graph benchmarking datasets and tasks later on.

— **Node prediction:** As we mentioned, node classification involves predicting a label y_ν for a node $\nu \in \mathcal{V}$ of a graph in the form of classification or regression. Usually, there is a labeled training and unlabeled test set of nodes, which compose the complete node set of the graph: $\mathcal{V}_{\text{train}} \cup \mathcal{V}_{\text{test}} = \mathcal{V}$. Node classification tasks are typically transductive (though we will see inductive node classification datasets as well), i.e. the "dataset" consists of a single large graph. Examples include user property prediction in social networks (e.g. for advertising purposes) or document classification in citation graphs.

— **Edge prediction:** Edge prediction tasks are also mostly transductive, similarly predicting y_ε for an edge $\varepsilon \in \mathcal{E}$. We may want to predict edge properties such as the type of bond in a molecular graph, or interaction strength in a protein-protein interaction graph.

— **Graph clustering:** Also known as community detection, graph clustering is the grouping of graph nodes into clusters based on some property. It is analogous to the traditional unsupervised clustering problems in machine learning. Clustering citation and collaboration graphs to discover subcommunities or fields such as institution or research area are common. Graph-level clustering tasks also exist, where a set of graphs is clustered into subcommunities.

— **Graph-level prediction:** Along with node prediction, graph prediction is the most popular task type in graph representation learning. Graph prediction is inductive; the models are trained and tested on *sets* of graphs. In this sense, graph prediction is very similar to standard supervised learning, where data points are assumed to be independent and identically distributed (i.i.d.), an assumption that does not hold in node or edge prediction tasks. The graphs also tend to be smaller compared to the transductive tasks: Graph prediction tasks are commonplace in bio/cheminformatics where graph representations of molecules are used to infer molecular properties such as solubility, reactivity or toxicity; these graphs are on average much smaller than citation/social network graphs we see in transductive tasks where graphs may have millions of nodes.

We now have some insights into what type of data and tasks graph learning represents. Next, we will introduce a set of mathematical tools for analyzing graphs, and then take a closer look into the learning process via different machine learning paradigms.

1.2.4. Graph matrices

Graphs for matrix encoding. The most intuitive use of matrices in the context of graphs is to *represent* the graph itself, or at least capture certain properties of it. When considering matrix representations of graphs, it is natural to think in terms of indices. For some graph G, its matrix representation (for example in the case of adjacency matrix as we will see below) uses some order of vertices. An actual topological sorting is only possible for directed acyclic graphs (DAG), so in many cases this ordering of vertices for rows and columns is arbitrary. In this case, we can refer to some vertex a through its index i in the ordering, ν_i, w.l.o.g. In the following definitions, we will use this matrix representation notation, but note that the information in the matrix depends on the vertices themselves and not the ordering; in this sense, for some matrix $\mathbf{M}$ that encodes graph G with vertices $\{a, b\}$, $\mathbf{M}(a, b) \iff \mathbf{M}(\nu_i, \nu_j)$, where i and j denote the ordering indices of a and b respectively.

Definition 1.2.4 (Adjacency matrix)**.** *The* adjacency matrix *is perhaps the most common type of matrix associated with a graph. For a weighted graph $G = (\mathcal{V}, \mathcal{E}, w)$, the adjacency*

matrix is defined as:

$$\mathbf{A}_{ij} = \mathbf{A}[i,j] = \begin{cases} w_{ij} & \{\nu_i, \nu_j\} \in \mathcal{E} \\ 0 & \{\nu_i, \nu_j\} \notin \mathcal{E} \end{cases} \tag{1.2.1}$$

For unweighted graphs, we have $w_{ij} = 1 \quad \forall \{\nu_i, \nu_j\} \in \mathcal{E}$.

Definition 1.2.5 (Degree matrix)**.** *Let us consider the* neighborhood *of some vertex* $\nu_i \in G$*, defined as the set of vertices* ν_i *has an edge to:*

$$\mathcal{N}_G(\nu_i) = \mathcal{N}(\nu_i) \coloneqq \{\nu_j \in \mathcal{V} : \{\nu_i, \nu_j\} \in \mathcal{E}\} \tag{1.2.2}$$

The degree of a vertex for an unweighted graph is the cardinality of its neighborhood:

$$\deg(\nu_i) \coloneqq |\mathcal{N}_G(\nu_i)|$$

For a weighted graph, it is the sum of the weights between the vertex and its neighbors:

$$\deg(\nu_i) \coloneqq \sum_{\nu_j \in \mathcal{N}(\nu_i)} w(\nu_i, \nu_j)$$

Let us define the degree vector $\mathbf{d}$*, where* $\mathbf{d}_i \coloneqq \deg(\nu_i)$*. The degree matrix is then defined as an* $n \times n$ *diagonal matrix, which has* $\mathbf{d}$ *as its diagonal:*

$$\mathbf{D}_G(\nu_i, \nu_j) \coloneqq \begin{cases} \mathbf{d}_i & \nu_i = \nu_j \\ 0 & \nu_i \neq \nu_j \end{cases} \tag{1.2.3}$$

Graphs as operators I: Random walks. Matrices are not only used to encode graphs, however; they can also be used as operators on function or vector spaces on vertices $\mathbf{x}$ as $\mathbf{Mx}$. In addition, they can operate as quadratic forms that map function or vectors on vertices $\mathbf{x}$ to some scalar via $\mathbf{x}^\top \mathbf{Mx}$.

The *random walk* operator is a commonly encountered matrix operator on graphs. The idea of a random walk is simple: Starting from some vertex $a \in \mathcal{V}$, we move to one of its neighbors $b \in \mathcal{V}$, $(a, b) \in \mathcal{E}$. However, b is a selected randomly; for unweighted graphs b is selected from a uniform distribution on the neighbors, if weighted the probabilities are proportional to the weights. In matrix form, the random walk operator is also known as the *transition matrix*.

Definition 1.2.6. *Based on the definitions of the adjacency matrix* $\mathbf{A}_G$ *(Definition 1.2.4) and degree matrix* $\mathbf{D}_G$ *(Definition 1.2.5), the random walk matrix is defined as:*

$$\mathbf{W} = \mathbf{W}_G \coloneqq \mathbf{A}_G \mathbf{D}_G^{-1} \tag{1.2.4}$$

We can then define a function δ_a to denote the initial probability distribution:

$$\delta_a(b) \coloneqq \begin{cases} 1 & b = a \\ 0 & b \neq a \end{cases} \tag{1.2.5}$$

Since we start at vertex a, the corresponding index of δ_a is 1, and all others 0. To take a step in our random walk, we multiply $\mathbf{W}$ with δ_a: $\mathbf{W}\delta_a$. The output will provide a new δ vector denoting the probabilities of ending up in each vertex after the first time step, and the values will sum to 1. What happens after t time steps? We can simply apply our operator t times to find out: $\mathbf{W}^\top \delta_a$.

Graphs as operators II: Graph Laplacian. The *graph Laplacian* lies at the heart of spectral graph theory, and is an operator we will familiarize with closely in the coming chapters.

Definition 1.2.7 (Graph Laplacian)**.** *Given a simple, i.e. undirected, unweighted graph G, the graph Laplacian (also known as Laplacian matrix) L is defined by*

$$\mathbf{L}_{ij} = \begin{cases} -1 & \{\nu_i, \nu_j\} \in \mathcal{E} \\ |\delta(\nu_i)| & \nu_i = \nu_j \\ 0 & \textit{otherwise} \end{cases} \tag{1.2.6}$$

Equivalently, the graph Laplacian can be defined as

$$\mathbf{L} = \mathbf{D} - \mathbf{A} \tag{1.2.7}$$

where $\mathbf{D}$ and $\mathbf{A}$ are the degree matrix and adjacency matrix of G, respectively.

The definition of the graph Laplacian is quite simple, but its intuition is not obvious, especially if one is unfamiliar with the Laplace operator (commonly referred to as the Laplacian). Therefore, let us begin by reviewing the Laplacian, and then extending it to its graph counterpart.

Definition 1.2.8 (Laplacian)**.** *Given a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$, the Laplacian of f is defined as the divergence of its gradient:*

$$\Delta f(\mathbf{x}) \coloneqq \nabla \cdot \nabla f(\mathbf{x}) \tag{1.2.8}$$

The gradient operator here takes in a multivariate function f, and outputs a vector field ∇f. The vector at each point $\nabla f(\mathbf{x})$ has a direction which points to f's steepest ascent at $\mathbf{x}$, while its magnitude is proportional to the steepness at this point.

The divergence operator, on the other hand, takes in a vector field $\nu(\mathbf{x})$ and produces a scalar field, which can also be defined by a multivariate function. The divergence at a point $\mathbf{x}$, $\nabla \cdot \nu(\mathbf{x})$, is a scalar quantifying the outward flux of ν from an infinitesimal volume around the point $\mathbf{x}$. More intuitively, the divergence at $\mathbf{x}$ denotes how much "flow" (defined by the vector field) goes in and out of $\mathbf{x}$; if inflow is greater than the outflow (such as in a sink) at $\mathbf{x}$, the divergence is negative, and vice versa.

Circling back to our original definition, the Laplacian is a measure of how the steepness of a function is changing at a given point; it is essentially the analogue of the second derivative of the single-variable function for the multivariate case.

The graph Laplacian $\mathbf{L}$ is the analogue of the Laplacian operator on graphs, which is probably not a surprise to the reader at this point. How does $\mathbf{L} = \mathbf{D} - \mathbf{A}$ capture the same idea though? To understand this, we need to identify the analogous components and relationships between the components in multivariate functions and graphs.

Points: The vertices $\nu \in \mathcal{V}$ of is analogous to the points in a Euclidean space.

Functions: A function f can be defined on $G := (\mathcal{V}, \mathcal{E})$ such that it maps each vertex $\nu \in \mathcal{V}$ to a scalar. Along with an ordering of the vertices, the function can be represented by the following vector:

$$\boldsymbol{f} = f(\boldsymbol{\nu}) := [f(\nu_1), f(\nu_2), \ldots, f(\nu_n)]$$

Gradient: Remember that the gradient of a function is essentially a measure of how much and in which direction a function is changing at each point. Since vertices are analogous to points, the gradient is analogous to the difference of the function values in two vertices. Note that in the continuous case, we refer to the change in an infinitesimal region in terms of proximity to the given point; in the discrete case of graphs, we need to define the gradient through pairs of vertices that have some sort of relationship between them. In graphs, the edges are precisely the structures used to capture such relationships, and provide a natural analogue here. Therefore, we define this gradient *on* edges, i.e. for some edge $\varepsilon = (\nu_i, \nu_j) \in \mathcal{E}$ (we revert to the "ordered" edge definition here, where $i < j$, though the order itself is arbitrary), we can define the gradient analogue as:

$$g(\epsilon_k) := f(\nu_i) - f(\nu_j)$$

Again, using the arbitrary order of vertices, the vector representation of the gradient becomes:

$$\boldsymbol{g} = g(\boldsymbol{\varepsilon}) := [g(\varepsilon_1), g(\varepsilon_2), \ldots, g(\varepsilon_m)]$$

The next step is to construct a matrix operator $\mathbf{K}$ that computes the gradient of f according to the definitions above, i.e. maps $\boldsymbol{f}$ to $\boldsymbol{g}$. $\mathbf{K}$ is known as the *incidence matrix*, though several variations for directed and undirected graphs exist. In our case, we construct $\mathbf{K}$ as follows, where rows correspond to the ordered vertices, and the columns correspond to the ordered edges. Note that we again refer to the ordered notation of edges, and consider

$a < b$:

$$\mathbf{K}_{ij} = \begin{cases} -1 & \nu_i \in \varepsilon_j \coloneqq (\nu_a, \nu_b), \quad i = a \\ 1 & \nu_i \in \varepsilon_j \coloneqq (\nu_a, \nu_b), \quad i = b \\ 0 & \text{otherwise} \end{cases} \tag{1.2.9}$$

Multiplying the transpose of $\mathbf{K}$ with the function vector $\boldsymbol{f}$ gives us $\boldsymbol{g}$: $K^\top \boldsymbol{f} = \boldsymbol{g}$.

Divergence: Previously, we defined divergence as the difference between inflow and outflow at a given point, where the flow is defined by the gradient vector field. Since our vertices are analogous to points, and we define our gradient vector $\boldsymbol{g}$ through the edges, we can compute the divergence for a vertex ν_i by the edges it is adjacent to, $\mathcal{E}_i$. The divergence for a vertex in this case is the sum of all of its outflowing "gradients" encoded in the adjacent edges, minus the sum of all inflowing gradients; we again have to keep in mind here that for undirected graphs, this in/outflow is determined by the arbitrary order of the vertices in matrix form, they do not represent the flow of information in the actual graph. Luckily, $\mathbf{K}$ already encodes this flow information for us: multiplying $\mathbf{K}$ with $\boldsymbol{g}$ computes the divergence! This also validates the notion that divergence (and therefore by definition the Laplacian) is the analogue of the second-order gradient, since we multiply with $\mathbf{K}$ twice, one for the gradient and the second for the divergence:

$$\Delta f(\mathbf{x}) \coloneqq \nabla \cdot \nabla f(\mathbf{x}) = \mathbf{K}\mathbf{K}^\top f(\mathbf{x}) = \mathbf{L} f(\mathbf{x}) \tag{1.2.10}$$

The Laplacian $\mathbf{L}$ can always be factored as $\mathbf{K}\mathbf{K}^\top$, and being able to be factorized in this form is one of the several definitions of a positive semi-definite matrix. Therefore the Laplacian is positive semi-definite, meaning it is symmetric and all of its eigenvalues are non-negative.

While we now have an understanding of how $\mathbf{L}$ is constructed, it is still not obvious how we arrive at the $\mathbf{L} = \mathbf{D} - \mathbf{A}$ definition. To do so, let us look at $\mathbf{K}$ again. Row $\mathbf{K}_{i*}$ corresponds to vertex ν_i, and column $\mathbf{K}_{*j}$ corresponds to edge ε_j; a nonzero $\mathbf{K}_{ij}$ indicates that edge ε_j is connects to vertex ν_i.

With these in mind, let us inspect the diagonal of $\mathbf{K} = \mathbf{K}\mathbf{K}^\top$. $\mathbf{L}_{ij}$ is simply the dot product of the rows i and j of $\mathbf{K}$. The diagonal elements are then multiplication of each row (corresponding to a vertex) with itself, and is equivalent to the sum of squares of its elements. Since each entry of the row indicates whether the corresponding edge is incident to the vertex at hand, the sum of the squares (as we only have $\{-1, 1\}$), we simply end up summing up the number of edges adjacent to each vertex, i.e. the degree. Thus we have shown that the computing the diagonal elements is equivalent to computing $\mathbf{D}$.

Moving on, the off-diagonals $\mathbf{L}_{ij, i \neq j}$ are essentially the dot product of rows i and j of $\mathbf{K}$. Remember that nonzero entries of a row indicate incidence to the corresponding edge; therefore in the dot product the multiplication of individual entries is nonzero only if the corresponding edge is incident to both rows, i.e. the edge connects the two respective vertices.

By our definition, a pair of vertices can be connected only by a single edge, and if two vertices are adjacent, one will have 1 as the corresponding entry, while the other will have -1, resulting in a dot product of -1 in the case of adjacency and 0 otherwise. Consequently, the off-diagonal entries will be equivalent to $-\mathbf{A}$.

This sums up our introduction to the graph Laplacian. As mentioned, the graph Laplacian is a very useful tool to understand the behavior of graph functions, and therefore we will encounter it frequently. Next, we will introduce some graph signal processing tools, and see how these tools leverage spectral theory in graph learning.

1.2.5. Graph Signal Processing

Graph Signal Processing (GSP) refers to the set of approaches used to encode, extract and analyze signals $\mathbf{x} : \mathcal{V} \to \mathbb{R}$ on graph data. This requires extending classical signal processing notions such as Fourier transforms, signal filtering and frequency responses to graphs. This process is conceptually analogous to the extension of deep learning tools to graph data: Classical signal processing is quintessentially done on regular domains such as Euclidean data or time; GSP then aims to extend classical signal processing tools to the graph domain.

Oftentimes, a "classical" time-dependent signal is analyzed through the Fourier transform in signal processing. This approach translates well to GSP, where the graph Fourier transform is one of the most commonly used tools. In the context of graph learning, the graph Fourier transform is essential in motivating graph convolutions and has proven to be elemental to the development of GNN algorithms in recent years. Therefore, defining the classical Fourier transform and extending it to graphs serves as a suitable entry point to graph signal processing. This subsection by no means attempts to be a comprehensive overview of GSP, it is rather intended to present few GSP tools that are required for us to understand GNNs. For a more comprehensive overview, we refer the reader to Ortega et al. [104], Ortega [103], Stankovic et al. [138, 137].

1.2.5.1. The Fourier transform and convolutions. The Fourier transform is commonly described as a mathematical transform that decomposes functions in *time domain* to functions in *frequency domain*. In more general terms, it allows us to represent an input signal (defined in the time domain) as a weighted sum of complex sinusoids. The coefficients of this weighted sum determine the amplitude of each frequency that constitutes the original function.

For a function $f(\mathbf{x})$, its Fourier transform

$$\mathcal{F}(f(\mathbf{x})) = \hat{f}(\mathbf{s}) = \int_{\mathbb{R}^d} f(\mathbf{x}) e^{-2\pi \mathbf{x}^\top si} d\mathbf{x} \tag{1.2.11}$$

decomposes a signal $f(x)$ into a series of complex exponentials $e^{-2\pi \mathbf{x}^\top si}$, where s can be interpreted as the frequency of the sinusoidal component represented by the complex exponential. These sinusoidal components are derived from the fundamental relationship between the trigonometric functions and the complex exponential, as given by Euler's formula:

$$e^{ix} = \cos(x) + i\sin(x)$$

Let us turn our attention to convolutions. Convolution is a mathematical operation on two functions f and g that produces a third function $(f * h)$ that is the integral of the product of the two functions after one is reversed and shifted.

Definition 1.2.9 (Convolution (Continuous))**.**

$$(f * h)(\mathbf{x}) = \int_{\mathbb{R}^d} f(\mathbf{y})h(\mathbf{x}-\mathbf{y})d\mathbf{y} \tag{1.2.12}$$

The convolution operation is tightly related to the Fourier transform via the *convolution theorem.*

Theorem 1.2.10 (Convolution theorem)**.** *The Fourier transform of a convolution of two functions is equal to the element-wise product of the Fourier transforms of the two functions*

$$(f * h)(\mathbf{x}) = \mathcal{F}^{-1}(\mathcal{F}(f(\mathbf{x})) \odot \mathcal{F}(h(\mathbf{x}))) \tag{1.2.13}$$

where $\odot$ denotes the element-wise product. In other words, the convolution operation in one domain (e.g. frequency/spectral) is equivalent to the element-wise product operation in the other (e.g. time/vertex) domain.

In a discrete domain $t \in \{0, \ldots, N-1\}$, the convolution theorem also applies, where $\mathcal{F}$ refers to the discrete-time Fourier transform (DTFT) instead. However, if at least one of the sequences are N-periodic (which applies to our discrete domain of N points) we are able to replace the use of DTFT with DFT with a discrete circular convolution:

Definition 1.2.11 (Convolution (Discrete circular))**.**

$$(f *_N h)(t) = \sum_{\tau=0}^{N-1} f(\tau)h\left((t-\tau)_{\bmod N}\right) \tag{1.2.14}$$

The DFT for a discrete sequence $(f(x_0), f(x_1), \ldots, f(x_{N-1}))$ is then given by:

$$\begin{aligned} s_k &= \frac{1}{\sqrt{N}} \sum_{t=0}^{N-1} f(x_t)\, e^{-\frac{i2\pi}{N}kt} \\ &= \frac{1}{\sqrt{N}} \sum_{t=0}^{N-1} f(x_t) \left(\cos\left(\frac{2\pi}{N}kt\right) - i\sin\left(\frac{2\pi}{N}kt\right)\right) \end{aligned} \tag{1.2.15}$$

where $s_k \in \{s_0, s_1, s_{N-1}\}$ corresponding to each element of the above sequence. If the input sequence and the DFT are real-valued, then the sequence $[s_k], k \in [0, \ldots, N-1]$ gives us the coefficients of a Fourier series, where s_k provides the amplitude of each sinusoidal component $e^{-\frac{i2\pi}{N}k}$ corresponding to frequency $\frac{2\pi}{N}k$. Note that larger k values imply higher-frequency

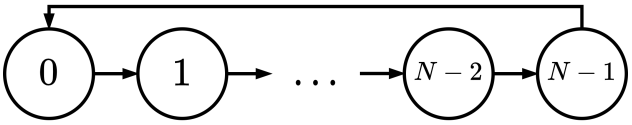


Figure 1.3 – A cycle graph where each node corresponds to a point in a time series of length N.

components, an important property when we extend the Fourier transform and convolutions to graphs.

The discrete convolution $f * h$ can be viewed as applying a filter h over the series $(f(x_0), f(x_1), \ldots, f(x_{N-1}))$; this signal filtering view forms the base of convolutional neural network (CNN) models, which we will briefly discuss later on. Readers familiar with CNNs will also note that convolutions are shift equivariant, i.e. shifting a signal and applying a convolution is equivalent to shifting it after the convolution:

$$f(t+a) * g(t) = f(t) * g(t+a) = (f * g)(t+a) \tag{1.2.16}$$

This in turn makes convolutions equivariant to the difference operator:

$$\Delta f(t) * g(t) = f(t) * \Delta g(t) = \Delta(f * g)(t) \tag{1.2.17}$$

where $\Delta f(t) = f(t+1) - f(t)$.

1.2.5.2. Extending convolutions to graphs. Our coverage of the discrete convolutions has so far focused on discrete time-varying signals in the form of $f(x_0), f(x_1), \ldots, f(x_{N-1})$. As a first step towards extending convolutions to graphs, we can consider the cycle (chain) graph (Figure 1.3), where each time point t is represented by a node. The signal on this graph can be represented by a vector $\mathbf{f}$ where $\mathbf{f}[k] = f(k)$. The edges then represent the propagation of the signal in time (except the edge from $N-1$ to 0, required to keep the domain finite).

The cycle graph is a special type of graph where each vertex has one incoming and one outgoing edge; time flows in only one direction. This makes its adjacency matrix $\mathbf{A_c}$ equivalent to its random walk graph:

$$\mathbf{A}_c[i,j] = \begin{cases} 1 & \text{if } j = (i+1) \bmod N \\ 0 & \text{otherwise} \end{cases} \tag{1.2.18}$$

Since we also have $\mathbf{D}_c = \mathbf{I}$, we have the unnormalized Laplacian in the form of:

$$\mathbf{L}_c = \mathbf{I} - \mathbf{A}_c \tag{1.2.19}$$

Recall our definition of the adjacency matrix (Definition 1.2.4). We see that multiplying a graph function $\mathbf{f}$ with the adjacency matrix operates as a time-shift for the cycle graph:

$$(\mathbf{A}_c\mathbf{f})[t] = \mathbf{f}[(t+1)_{\bmod N}] \tag{1.2.20}$$

This can be generalized to all graphs in the following form: Multiplying a graph function with the adjacency matrix propagates the graph signal at each vertex to its neighbors.

We observe a similar phenomenon with the Laplacian of the cycle graph. Instead of time shifts, it computes the difference between sequential time-steps:

$$(\mathbf{L}_c\mathbf{f})[t] = \mathbf{f}[(t+1)_{\bmod N}] - \mathbf{f}[t] \tag{1.2.21}$$

When extending to general graphs, the Laplacian $\mathbf{L}$ computes the difference between the signal at a node and its neighbors. In other words, the Laplacian measures the *smoothness* of a graph function: A smooth function means the value of the function at a vertex is bound to be similar to its neighbors. This property makes the Laplacian an indispensable tool in GSP, and has advantages in designing convolutional filters as we will see later on.

Next up is the convolution operation, which can also be represented in matrix form based on Equation 1.2.11 (we omit the modulo N since we deal with $=< N$ time steps):

$$\begin{aligned}(f * h)(\mathbf{t}) &= \sum_{\tau=0}^{N-1} f(t-\tau)h(\tau) \\ &= \mathbf{Q}_h\mathbf{f},\end{aligned} \tag{1.2.22}$$

$\mathbf{Q}_h \in \mathbb{R}^{N\times N}$ here represents the convolution by filter h. How do we design $\mathbf{Q}_h$ though? In digital signal processing, filters can be represented by polynomial functions of the shift operator [104], represented by $\mathbf{A}_c$ for the cycle graph:

$$\mathbf{Q}_h = \sum_{i=0}^{N-1} \alpha_i \mathbf{A}_c^i \tag{1.2.23}$$

This representation also satisfies the shift and difference equivariance requirement of convolutions (Eqs. 1.2.16 and 1.2.17):

$$\begin{aligned}\mathbf{A}_c\mathbf{Q}_h &= \mathbf{Q}_h\mathbf{A}_c \\ \mathbf{L}_c\mathbf{Q}_h &= \mathbf{Q}_h\mathbf{L}_c\end{aligned} \tag{1.2.24}$$

For general graphs, this formulation also applies as the adjacency and Laplacian matrices still function as shift and difference operators respectively. Additionally, we need to factor in the m node features $\mathbf{x} \in \mathbb{R}^{N\times m}$ to arrive at the following filter:

$$\mathbf{Q}_h\mathbf{X} = \alpha_0\mathbf{I}\mathbf{X} + \alpha_1\mathbf{A}\mathbf{X} + \alpha_2\mathbf{A}^2\mathbf{X} + \cdots + \alpha_N\mathbf{A}^N\mathbf{X} \tag{1.2.25}$$

This induces a *spatial* notion of convolution, where the signal $\mathbf{Q}_h\mathbf{x}_\nu$ for a vertex $\nu \in \mathcal{V}$ corresponds to a weighted sum of node features aggregated from different hops in each vertices' N-hop neighborhood.

1.2.6. Spectral Theory

Spectral theory can be summarized as the study of *eigenvalues* and *eigenvectors* of matrices. It applies to a broader spectrum than just graphs and is used on matrices and operators on a variety of mathematical spaces, though we will use spectral theory in the context of graph matrices as described in Section 1.2.4. Let us recall the definitions of the two terms: For an $n \times n$ matrix $\mathbf{A}$, an $n \times 1$ vector $\mathbf{v}$ is an eigenvector of $\mathbf{A}$ with corresponding eigenvalue $\lambda \in \mathbb{R}$ if

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

and $\mathbf{v}$ is not a zero vector.

One advantage we have when dealing with graph matrices is that they are almost always real valued, and usually symmetric, which come with some nice spectral properties as demonstrated by the spectral theorem.

Theorem 1.2.12 (Spectral theorem)**.** *Let* $\mathbf{A}$ *be an* $n \times n$ *real, symmetric matrix. Then there exists* n *real eigenvalues* $\lambda_1, \ldots, \lambda_n \in \mathbb{R}$ *and* n *corresponding real, orthonormal eigenvectors* $\mathbf{v}_1, \ldots, \mathbf{v}_n$ *such that:*

$$\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i$$
$$\langle \mathbf{v}_i, \mathbf{v}_j \rangle = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

In addition, in many cases, the graph matrices we encounter are positive semi-definite, such as the Laplacian as we have shown in Equation 1.2.10.

1.2.6.1. The graph Fourier transform. In our construction of graph convolutions, we have restricted ourselves to the adjacency matrix so far. Just as we can construct convolutions *spatially* using the adjacency matrix, we can do so *spectrally* through the Laplacian and its connections to the graph Fourier transform. In fact, the spectral paradigm is arguably dominant over the spatial one when considering graph convolutional networks, as we will explore in further detail in Section 2.4.

We have covered the second-derivative formulation of the Laplacian for continuous signals in Equation 1.2.8, and also shown how it is constructed for a graph. Recall that the divergence of the gradient of a graph signal $\mathbf{x}$ on a vertex is equal to the sum of differences of the function at the vertex and its neighbors:

$$(\mathbf{L}\mathbf{x})_i = \sum_{j \in \mathcal{V}} \mathbf{A}_{ij}(\mathbf{x}_i - \mathbf{x}_j) \tag{1.2.26}$$

This formulation also commutes with the difference operation on the cycle graph, which is simply a special case of Equation 1.2.26.

The eigenvalue problem for the Laplace operator is known as the Helmholtz equation. It states that all eigenvalues λ of the Laplace operator Δ have a corresponding eigenfunction

(equivalent to eigenvectors, but on a function space as opposed to a vector space) f such that:

$$-\Delta f = \lambda f \tag{1.2.27}$$

The eigenfunctions of the Laplacian operator are of the form:

$$-\Delta e^{nxi} = n^2 e^{nxi} \tag{1.2.28}$$

Recall the sinusoidal components of the continuous Fourier transform (Equation 1.2.11), given by $e^{-2\pi xsi}$. The following then holds, where $n = -2\pi s$:

$$-\Delta e^{-2\pi xsi} = (-2\pi s)^2 e^{-2\pi xsi} \tag{1.2.29}$$

This means the eigenfunctions of $-\Delta e^{-2\pi xsi}$ are the complex exponentials that form the sinusoids of the Fourier transform, also known as the Fourier modes. This connection allows us to extend the Fourier transforms to graphs by eigendecomposing the graph Laplacian $\mathbf{L} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top$. The graph Fourier transform then becomes:

$$\mathcal{F}(\mathbf{x}) = \mathbf{U}^\top \mathbf{x} = \hat{\mathbf{x}} \tag{1.2.30}$$

We will provide a more intuitive explanation of the graph Fourier transform in Section 2.4.1, where we will use the eigenfunctions $\mathbf{U}$ to build convolutions in the Fourier domain.

A final relationship we will utilize when building our taxonomy framework is the relationship between the Laplacian eigenvectors and the signal frequencies they capture. Namely, the eigenvectors corresponding to smaller eigenvalues capture low frequency signals; as the magnitude of eigenvalues increase, the associated eigenvectors capture higher frequency components of the graph signal [124]. Low frequencies also imply localization: If the signal variation is low, the signal changes more smoothly and nodes that are spatially close tend to have similar signal values for these components. High-frequency signals on the other hand may oscillate significantly for proximal nodes, and therefore do not provide much positional information. We will see that these properties are leveraged when building spectral filters (e.g. high-pass or low-pass filters) or positional node encodings, two of the many tools we use in developing and benchmarking GNN algorithms.

1.3. Foundations of Geometric Deep Learning

Until the advent of geometric deep learning, the standard way of dealing with graph data was to (a) preprocess the graph to eliminate or transform the graph structure to obtain a simpler representation (such as a vector in $\mathbb{R}$) that is suitable to traditional machine learning, and (b) apply traditional machine learning algorithms or feedforward neural networks to the resulting set of representations, as shown in Figure 1.4. This preprocessing step, in most

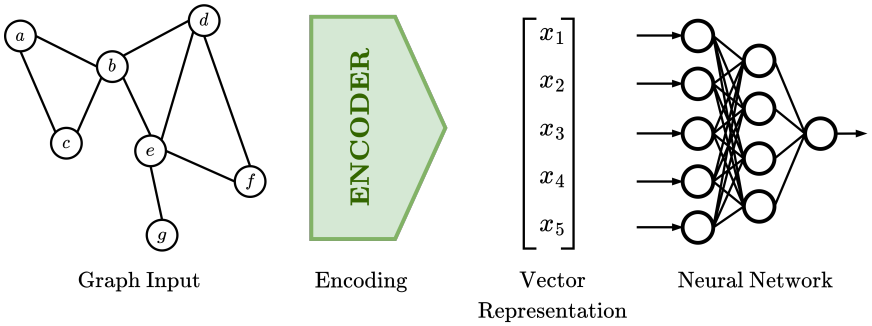


Figure 1.4 – A typical pre-GNN deep learning pipeline: The graph is processed by an encoder to output a vector representation $\mathbf{x}$, which serves as inputs to an MLP. Figure adapted from Sperduti and Starita [134].

cases, leads to the elimination of most if not all topological information and relationships in the graph data.

The main motivator of this encoding approach was the fixed input size of machine learning algorithms in contrast to the variable sizes of graphs. These encodings are usually domain-specific and nontrivial to build; in molecular biology and chemistry, for example *topological indices* [88, 111] are calculated by domain experts to produce the appropriate encodings. Such encoding processes are however often defined *a priori* and do not depend on the statistical learning task. This means that the resulting features may not be relevant to the task at hand: Without prior knowledge of the task, it is possible that the information required is lost already in the encoding process.

The first steps of geometric deep learning emerged from statistical learning models that incorporated preprocessing steps that were able to take the graph structure into consideration. We will here cover several of these statistical learning methods that lead to the first "truly" graph-specialized deep learning algorithm, the Graph Neural Network (GNN) [126].

1.3.1. Recursive Neural Networks

Perhaps the first neural network variant that aims to leverage structural relationships between entities (represented by nodes) is the Recursive Neural Network (RvNN) [134, 42]. The RvNN paper by Sperduti and Starita proposed implementing an adaptive encoder by training another neural network alongside the "main" neural network that is used for the prediction/classification task. This encoder network aimed to learn the best way to encode the graph data such that the encodings capture the task-relevant information for any given learning task. The reader may notice that using neural networks as encoders is currently common or even standard practice in many domains such as computer vision or natural language processing; for late '90s, though, it was a novel approach.

To understand the recursive neural network, let us revisit the standard neuron in a neural network, and introduce the *recurrent neuron*. Recall that the output $o^{(s)}$ of a standard neuron is

$$o^{(s)} = \sigma\left(\sum_i w_i x_i\right) \tag{1.3.1}$$

in the form of some nonlinearity (e.g. rectified linear unit, ReLU) σ applied to the weighted sum of inputs $\mathbf{x}$. For a recurrent neuron with a single self-recurrent connection, this equation becomes:

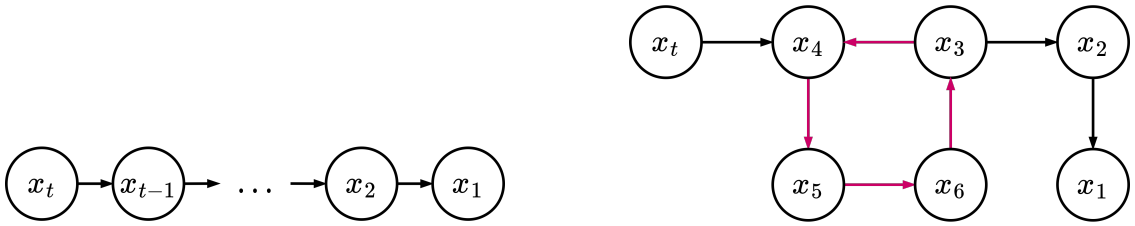
$$o^{(r)}(t) = \sigma\left(\sum_i w_i x_i(t) + w_s o^{(r)}(t-1)\right) \tag{1.3.2}$$

Here, the term within the nonlinearity again the weighted sum, plus the previous output multiplied by some self-weight w_s. Do note that it is not necessary to use the previous output; one can use outputs from several steps before, or sum multiple previous steps to produce the new output.

One major roadblock in the development of these encoder networks was that neither the standard nor recurrent neural networks are not suitable to capture structural relationships in graphs – the standard neuron does not take any structure into consideration, while the recurrent neuron is designed to handle sequential structures only. To overcome this, they propose the *generalized recursive neuron*. The generalized recursive neuron extends the recurrent neuron; instead of incorporating the output of the unit/node in the previous time step, it incorporates the outputs of the corresponding units for all vertices with outgoing edges to the input node. Then, the output $o^{(g)}(x)$ of the generalized recursive neuron corresponding to some vertex x in graph G is computed by

$$o^{(g)}(x) = f\left(\sum_{i=1}^{N_L} w_i l_i + \sum_{j=1}^{\text{out\_degree}_G(x)} \hat{w}_j o^{(g)}(\text{out}_G(x,j))\right) \tag{1.3.3}$$

where N_L is the number of units encoding the label $l = \phi_G(x)$ for the current input x, and $\hat{w}_j$ constituting the weights for the recursive connections from each incoming edge. In other words, the output of the generalized recursive neuron for vertex x is dependent on the output of its (directed) neighbors. The recurrent neuron is then just a special case of the generalized recursive neuron, where the graph is essentially a linear linked list (Fig. 1.5a). Sperduti and Starita [134]'s definition is somewhat counterintuitive here when applied to temporal graphs, as the direction of the arrows is in reverse time. In any case, the recursive application of Equation 1.3.3 produces the following set of equations for this linked list; we make the time variable t explicit here for better interpretability:



(a) A temporal graph as a linked list **(b)** A directed graph with a cycle

Figure 1.5 – The recurrent neuron is suitable for sequential data that can be represented by (acyclic) temporal graphs, while the generalized recurrent neuron can handle cyclical graphs.

$$o(t=1) = \sigma\left(\sum_{j=1}^{N_L} w_j x_j^{(1)}\right)$$

$$o(t) = \sigma\left(\sum_{j=1}^{N_L} w_j x_j^{(t)} + \hat{w}_1 o(t-1)\right) \quad t = 2, \ldots, k$$

Due to the sequential/temporal nature of the data, its graph representation does not have any cycles; it is clear that the recurrent neuron is not designed to deal with cycles. The generalized recursive neuron must however account for cycles. To borrow the example from Sperduti and Starita [134], the cyclic graph in Fig. 1.5b is defined by the following system of equations:

$$o\left(x_1\right) = \sigma\left(\sum_{j=1}^{N_L} w_j x_j^{(1)}\right)$$

$$o\left(x_2\right) = \sigma\left(\sum_{j=1}^{N_L} w_j x_j^{(2)} + \hat{w}_1 o\left(x_1\right) + \hat{w}_2 o\left(x_4\right)\right)$$

$$o\left(x_3\right) = \sigma\left(\sum_{j=1}^{N_L} w_j x_j^{(3)} + \hat{w}_1 o\left(x_2\right)\right)$$

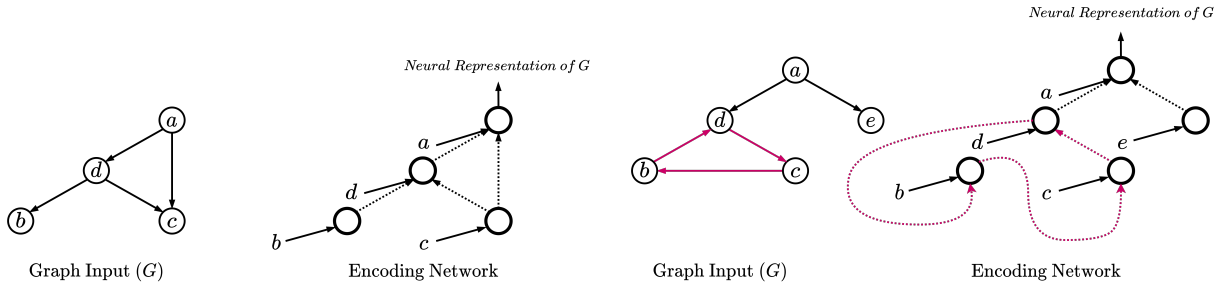
$$o\left(x_4\right) = \sigma\left(\sum_{j=1}^{N_L} w_j x_j^{(4)} + \hat{w}_1 o\left(x_5\right)\right)$$

$$o\left(x_5\right) = \sigma\left(\sum_{j=1}^{N_L} w_j x_j^{(5)} + \hat{w}_1 o\left(x_3\right)\right)$$

$$o\left(x_6\right) = \sigma\left(\sum_{j=1}^{N_L} w_j x_j^{(6)} + \hat{w}_1 o\left(x_5\right)\right)$$

where the set of outputs for x_2, x_3, x_4 and x_5 are interdependent.

Moving on from a single neuron, let us now consider N_g interconnected generalized recursive neurons. We can expand Equation 1.3.3 to a matrix form, with $\boldsymbol{\sigma}_i \boldsymbol{\nu} = \sigma(\nu_i)$, $\mathbf{x} \in \mathbb{R}^{N_L}$, $\mathbf{W} \in \mathbb{R}^{N_g \times N_L}$, $\mathbf{o}^{(g)}(x)$, $\mathbf{o}^{(g)}(\text{out}_G(x,j)) \in \mathbb{R}^{N_g}$, $\hat{\mathbf{W}}_j \in \mathbb{R}^{N_g \times N_g}$:



(a) DAG **(b)** Cyclic graph

Figure 1.6 – Neural representations of a **(a)** directed acyclic graph (DAG) and **(b)** directed cyclic graph (with cycle highlighted). Figure adapted from Sperduti and Starita [134].

$$\mathbf{o}^{(g)}(x) = \boldsymbol{\sigma}\left(\mathbf{W}\mathbf{x} + \sum_{j=1}^{\text{out\_degree}_G(x)} \hat{\mathbf{W}}_j \mathbf{o}^{(g)}(\text{out}_G(x,j))\right) \tag{1.3.4}$$

1.3.1.1. Building the graph encoder. We now proceed with building graph encoders from generalized recursive neurons. The authors posit two conditions for some graph G and generalized recursive neuron u:

(1) **Number of connections:** u must have as many recursive connections as $\max_{x \in \mathcal{V}}\{\text{out\_degree}_G(x)\}$.

(2) **Supersource:** G must have a *supersource*, i.e. a vertex from which all other vertices are reachable. The authors note that in the absence of a supersource, it is possible to add a vertex s to the graph and add the minimum number of outgoing edges to existing vertices of the original graph in order to generate a supersource.

With these conditions in place, we build our encoding framework such that output of the neuron u for the supersource s encapsulates the representation of the whole graph G. Since u is recursive and its output takes into consideration all outgoing neighbors of s, the resulting encoder is a network that mirrors the topology of the original graph, where each vertex has a corresponding generalized recursive neuron. If the graph is acyclic, the resulting encoder is a feed-forward neural network (Fig. 1.6a), if it has cycles, the network is recurrent instead (Fig. 1.6b).

1.3.1.2. Training the graph encoder. For supervised learning on graphs, the encoder we have shown which we designate by Ψ is designed to be trained in conjunction with a feed-forward neural network Φ as a *prediction head* in modern terms. The network prediction is then defined as

$$\mathbf{o}(X) = \Phi(\Psi(X)) \tag{1.3.5}$$

with the errors to backpropagate as follows, where $\mathbf{y} = \Psi(X)$, the output of the encoder and the input to the prediction head:

$$\Delta \mathbf{W}_\Phi = -\eta \frac{\partial \text{Error}(\Phi(\mathbf{y}))}{\partial \mathbf{W}_\Phi} \tag{1.3.6}$$

$$\Delta \mathbf{W}_\Psi = -\eta \frac{\partial \text{Error}(\Phi(\mathbf{y}))}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{W}_\Psi} \tag{1.3.7}$$

In Sperduti and Starita [134]'s framework, the backpropagation algorithm to be used for the encoder depends on the type of graphs. If training data consists of DAGs, standard backpropagation is sufficient; for cyclic graphs, *recurrent backpropagation* [110] is required. In addition, when dealing with cyclic graphs, convergence of the encoding network is not guaranteed. The generalized RvNN framework by Frasconi et al. [42] also avoids cyclic graphs, and also considers the Sperduti and Starita [134] paper on DAGs. In short, we see that recursive neural networks are relatively suitable frameworks to process graphs that lack cycles, but are not well-suited for cyclic graphs. This is a severe restriction; in most domains where graph learning is ubiquitous, be it biochemistry or social networks, cycles are commonplace.

1.3.2. Kernel Methods

Graph kernels are arguably the most popular pre-GNN methods in graph learning, and are still commonly used today. Prediction with graph kernels are used in conjunction with a family of learning algorithms called *kernel machines*, the most popular of which is the support-vector machine (SVM). Without kernels, SVMs are essentially binary linear classifiers: For two classes of linearly separable data in N dimensions, SVM computes an $N-1$ dimensional hyperplane with the maximal distance from the nearest data point from each class. However, through the introduction of slack variables and kernel functions, we adapt SVMs to detect more complex patterns in data. Kernels function as a form of similarity measure over all pairs of data instances computed via inner products. This in turn means we can construct kernel functions for graphs as well.

We will begin with a slight digression into SVMs for a more intuitive understanding of how they function with and without kernels; some parts of the formulation are omitted here in the main body to not detract us too much from our focus on graph kernels, but a more complete version of the SVM formulations are available in Appendix A. We will then introduce the notion of kernel functions, and cover several useful graph kernels.

1.3.2.1. Linear SVMs. Assume we have p data points from two classes in the form of

$$(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_p, y_p)$$

where $\mathbf{x}_i$ is some N-dimensional vector, and $y \in 1, -1$. SVM training outputs a decision function $D(\mathbf{x})$ such that $D(\mathbf{x}) > 0 \implies y = 1$ and $y = -1$. otherwise. For the linear SVM, the decision function defines a separating hyperplane in the form of $\langle \mathbf{w}, \mathbf{x} \rangle + b$, where sign of the output determines the class:

$$D(\mathbf{x}) = \text{sign}(\langle \mathbf{w}, \mathbf{x} \rangle + b) \tag{1.3.8}$$

Here, $\mathbf{w}$ is an N-dimensional vector that determines the orientation of the hyperplane, and b is a bias that represents the offset from the origin. For linearly separable training data $(\mathbf{x}_i, y_i)$, the decision function satisfies

$$\frac{y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)}{\|\mathbf{w}\|} \geq M \tag{1.3.9}$$

and the training objective is to find some $\mathbf{w}$ (which we can normalize to unit length) that maximizes the margin M, resulting in the *maximum-margin hyperplane*:

$$\begin{aligned} M^* &= \max_{\mathbf{w}, \|\mathbf{w}\|=1} M \\ \text{s.t.} \quad & y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq M, \quad i = 1, 2, \ldots, p \end{aligned} \tag{1.3.10}$$

The maximum-margin hyperplane separates the data points of the two classes in a way such that the distance of the hyperplane to the nearest data point from either class is maximized. Inevitably, some data points will satisfy

$$\min_i y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) = M^* \tag{1.3.11}$$

by defining the maximum margin; these points constitute the *support vectors* where the algorithm derives its name from. Now, let us denote $\mathbf{w}_M = \frac{\mathbf{w}}{M}$ and $b_M = \frac{b}{M}$. This allows us to reframe Equation A.1.3 as:

$$\begin{aligned} M^* &= \max_{\mathbf{w}, \|\mathbf{w}\|=1} M \\ \text{s.t.} \quad & y_i(\langle \mathbf{w}_M, \mathbf{x}_i \rangle + b_M) \geq 1, \quad i = 1, 2, \ldots, p \end{aligned} \tag{1.3.12}$$

Since $\|\mathbf{w}_M\| = \frac{\|\mathbf{w}\|}{M} = \frac{1}{M}$, the size of the full margin becomes $2M = \frac{2}{\|\mathbf{w}_M\|}$ (as M denotes the distance denotes the distance from the hyperplane to only one side of the full margin). Thus, maximizing M is equivalent to minimizing the norm $\|\mathbf{w}_M\|$. We can then once again reframe our objective as the following optimization problem:

$$\begin{aligned} &\arg\min_{\mathbf{w}_M, b_M} \frac{1}{2}\|\mathbf{w}_M\|^2 \\ \text{s.t.} \quad & y_i(\langle \mathbf{w}_M, \mathbf{x}_i \rangle + b_M) \geq 1, \quad i = 1, 2, \ldots, p \end{aligned} \tag{1.3.13}$$

The resulting problem is called is called the *hard-margin* SVM. It is a convex quadratic programming (QP) problem, and can be solved directly. However, for high-dimensional

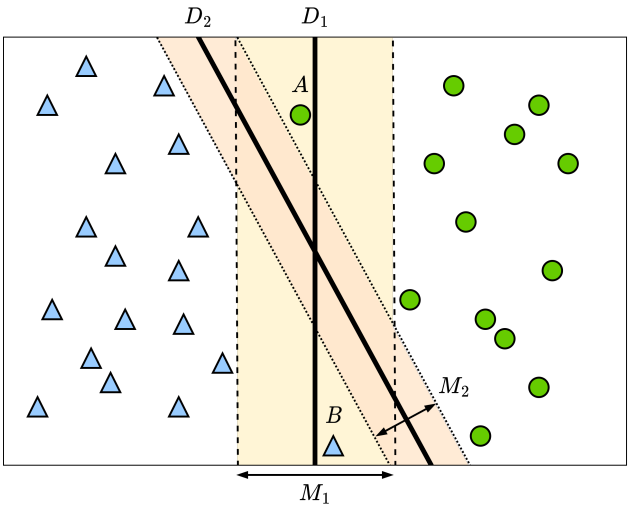


Figure 1.7 – A linearly separable data with two outliers A and B that significantly reduce the separation margin from M_1 to M_2, corresponding to soft-margin classifier D_1 and hard-margin classifier D_2.

spaces the solution space grows extremely large. One can instead introduce Lagrange multipliers $\lambda_i \geq 0, i = 1, \ldots, p$ for the inequality constraints, and solve for the Lagrangian dual of this problem for a more efficient solution (we skip the primal-dual conversion for brevity here, see Appendix A.1):

$$\begin{aligned} & \arg\max_{\boldsymbol{\lambda}} \sum_{i=1}^{p} \lambda_i - \frac{1}{2} \sum_{i=1}^{p} \sum_{j=1}^{p} \lambda_i \lambda_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \\ \text{s.t.} \quad & \sum_{i=1}^{p} \lambda_i y_i = 0, \qquad i = 1, 2, \ldots, p \\ & \lambda_i \geq 0, \qquad i = 1, 2, \ldots, p \end{aligned} \tag{1.3.14}$$

which is also a QP problem that can be solved numerically, and does not depend on $\mathbf{w}$ nor b but only on $\boldsymbol{\lambda}$. Also note that we use the inner product of the data points $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$ here, a property that will prove useful when we leverage kernel functions. The solution $\hat{\boldsymbol{\lambda}}$ can then be used to obtain $\hat{\mathbf{w}}$ corresponding to the maximum-margin hyperplane:

$$\hat{\mathbf{w}} = \sum_{i=1}^{p} \hat{\lambda}_i y_i \mathbf{x}_i \tag{1.3.15}$$

The *hard-margin* SVM has several shortcomings. To begin, it is unsuitable for linearly non-separable data. Furthermore, it is sensitive to outliers. Even in cases where the data is linearly separable, outliers may alter the separating hyperplane in a way that reduces the separation margin, as illustrated by Fig 1.7. In such cases, it may be beneficial to relax the linear separability constraints for outliers in order to obtain a larger margin for the majority of data points. This brings us to the *soft-margin* SVM algorithm.

The soft-margin SVM relaxes the separability constraints by introducing slack variables $\xi_1, \ldots, \xi_p \geq 0$ to the constraints in Eq. 1.3.13:

$$y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1 - \xi_i, \quad i = 1, 2, \ldots, p \tag{1.3.16}$$

These slack constraints allow for data points within the margin, or even on the other side of the decision boundary. For each such data point $\mathbf{x}_i$, ξ_i denotes the distance of the data point from the margin. Now, we want our classifier to not just maximize the margin (and hence minimize $\|\mathbf{w}\|^2$ as shown above), but also to reduce the errors by minimizing ξ_i. We therefore rewrite our optimization problem in Eq. 1.3.13 as

$$\begin{aligned} & \arg\min_{\mathbf{w}} \frac{1}{2}\|\mathbf{w}\|^2 + \beta \sum_{i=1}^{p} \xi_i \\ \text{s.t.} \quad & y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1 - \xi_i, \qquad i = 1, 2, \ldots, p \\ & \xi_i \geq 0, \qquad\qquad\qquad\qquad i = 1, 2, \ldots, p \end{aligned} \tag{1.3.17}$$

where β denotes a configurable constant that is used to manage the trade-off between the two terms to optimize. Whenever we have $y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) < 1$ for some x_i (meaning this data point is within the margins), we pay a cost of $\beta\xi_i$ in the objective function. A point is misclassified if $\xi_i \geq 1$. A large β keeps classification errors to a minimum but may result in a reduced margin, while a small β permits more misclassified examples for a larger margin on the remaining data points.

The dual Lagrangian problem formulation of the soft-margin linear SVM is identical to its hard-margin counterpart in A.1.12, except the final set of constraints on $\boldsymbol{\lambda}$ which are replaced by

$$\beta \geq \lambda_i \geq 0, \quad i = 1, 2, \ldots, p \tag{1.3.18}$$

One again solves the convex QP problem to obtain $\hat{\boldsymbol{\lambda}}$, and derive $\hat{\mathbf{w}}$ similarly:

$$\hat{\mathbf{w}} = \sum_{i=1}^{p} \hat{\lambda}_i y_i \mathbf{x}_i \tag{1.3.19}$$

1.3.2.2. Nonlinear SVMs and the kernel trick. So far, we have only considered SVMs in the input space, i.e. have assumed that the data is (at least partially) separable using a linear decision boundary. In real world data, this is rarely the case. What happens when a linear decision boundary does not capture the true decision boundary? Boser et al. [11] resolve this by the so-called "kernel trick", which transforms the data from the input space to a feature space where the data *is* linearly separable via kernel functions.

As a starting point, let's consider some function ϕ that transforms the data $\mathbf{x}$ from the input space $\mathbb{R}^N$ into the aforementioned feature space, which is no longer necessarily in $\mathbb{R}^N$:

$$\begin{aligned} &\arg\min_{\mathbf{w}}, \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{s.t.} \quad & y_i(\langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle + b) \geq 1, \quad i = 1, 2, \ldots, p \end{aligned} \tag{1.3.20}$$

The resulting dual Lagrangian problem for the soft-margin SVM is:

$$\begin{aligned} &\arg\max_{\boldsymbol{\lambda}} \sum_{i=1}^{p} \lambda_i - \frac{1}{2}\sum_{i=1}^{p}\sum_{j=1}^{p} \lambda_i\lambda_j y_i y_j \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j)\rangle \\ \text{s.t.} \quad & \sum_{i=1}^{p} \lambda_i y_i = 0 \\ & \beta \geq \lambda_i \geq 0, \quad i = 1, 2, \ldots, p \end{aligned} \tag{1.3.21}$$

The optimization process does not require the explicit feature transform ϕ; the inner products $\langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j)\rangle$ are sufficient to recover $\hat{\boldsymbol{\lambda}}$. Similarly, $\hat{b}$ and the decision function $D(\mathbf{x})$just rely on the inner product of the features:

$$\hat{b} = \frac{1}{|\mathrm{SV}_1|}\sum_{i \in \mathrm{SV}_1}\left(y_i - \sum_{j \in \mathrm{SV}} \hat{\lambda}_i y_i \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j)\rangle\right) \tag{1.3.22}$$

$$D(\mathbf{x}) = \mathrm{sign}(\langle \hat{\mathbf{w}}, \mathbf{x}_i\rangle + \hat{b}) = \mathrm{sign}\left(\sum_{i \in \mathrm{SV}} \hat{\lambda}_i y_i \langle \phi(\mathbf{x}_i), \phi(\mathbf{x})\rangle + \hat{b}\right) \tag{1.3.23}$$

The *kernel trick* relies on using a kernel function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ such that $K(x, y) = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j)\rangle$ instead of explicitly using ϕ. But why is this important, or useful? First of all, it is often simpler to design kernels than feature maps. Secondly, designing kernels do not require knowledge of the dimensionality of the feature space. Let us demonstrate with an example.

Consider a feature map $\phi : \mathbb{R}^2 \to \mathbb{R}^6$, $\phi(z) = (z[1]2, z[2]2, \sqrt{2}z[1], \sqrt{2}z[2], \sqrt{2}z[1]z[2], 1)$. Instead, one can use the quadratic kernel $K(x, y)$:

$$\begin{aligned} K(x, y) &= (\langle x, y\rangle + 1)^2 \\ &= x[1]^2y[1]^2 + x[2]^2y[2]^2 + 2x[1]y[1] + 2x[2]y[2] + 2x[1]x[2]y[1]y[2] + 1 \\ &= \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j)\rangle \end{aligned} \tag{1.3.24}$$

Using the corresponding kernel K implicitly computes dot products in $\mathbb{R}^6$ while the explicit computation is only in $\mathbb{R}^2$. Not all kernel functions are suitable though, as bound my the Mercer theorem (1.3.2):

Definition 1.3.1 (Positive-definite kernel)**.** *Let $\mathcal{X}$ be a nonempty set. A symmetric (i.e. $f(x, y) = f(y,x)$) function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a positive-definite kernel on $\mathcal{X}$ if $\forall n \in \mathbb{Z}^+$,*

$x_1, \ldots, x_n \in \mathcal{X}$, *and* $c_1, \ldots, c_2 \in \mathbb{R}$, *the following holds:*

$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j K(x_i, x_j) \geq 0$$

Theorem 1.3.2 (Mercer theorem)**.** *If* $K(x, y)$ *is symmetric, continuous and positive definite, then there exists a function such that* $K(x, y) = \langle \phi(x), \phi(y) \rangle$

Kernel SVM training is based on the dual Lagrangian problem once again; the inner product of feature maps in Eq. 1.3.21 are replaced by the kernel function:

$$\begin{aligned} & \arg\max_{\boldsymbol{\lambda}} \sum_{i=1}^{p} \lambda_i - \frac{1}{2} \sum_{i=1}^{p} \sum_{j=1}^{p} \lambda_i \lambda_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)) \\ \text{s.t.} \quad & \sum_{i=1}^{p} \lambda_i y_i = 0 \\ & \beta \geq \lambda_i \geq 0, \quad i = 1, 2, \ldots, p \end{aligned} \tag{1.3.25}$$

$\hat{b}$ and $D(\mathbf{x})$ are again computed similarly:

$$\hat{b} = \frac{1}{|\mathrm{SV}_1|} \sum_{i \in \mathrm{SV}_1} \left(y_i - \sum_{j \in \mathrm{SV}} \hat{\lambda}_i y_i \langle K(\mathbf{x}_i, \mathbf{x}_j) \rangle \right) \tag{1.3.26}$$

$$D(\mathbf{x}) = \mathrm{sign}(\langle \hat{\mathbf{w}}, \mathbf{x}_i \rangle + \hat{b}) = \mathrm{sign}\left(\sum_{i \in \mathrm{SV}} \hat{\lambda}_i y_i K(\mathbf{x}_i, \mathbf{x}) \rangle + \hat{b} \right) \tag{1.3.27}$$

1.3.2.3. Kernel functions and graphs. For Euclidean spaces $\mathcal{X} \in \mathbb{R}^d$, two commonly used kernel functions are the *polynomial kernel* $K(\mathbf{x}, \mathbf{y}) = (\langle \mathbf{x}, \mathbf{y} \rangle + 1)^q$, which learns a degree-$q$ polynomial decision function, and the *radial basis function (RBF) kernel* $K(\mathbf{x}, \mathbf{y}) = \exp(-\frac{|x-y|^2}{\sigma^2})$, also known as the Gaussian kernel.

However, for a kernel method to be successful on some domain, more than just validity is required. Gärtner [55] identifies three properties of a "good" kernel for a given domain:

(1) *Completeness*: A kernel is complete if it takes into account the necessary information to solve the problem at hand.

(2) *Correctness*: A kernel that satisfies this property reflects the underlying semantics of the problem at hand. In other words, the hypotheses that the kernel structure makes about the problem (and how the kernel will help solve it) are correct.

(3) *Appropriateness:* A kernel is appropriate for a problem if examples that are close or identical to each other in terms of their class or output are also close to each other in the mapped feature space.

A kernel that employs these properties are (a) able to learn underlying concepts in class distinction well; they are also expected to generalize well. Admittedly, the stated properties are somewhat abstract, but still are helpful to obtain a sketch of a good kernel given domain-specific data.

When looking for suitable kernels for structured data, and in particular graphs, we can see that the commonly used kernels do not satisfy the above criteria for a number of reasons. Primarily, standard kernels fail the completeness criteria as they operate on vectors of reals and cannot leverage structural relationships in graph data. As a result, several different kernel approaches that *do* leverage graph structure are proposed.

Diffusion kernels. Kondor and Lafferty [82] borrow ideas from spectral graph theory to propose diffusion kernels for node-level classification tasks. Diffusion kernels are built on exponential kernels. For some square matrix $\mathbf{H}$, its exponential is defined as

$$e^{\beta \mathbf{H}} = \lim_{n \to \infty} \sum_{i=0}^{n} \frac{(\beta \mathbf{H})^i}{i!} \tag{1.3.28}$$

This limit always exists, and $e^{\beta \mathbf{H}}$ is a positive definite matrix if $\mathbf{H}$ is symmetric. A symmetric $\mathbf{H}$ then can be diagonalized $\mathbf{H} = \mathbf{T}^{-1}\mathbf{D}\mathbf{T}$, the diagonal matrix $\mathbf{D}$ can be exponentiated elementwise easily to compute $e^{\beta \mathbf{H}} = \mathbf{T}^{-1} e^{\beta \mathbf{D}} \mathbf{T}$. The matrix $\mathbf{H}$ is called the *generator* matrix.

To construct a matrix $\mathbf{H}$ that represents an undirected graph, Kondor and Lafferty [82] proposes using the negative Laplacian of the graph. Consider an undirected graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the vertex set, and $\{\nu_i, \nu_j\} \in \mathcal{E}$ for each pair of vertices ν_i and ν_j that have an edge in between. The negative Laplacian of the graph is simply the negative of our Laplacian definition in Equation 1.2.6:

$$\mathbf{H}_{ij} = \begin{cases} 1 & \{\nu_i, \nu_j\} \in \mathcal{E} \\ -|\delta(\nu_i)| & \nu_i = \nu_j \\ 0 & \text{otherwise} \end{cases} \tag{1.3.29}$$

Consequently, it is symmetric, negative semi-definite and can be similarly utilized as an operator on functions. The name diffusion kernel, also known as the heat kernel, derives from an analogy with physics. In classical physics, the continuous Laplacian Δ is used in heat equations used to model the diffusion of heat over time in a given region, with equations of the form

$$\frac{\partial}{\partial t}\psi = \mu \Delta \psi$$

As they consider kernels of the form $\mathbf{K}_\beta = e^{\beta \mathbf{H}}$, differentiating this form with respect to β results in

$$\frac{d}{d\beta}\mathbf{K}_\beta = \mathbf{H}\mathbf{K}_\beta$$

With the use of the negative graph Laplacian as the operator $\mathbf{H}$, describing a similar process in discrete form over a graph, the analogy becomes clear. The underlying idea is that unlike standard kernels, the the diffusion process over different vertices will depend on local structural information embedded in the negative Laplacian, meaning kernel methods like SVMs will be able to leverage this diffusion process to classify graph structured data.

Random walk kernels. The kernel function proposed by Kashima et al. [77] uses random walks over graphs as a global similarity measure instead, leveraging marginalized kernels [146]. Unlike diffusion kernels, random walk kernel functions take in two graphs with vertex and edge labels: $\mathbf{K}(G, G')$. The random walk is represented by some hidden variable $h = (h_1, \ldots, h_l)$, where l represents the length of the random walk. The starting vertex is drawn from some initial probability distribution $p_s(h)$ (assumed to be uniform if no prior distribution is known), and subsequent vertices in the walk are sampled from the probability distributions provided by the repeated application of the random walk matrix to the initial distribution. A constant probability is reserved to terminate the random walk if drawn.

Given a pair of graphs (G, G'), a pair of random walks (h, h') can be generated. The traversed vertex and edge labels for a walk can be listed in the form of

$$\nu_{h_1} e_{h_1} \nu_{h_2} e_{h_2} \ldots$$

Assume two non-negative kernel functions $\mathbf{K}(\nu, \nu')$ and $\mathbf{K}(\varepsilon, \varepsilon')$ between vertex labels and edge labels respectively. An example kernel function could just return whether the two vertices are equivalent:

$$\mathbf{K}(\nu, \nu') := \begin{cases} 1 & \nu = \nu' \\ 0 & \nu \neq \nu' \end{cases} \tag{1.3.30}$$

For labels in $\mathbb{R}$, the Gaussian kernel is a natural choice. The authors then define a *joint kernel* as the product of the label kernels, where $\boldsymbol{z} = (G, h)$:

$$\mathbf{K}_z(\boldsymbol{z}, \boldsymbol{z}') = \begin{cases} 0 & (\ell \neq \ell') \\ \mathbf{K}\left(\nu_{h_1}, \nu'_{h'_1}\right) \prod_{i=2}^{\ell} \mathbf{K}\left(\varepsilon_{h_{i-1}h_i}, \varepsilon'_{h'_{i-1}h'_i}\right) \times \mathbf{K}\left(\nu_{h_i}, \nu'_{h'_i}\right) & (\ell = \ell') \end{cases} \tag{1.3.31}$$

The marginalized kernel is then defined as the expectation of the joint kernel $\mathbf{K}_{\boldsymbol{z}}$ over all possible h and h'. Note that this computation is intractable since the possible values of l is infinite. The authors however show that for non-negative kernels $\mathbf{K}(\nu, \nu')$ and $\mathbf{K}(\varepsilon, \varepsilon')$ and a constant nonzero walk termination probability γ, this expectation converges in the limit of l to infinity if

$$\mathbf{K}(\nu, \nu')\mathbf{K}(\varepsilon, \varepsilon') < \frac{1}{(1-\gamma)^2}$$

We omit additional details here, but the underlying idea is important: Two graphs that are likely to have similar random walks are more likely to belong to the same class, while diverging random walks imply distinct classes.

1.3.3. Markov chains

Markov chains, also known as Markov processes, are stochastic models describing sequences of events in which the probability of an event depends only on the previous state, i.e. the result of the previous event. A cornerstone of stochastic modeling and simulation

methods, they come in discrete and continuous-time flavors. Due to the discrete structures of graphs, the discrete-time variant is of importance to us.

Discrete-time Markov chains (DTMC) can be thought of as a (finite) number of states with assigned transition probabilities for each state. One may notice that this formulation is immediately applicable to graphs; each vertex represents a state, while the edges from the vertex to its neighbors determine the states it may transition into. For weighted graphs, the edge weights may encode the transition probabilities to each neighboring state. The reader may at this point realize that we have already covered a form of Markov process in previous chapters: The random walk!

Time-homogeneous (i.e. the transition probabilities do not change over time steps) DTMCs are commonly represented by a *transition matrix*. Recall the random walk matrix $\mathbf{W}$ in Definition 1.2.6 – $\mathbf{W}$ is precisely this transition matrix for the Markov chain representation of the random walk process.

An important property of Markov chains is irreducibility. In an irreducible Markov chain, any state is reachable from any other state in a finite number of time steps. When applied to our analogy with graphs, it means an undirected graph is connected, or for the directed case, strongly connected; the criterion is that any two vertices in the graph are reachable from each other. When a Markov chain is both irreducible and all its states positive recurrent (i.e. all states can reach themselves in finite time as well), it can be proven that it has a *stationary distribution* $\boldsymbol{\pi}$. This is known as the fundamental theorem of Markov chains.

Definition 1.3.3 (Fundamental theorem of Markov chains)**.** *For an irreducible and aperiodic Markov chain with transition matrix* $\mathbf{P}$*, there exists a unique stationary distribution* $\boldsymbol{\pi}$ *such that* $\boldsymbol{\pi}\mathbf{P} = \boldsymbol{\pi}$*. This also implies that for any initial distribution* $X_0 = x$ *the the average probability distribution converges to* $\boldsymbol{\pi}$*:*

$$\lim_{t\to\infty} P(X_t = y | X_0 = x) = \pi_y$$

In the context of random walks on graphs, the fundamental theorem implies that for (strongly) connected graphs, a stationary distribution $\boldsymbol{\pi}$ exists, and the probability that we arrive at vertex ν_i at some time step t approaches π_i as t approaches infinity.

We have now shown how Markov chains can be associated with graphs, and how the random walk operator represents a Markov process over a graph structure. This natural association led to development of random walker based algorithms on graphs, such as the well-known PageRank algorithm [13, 105] which models the hyperlink structure of the internet as a Markov chain, and essentially leverages the stationary vector of the web pages to compute their associated rankings. Page et al. [105] refer to the random walker interpretation as a "random surfer" that clicks on links in a web page at random; the steady state for a web page is then the probability of the random surfer arriving at the web page as $t \to \infty$.

To briefly summarize, PageRanks are computed through the recursive algorithm:

$$X(t+1) = d\mathbf{W}X(t) + (1-d)\mathbb{1}_n \tag{1.3.32}$$

$X(t) \in \mathbb{R}^n$ denotes the PageRanks for the n web pages at time step t, W is the $n \times n$ PageRank matrix constructed based on the graph representation of the web, d is a decay factor and $\mathbb{1}_n$ is a n-length vector of ones. The final term $(1-d)\mathbb{1}_n$ prevents "rank sinks", since clusters of web pages that do not have outgoing links form loops a random walker cannot escape from. The decay factor permits a nonzero probability of randomly jumping out of such loops. The steady state X is then computed as

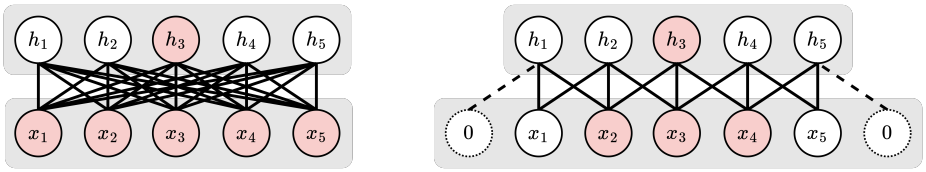
$$X = (1-d)(\mathbf{I} - d\mathbf{W})^{-1}\mathbb{1}_n \tag{1.3.33}$$

The prior work of three authors of the original GNN paper, namely Franco Scarselli, Ah Chung Tsoi and Markus Hagenbuchner, had focused on extending the PageRank algorithm with learning capabilities, with limited success. Their first paper on adaptive page ranking aims to change the steady state through an "optimized" decay vector instead of $\mathbb{1}_n$ [144] to influence the final ranks. It is straightforward to see that changing the decay vector changes the steady state through Equation 1.3.33. This optimization is done via quadratic programming, but is intractable for extremely large n, as is the case for the complete web. The authors then propose several approaches to simplify the problem, such as clustering web pages to reduce n, and relaxing the original constraints to arrive at suboptimal solutions.

A later iteration of this work focuses on building parametric models that learn the page ranking function from examples in order to construct customized page rank algorithms for users [145]. The examples are designed to be subsets of web pages sorted based on specific page features (e.g. prioritizing certain keywords, domains or URLs) in order to reflect the desired custom page rank ordering for these subsets. The authors again formulate customized page ranking as a quadratic problem, and propose several methods to build problem constraints out of page features; each page is assumed to have a set of features F, and a corresponding "focused page rank" for each feature. As an example, the feature may be the existence of some word, and then the focused page rank for this feature would be high for pages that include the word. The adaptive page ranks are then parametrized as a linear combination of the focused page ranks. In addition, in order to leverage the global page ranks (as computed by the original PageRank algorithm) along with the focused ranks, several cost functions to minimize the difference between the two are considered in the optimization process.

1.3.4. Convolutional Neural Networks

Convolutional neural networks (CNN) [29, 86] are one of the most ubiquitous families of neural networks, particularly used for processing regular grid-based data such as images.



(a) A standard MLP layer **(b)** A standard CNN layer with sparsity

Figure 1.8 – Fully-connected MLP and sparsely-connected CNN layers. In the CNN, each unit is connected to only the 3 (determined by kernel size) closest nodes, shown in pink.

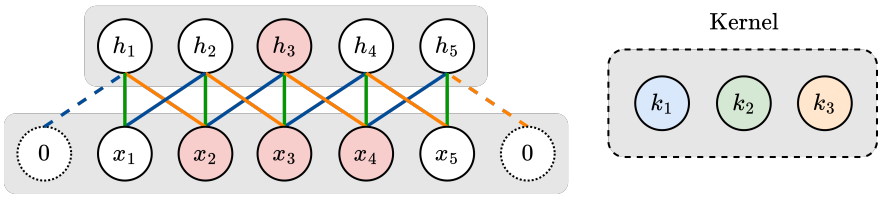


Figure 1.9 – An example of a 1D convolutional layer with a kernel size of 3. The kernel is implemented in the computational graph via sparse connections with shared weights. Edges of the same color share the same weight. The 0-annotated nodes represent "padding" that serves as placeholders for spatial edges.

Fundamentally, CNNs are quite similar to FNNs: They consist of computational units that are composed of trainable parameters and biases, and information is propagated in feedforward fashion. The main differentiators of CNNs are the convolutional layers. Let us recall the the discrete convolution from Section 1.2.5.1 (Definition 1.2.11):

$$(f * h)(t) = \sum_{\tau=0}^{N-1} f(\tau) h\left((t-\tau)\right)$$

CNN layers apply convolutions on local "patches" of image data via a kernel filter. This kernel consists of a set of trainable parameters that typically cover a small portion of the actual domain. This is equivalent to saying the filter h is 0 outside this window. We consider a window of size $2n+1$, such that $h(s) = 0$ for $|s| > n$. We can then modify the discrete convolution as follows:

$$(f * h)(t) = \sum_{\tau=t-n}^{t+n} f(\tau) h(t-\tau) \tag{1.3.34}$$

However, CNNs often deal with data in two dimensions, meaning both our domain and kernel are 2D. Nevertheless, it is not difficult to extend the discrete convolution to two dimensions, even if somewhat tiresome notationally:

$$(f * h)(i,j) = \sum_{\tau=i-n}^{i+n} \sum_{j=\gamma-n}^{\gamma+n} f(\tau,\gamma) h(i-\tau, j-\gamma) \tag{1.3.35}$$

In convolutional layers, these mathematical operations are modeled by specialized connectivity structures and weight sharing. Without loss of generality, we will cover the 1-dimensional case; the model is generalizable to an arbitrary number of dimensions. Consider the comparison of MLP and CNN network connectivities in Figure 1.8. In a standard MLP, every two pairs of nodes from sequential layers are connected. In a CNN convolutional layer, a computational unit s_3 is connected only to those that are spatially closest in the grid structure, forming a *receptive field* $\{x_2, x_3, x_4\}$. This sparse structure mimics the kernel window that is applied to only a small patches (of size 3 in our example) of data at a time, and significantly reduces the computational complexity: A fully-connected layer with N inputs and M outputs requires $N \times M$ parameters, while for a convolutional layer with kernel size K this number reduces to $K \times M$, where $K << N$.

This number is further reduced by parameter sharing, meaning applying the same parameters (represented by the kernel weights) to different sets of inputs in order to produce the layer outputs. Parameter sharing is also demonstrated in Figure 1.9, where edges of the same color share the same parameter. This is equivalent to "sliding" the kernel filter over the layer inputs and completes our model, and further reduces the number of parameters to K from $K \times M$. More importantly, it renders our kernels "convolutional" by implementing the shift equivariance property of convolutions we established in Section 1.2.5.1: The kernel in Figure 1.9 would still produce the same output pattern if the inputs were shifted by any number of units. This property renders convolutional layers excellent "feature extractors" for images, since the visual features that compose an object (e.g. a human face) can be detected by the kernel activations no matter where they are positioned in the image.

The kernel formulation of CNN convolutions brought a new perspective to GNNs as models learning through convolutions. From this perspective, CNNs and GNNs share a commonality in that they employ inductive biases (e.g. translation equivariance, assumption of local connections) that make them more efficient learners. In the next chapter, we will see draw explicit connections between the GNN message-passing framework and convolutions on graphs.

We have now covered the main areas of prior work on graph structures that have both inspired the class of neural networks we broadly group under the umbrella term Graph Neural Network (GNN). Therefore, let us turn our attention to the *neural message passing* framework, the cornerstone of all GNN algorithms.

Chapter 2

Graph Neural Networks: A Survey

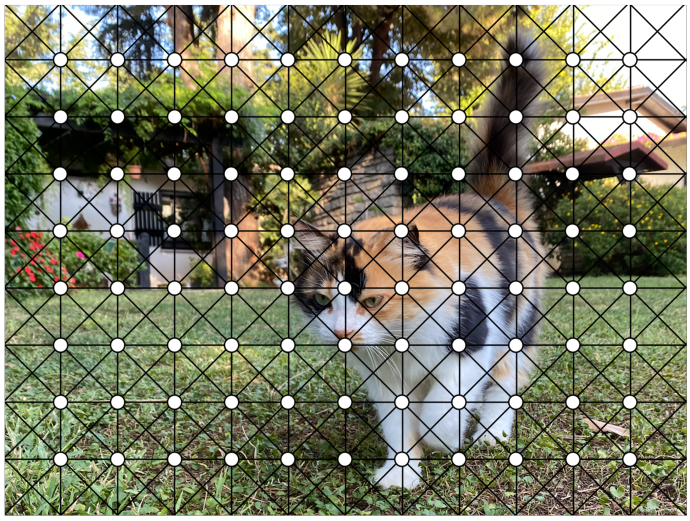
2.1. Motivation for Graph Neural Networks

The success of deep neural networks on unstructured data in the 2000s naturally extended to structured domains in the following years. Recurrent neural networks (RNN) [122, 64] were hugely successful on sequential data (e.g. text and gene sequences), while convolutional neural networks (CNN) replicated this success on grid-structured data (e.g. images), revolutionizing the field of computer vision. Therefore, the extension of deep learning to graph structures was arguably a natural progression. Nevertheless, extending deep learning to graph structures came with unique challenges. The problem stemmed from the structural flexibility of graphs compared to sequences and grids. In fact, sequences and grids can be viewed as specific types of graphs:

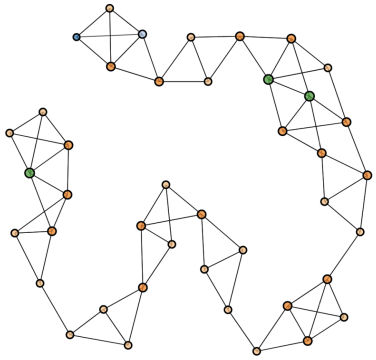
— A sequence is essentially a directed path: It consists of a finite ordered set of vertices, connected by a finite ordered set of edges. The edges are all directed in the same direction, and each vertex is only connected to the vertex before and after them (unless they are the start/end vertices, in which case they may have only one edge). The cycle graph (Fig. 1.3) is the canonical representation of a sequence in graph form.

— In the context of image data and CNNs, a grid is a finite *square grid* over two-dimensional Euclidean space $\mathcal{R}^2$ that forms a regular tiling. Each vertex (pixel) has edges to the eight other vertices that are spatially closest to it (Figure 2.1a). The vertices form a Cartesian plane, where the top-left-most coordinate can be thought to represent the *origin* $(0, 0)$. In its graph representation, the edges represent spatial relationships, and therefore an order of vertices can be established.

One important difference of the more general graph domain is that graphs are almost never regular, as compared in Figure 2.1. Why does this matter? Because RNN and CNN models are able to leverage precisely these structural regularities. For example, CNNs learn patterns in image data using convolution kernels; these kernels are 2D grids that mirror the regular structure of the much larger input space. The learned kernels slide along the input space



(a) An image in euclidean space, which can be viewed as a grid where each pixel is represented by a vertex connected to its spatial neighbors.



(b) A non-euclidean graph from the ENZYMES dataset [10], visualized by Rossi and Ahmed [118].

Figure 2.1 – Fully-connected MLP and sparsely-connected CNN layers. In the CNN, each unit is connected to only the 3 (determined by kernel size) closest nodes, shown in pink.

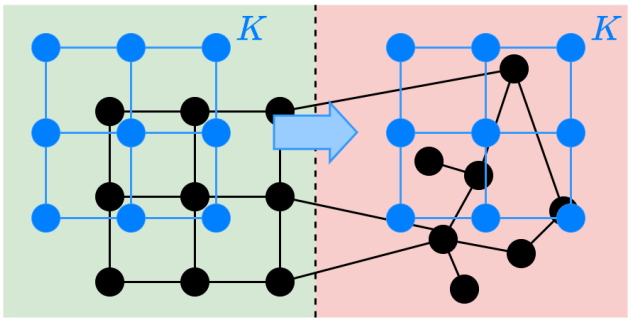


Figure 2.2 – A simplified visualization of standard convolutional kernels failing on graphs. A 3×3 convolution kernel is valid for the left half of the graph, it is not valid for the right.

and perform a Frobenius inner product with different patches of the input space to produce *feature maps*. This "sliding" over input space is only possible because of the regularity of the 2D grid – it does not translate to graphs since if we were to construct some kernel that mirrors the relationships at a region of the graph, it would fail at another region where the designed pattern doesn't apply. A visual representation of this phenomenon is provided in Figure 2.2, using a standard 2D kernel found in CNNs.

One might note that to handle the irregularities in graph structure, matrix representations of graphs may prove useful. We could, for example, flatten the adjacency matrix $\mathbf{A}$ to form a vector, and input the vector to an MLP to derive structural information. This approach has two problems. First, this requires all graphs to have the same number of vertices to maintain

the fixed input size. More importantly, this approach relies on a fixed order of vertices for all graphs, i.e. is not *permutation invariant*. We can define permutation invariance as follows:

Definition 2.1.1 (Permutation invariance)**.** *A function f that operates on an adjacency matrix $\mathbf{A}$ is permutation invariant if it satisfies the following:*

$$f(\mathbf{PAP}^\intercal) = f(\mathbf{A}) \tag{2.1.1}$$

where $\mathbf{P}$ denotes a permutation matrix.

This brings us to the second main difference between sequence/grid data and graphs: It is possible to define a natural order of vertices and/or spatial relationships in such structures; yet since graphs do not lie on a Euclidean domain a natural order or spatial relationships between vertices rarely exist. To summarize, we have identified two points our deep learning algorithm on graphs needs to account for: The irregularity of graph structure, and permutation invariance. These requirements gave birth to a new architectural framework in neural frameworks called neural message passing.

2.2. Neural Message Passing

The defining feature of most GNN architectures is that they use what is now called *neural message passing* [51], in which vertices iteratively exchange messages in the form of vectors (that are updated via the GNN) with their neighbors. Despite being formalized much later than the emergence of GNNs, it is now accepted that GNN algorithms, including the earlier ones, employ some sort of neural message passing. Therefore, we will make a conscious attempt to base our explanations of GNN models on this framework. We will also use the term message-passing neural networks (MPNN) to refer to GNNs that employ this mechanism.

In the message-passing framework, each vertex $\nu \in \mathcal{V}$ is assigned a *hidden state* $\mathbf{h}_\nu^{(k)}$, where k denotes the message-passing iteration, which can be thought of as the sequential message passing layers in a GNN. The initial state at $k = 0$ is defined as the feature set $\mathbf{x}$ for each vertex: $\mathbf{h}_\nu^{(0)} = \mathbf{x}_\nu, \forall \nu \in \mathcal{V}$. At each message-passing iteration, each vertex updates its hidden state by aggregating the hidden states flowing from its neighborhood $\mathcal{N}(\nu)$:

$$\begin{aligned} \mathbf{h}_\nu^{(k+1)} &= \text{UPDATE}^k\left(\mathbf{h}_\nu^{(k)}, \text{AGG}^{(k)}(\{\mathbf{h}_u^{(k)}, \forall u \in \mathcal{N}(\nu)\})\right) \\ &= \text{UPDATE}^k\left(\mathbf{h}_\nu^{(k)}, \mathbf{m}_{\mathcal{N}(\nu)}^{(k)}\right) \end{aligned} \tag{2.2.1}$$

where UPDATE and AGG are differentiable functions, and are parametrized by neural networks. The aggregation function AGG composes the "message" $\mathbf{m}_{\mathcal{N}_\nu}$ using its neighbors' states. UPDATE then updates produces an updated state $\mathbf{h}_\nu^{(k+1)}$ based on the message $\mathbf{m}_{\mathcal{N}_\nu}$ and the current state $\mathbf{h}_\nu^{(k)}$. The output of the final (Kth) message-passing layer is the final

states for each vertex:

$$\mathbf{z}_\nu = \mathbf{h}_\nu^{(K)}, \forall \nu \in \mathcal{V}$$

The neural message passing framework can be seen as an extension of the two main branches of prior work on graphs, recursive neural networks and random walk models. It extends recursive neural networks as it can be naturally applied on a more general class of graphs (as opposed to DAGs only), and can be utilized for node-level tasks (since RvNNs output "global" results that are collected in a supersource). On the other hand, GNNs extend random walk-based methods by incorporating a gradient-based learning algorithm as opposed to quadratic programming. In addition, one may note that the message propagation process is analogous to diffusion on graphs, which we briefly covered in Section 1.3.2.3, where each node diffuses its state to its neighbors.

At each iteration of message-passing algorithm, a vertex gathers information about the states of vertices further and further away in terms of connectivity. After k iterations, the state $\mathbf{h}_\nu^{(k)}$ of vertex ν incorporates information from its k-hop neighborhood $\mathcal{N}_k(\nu)$, i.e. from all vertices that are at most k hops away. This is similar to the *local feature aggregation* observed in CNNs. The main difference is that the CNN convolutions rely on the spatial relationships to aggregate information over the 2D grid, while in GNNs this aggregation behavior depends on graph connectivity.

We had mentioned in Section 1.2.2 that graphs embed information in the form of *node/edge features* and *graph structure*. It is obvious that the vertex states in MPNNs encode feature-based information from its k-hop neighborhoods. In addition, the message-passing framework leverages structural information in graphs by modulating the information propagation based on the graph structure, since the states are propagated to *only* to neighborhoods as determined by graph connectivity.

Our presentation of message-passing in Equation 2.2.1 depends on two functions UPDATE and AGG, which we have so far discussed in an abstract way. MPNN models implement these functions using neural networks; different MPNN flavors of differ in their implementations. We will now explore the original GNN algorithm from the lens of message passing, and work our way up to more recent and successful implementations that serve as common baselines in GNN benchmarking. Many of the older MPNNs were not published with the message-passing framework in consideration; the framing and notation in our explanations may occasionally differ from the original papers as result.

2.3. The Original GNN

Gori et al. and Scarselli et al. set out to unify ideas from recursive neural networks and random walk models by proposing the Graph Neural Network (GNN) that is first outlined in [53], and then further detailed in [126]. Their idea closely resembles the MPNN framework

we describe: Each vertex in the graph is assigned a state vector $\mathbf{h}$, and an *encoding network* is built that mirrors the structure of the graph (Figure 2.3). The "message" is constructed as simply the sum of all neighboring states:

$$\mathbf{m}_{\mathcal{N}(\nu)} = \text{AGG}(\{\mathbf{h}_u, \forall u \in \mathcal{N}(\nu)\}) = \sum_{u \in \mathcal{N}(\nu)} \mathbf{h}_u \tag{2.3.1}$$

Note that we did not need to use a neural network in the aggregation step since in this case, aggregation is equivalent to a sum. The neural network is used in the update step, however, with trainable weight matrices $\mathbf{W}_{\text{self}}^{(k)}$ and $\mathbf{W}_{\text{neigh}}^{(k)}$, $\in \mathbb{R}^{d^k \times k-1}$, and bias $\mathbf{b}^{(\mathbf{k})} \in \mathbb{R}^{d^k}$:

$$\text{UPDATE}(\mathbf{h}_\nu, \mathbf{m}_{\mathcal{N}(\nu)}) = \sigma(\mathbf{W}_{\text{self}}\mathbf{h}_\nu + \mathbf{W}_{\text{self}}\mathbf{m}_{\mathcal{N}(\nu)}) \tag{2.3.2}$$

For brevity, we omit the iteration superscripts (k) here. Also note that in most GNN layer formulations, the bias $\mathbf{b}^{(\mathbf{k})}$ may be omitted for brevity as well – using biases is the norm in practice, but since they are simply added to the layer computation at the end they introduce notational overhead. The full message-passing equation then becomes:

$$\mathbf{h}_\nu^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\,\mathbf{h}_\nu^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)} \sum_{v \in \mathcal{N}(\nu)} \mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\right) \tag{2.3.3}$$

Note that we also apply an elemental nonlinearity σ (e.g. ReLU) in our update step; this nonlinearity and bias are not explicitly stated in the paper, but are the current standard in implementation for optimal performance; we therefore opt to include them.

A common simplification to this MPNN model is using a single matrix $\mathbf{W}$ for $\mathbf{W}_{\text{self}}^{(k)}$ and $\mathbf{W}_{\text{neigh}}^{(k)}$, and to include the vertex itself in the aggregation step:

$$\mathbf{h}_\nu^{(k)} = \sigma\left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(\nu) \cup \{\nu\}} \mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\right) \tag{2.3.4}$$

This can be thought of adding self-loops to the adjacency matrix. The updates for all vertices can be represented using matrix notation, with the help of the adjacency matrix:

$$\mathbf{H}^{(k)} = \sigma\left((\mathbf{A} + \mathbf{I})\mathbf{H}^{(k-1)}\mathbf{W}^{(k)}\right) \tag{2.3.5}$$

The self-loop GNN formulation in fact operates as a 1-hop convolutional filter, based on our graph convolution design in Equation 1.2.25. Nonetheless, this simple adjacency matrix based formulation suffers from several issues:

(1) Despite our somewhat simplified description of the original GNN, the algorithm proves to be computationally expensive in practice as message propagation continues iteratively until convergence.

(2) Computing a simple sum of neighboring states in the AGG step means some vertex ν with a much higher degree than vertex u will have a hidden state $\mathbf{h}$ with a much

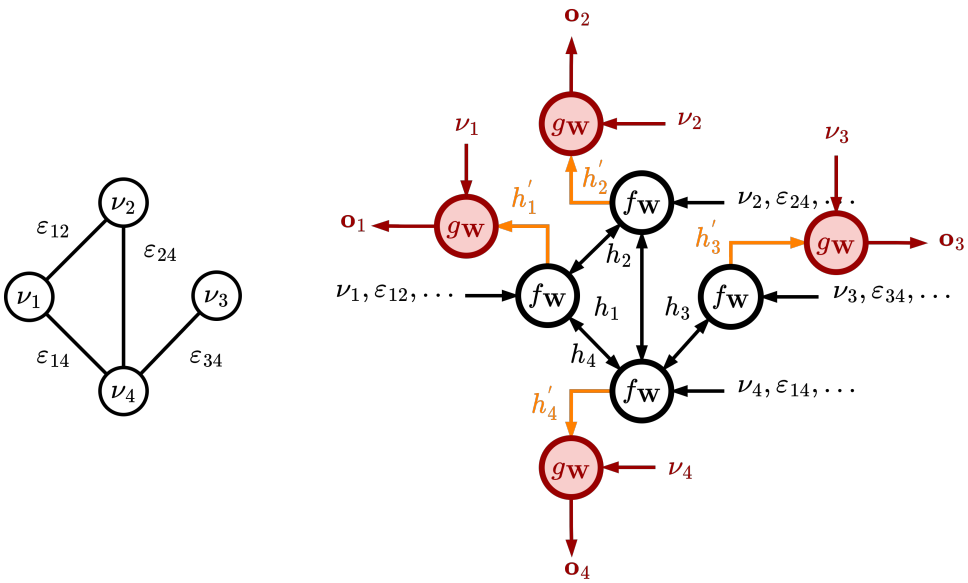


Figure 2.3 – A graph and its Scarselli et al. [126] encoding network representation. The computational units $f_\mathbf{W}$ and $g_\mathbf{W}$ are feedforward neural networks that take in current states h, vertex labels ν_n and edge labels e_{mn} and updates each state according to the modeled graph connectivity. The resulting network is a recurrent neural network (RNN), though it can be simplified into an MPNN form. Final states h'_n are then put through the output network $g_\mathbf{W}$ to produce node-level outputs $\mathbf{o}$.

higher vertex norm as well: $\| \sum_{v \in \mathcal{N}(\nu)} \mathbf{h_v} \| >> \| \sum_{v' \in \mathcal{N}(u)} \mathbf{h_{v'}} \|$. Such sensitivities to node degrees makes convergence more difficult and may lead to numerical instabilities.

(3) The authors' formulation requires the parametrized functions to be a *contraction map*, which limits the function space that can be used.

(4) When the fixed point is reached, it is likely that individual vertex states converge to similar values as well as many iterations mean information from even the furthest away nodes are incorporated into the states for each vertex; this is similar to heat diffusion in space where at convergence every point is at the same temperature. This is called the *oversmoothing* problem, and is commonly observed in other GNN variants as well.

The next breakthrough in GNNs would be inspired by the introduction of image convolutions to deep learning via CNNs and would center around improved extensions of convolutional kernels to graphs.

2.4. Graph Convolutional Networks (GCN)

The success of CNNs in the image domain and the structural analogies between image and graph data motivated researchers to generalize convolutional kernels to graph neural networks. Graph convolutions are usually categorized into two as spectral and spatial convolutions. Spectral GCNs define convolutions through spectral filters as commonly used in graph signal processing; these filters are based on the spectral decomposition of the matrix representations of graphs such as the Laplacian. Spatial GCNs define the convolution operation directly on the graph topology instead, and are motivated by information propagation processes on graphs. We will see that the two approaches are nevertheless connected, and can in fact be unified under certain conditions. Let us focus on spatial graph convolutions first; the relationship between the two categories will emerge naturally soon enough as we explore several variants.

2.4.1. Spectral convolutions on graphs

Spectral convolutions are concerned with the eigendecomposition of graph matrices, namely the adjacency matrix and the Laplacian. These two matrices are real and symmetric by design and therefore are diagonalizable by the spectral theorem (Theorem 1.2.12), hence the name "spectral". In Section 1.2.4, we have covered several properties the adjacency matrix and the graph Laplacian. Recall that multiplying a graph signal $\mathbf{x}$ with the adjacency/random walk matrix $\mathbf{A}$ diffuses/passes the signal to neighbors, while multiplication by the Laplacian $\mathbf{L}$ provides the difference between the signal at a node and its neighbors. In practice, their symmetric normalized variants, *symmetric normalized Laplacian/adjacency matrix* may be used:

$$\mathbf{L}_{\text{ns}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{L}\mathbf{D}^{-\frac{1}{2}} \tag{2.4.1}$$

$$\mathbf{A}_{\text{ns}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}} \tag{2.4.2}$$

$$\mathbf{L}_{\text{ns}} = \mathbf{I} - \mathbf{A}_{\text{ns}} \tag{2.4.3}$$

These symmetric normalized forms retain most of the original matrices' properties; e.g. the normalized Laplacian is still real, symmetric and positive semi-definite. However, they have two useful additional properties [59]:

(1) $\mathbf{L}_{\text{ns}}$ and $\mathbf{A}_{\text{ns}}$ have bounded spectra, i.e. their eigenvalues are bounded. This enables numerical stability and better convergence properties in the optimization process.

(2) $\mathbf{L}_{\text{ns}}$ and $\mathbf{A}_{\text{ns}}$ are *simultaneously diagonalizable*, meanning they share the same set of eigenvectors. This becomes evident in their diagonalization based on Equation 2.4.3

$$\mathbf{L}_{\text{ns}} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top \qquad \mathbf{A}_{\text{ns}} = \mathbf{U}(\mathbf{I} - \mathbf{\Lambda})\mathbf{U}^\top \tag{2.4.4}$$

where $\mathbf{\Lambda}$ is the diagonal matrix of the eigenvalues of the normalized Laplacian, and $\mathbf{U}$ the matrix of eigenvectors for both matrices. This property is particularly important when designing convolutional filters, since a filter matrix commutative with one of the matrices will imply commutativity with the other.

The Fourier transform is generalized to graphs through the eigendecomposition of its (normalized) Laplacian: $\mathbf{L}_{\text{ns}} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top$, where the eigenfunctions $\mathbf{U}$ constitute the graph Fourier modes (the complex exponentials of the Fourier series composing the function). The eigenfunctions of the normalized Laplacian form an orthonormal space, i.e. $\mathbf{U}^\top\mathbf{U} = \mathbf{I}$. Since in the graph domain we define graph signal/functions over vertices, this replaces the notion of time in the standard continuous/discrete Fourier transform. The notion of frequency domain is similarly replaced with *spectral domain* when we consider graph Fourier transforms. The graph Fourier transform projects a graph signal $\mathbf{x}$ to the orthonormal basis defined by $\mathbf{U}$:

$$\mathcal{F}(\mathbf{x}) = \mathbf{U}^\top\mathbf{x} = \hat{\mathbf{x}} \tag{2.4.5}$$

This is by formulation the same as Equation 1.2.30, the only difference is that the eigenfunctions are from the normalized Laplacian. The inverse is then defined as:

$$\mathcal{F}(\hat{\mathbf{x}}) = \mathbf{U}\hat{\mathbf{x}} \tag{2.4.6}$$

The convolution theorem (Theorem 1.2.10) is also applicable to graph domain, and proves useful here. Consider a graph signal $\mathbf{x}$ and a filter $\mathbf{h}$; the convolution theorem states that their convolution can be defined by the inverse Fourier transform of the element-wise product of their Fourier transforms:

$$\mathbf{x} *_G \mathbf{h} = \mathbf{U}(\mathbf{U}^\top\mathbf{x} \odot \mathbf{U}^\top\mathbf{h}) \tag{2.4.7}$$

Note the use of $*_G$ to represent the convolution operation over the domain of G. Since the Fourier transform is defined by $\mathbf{U}$ as it depends on the Laplacian of graph G, the convolution operation is defined only on a graph G; the subscript G is usually omitted for brevity.

Now, consider the filter $\mathbf{h}$ in the Fourier domain: $\mathbf{U}^\top\mathbf{h} \in \mathbb{R}^{|\mathcal{V}|}$. We can diagonalize it as $\mathbf{h}_\theta = \text{diag}(\mathbf{U}^\top\mathbf{h})$ and use it to simplify Equation 2.4.7:

$$\mathbf{x} *_G \mathbf{h} = \mathbf{U}(\mathbf{U}^\top\mathbf{x} \odot \mathbf{h}_\theta) \tag{2.4.8}$$

$$= \mathbf{U}\mathbf{h}_\theta\mathbf{U}^\top\mathbf{x} \tag{2.4.9}$$

Spectral convolutional GNNs all follow this definition of graph convolutions and take the form in Equation 2.4.9; GCN implementations mostly differ on how they define the spectral filters $\mathbf{h}_\theta$. The first spectral GCN proposed by Bruna et al. [17] simply uses a trainable diagonal matrix $\text{diag}(\theta_\mathbf{h})$ as its spectral filter. Such nonparametric filters however are not localized in space, and are computationally expensive for large graphs as they require an explicit eigendecomposition of the Laplacian.

A better approach is to parametrize $\mathbf{h}_\theta$ based on the eigenvalues of the normalized Laplacian $\mathbf{\Lambda}$, i.e. $\mathbf{h}_\theta(\mathbf{\Lambda})$. This enables the spectral convolution to commute with the normalized Laplacian, avoiding the expensive eigendecomposition step:

$$\begin{aligned} \mathbf{f} *_G \mathbf{h} &= (\mathbf{U}\mathbf{h}_\theta(\mathbf{\Lambda})\mathbf{U}^\top)\,\mathbf{x} \\ &= \mathbf{h}_\theta(\mathbf{L}_{\text{ns}})\mathbf{x} \end{aligned} \tag{2.4.10}$$

Defferrard et al. [27] leverages this property by parametrizing the filters as a polynomial function of the Laplacian eigenvalues using the Chebyshev expansion [61]. The Chebyshev polynomials have an efficient recurrent formulation in the form of:

$$\begin{aligned} T_0(\mathbf{x}) &= 1 \\ T_1(\mathbf{x}) &= \mathbf{x} \\ T_i(\mathbf{x}) &= 2\mathbf{x}T_{i-1}(\mathbf{x}) \end{aligned} \tag{2.4.11}$$

The spectral filter then takes the form $\mathbf{h}_\theta = \sum_{i=0}^{K} \theta_i T_i(\tilde{\mathbf{\Lambda}})$ where $\tilde{\mathbf{\Lambda}} = \frac{2\mathbf{\Lambda}}{\lambda_{\max}} - \mathbf{I}$ as the diagonal eigenvalue matrix normalized with the largest eigenvalue $\lambda_{\max}$. The resulting convolution operation is:

$$\mathbf{x} *_G \mathbf{h}_\theta = \mathbf{U}\left(\sum_{i=0}^{K} \theta_i T_i(\tilde{\mathbf{\Lambda}})\right)\mathbf{U}^T\mathbf{x} \tag{2.4.12}$$

This form still relies on the eigendecomposition of $\mathbf{L}_{\text{ns}}$, but we can use Equation 2.4.10 to obtain the form that relies on the Laplacian itself only:

$$\mathbf{x} *_G \mathbf{h}_\theta = \sum_{i=0}^{K} \theta_i T_i(\tilde{\mathbf{L}}_{\text{ns}})\mathbf{x} \tag{2.4.13}$$

Note that the Chebyshev polynomials are computed up to some order K, which is in fact an approximation of the original filter; the polynomial is truncated at order K. Furthermore, constructing spectral filters based on the Kth order polynomials of the Laplacian mean that our filters are now localized, since multiplication with the Laplacian implies gathering information from the immediate neighborhood of each vertex. Increasing orders of the Laplacian in turn imply aggregate information from further and further away. The form we have arrived is then a weighted sum of the information in each node's K-hop neighborhood, with the trainable parameters θ_i controlling the "importance" assigned to information flowing in from different hops.

Kipf and Welling [79] propose several simplifications to the Defferrard et al. formulation to arrive at a more efficient GCN model. This highly successful variant is usually the network referred to with the term *GCN*, and is considered a strong baseline model even today.

The authors first limit $K = 1$, this results in a convolution that is linear on the Laplacian layer-wise, but richer functions can still be modeled by stacking multiple convolutional layers. This allows for building deeper GCNs on limited computational budgets, since multiple layers of higher-order polynomial convolutions lead to an exponential increase in computational

requirements. They also approximate $\lambda_{\max} \approx 2$. These changes simplify Equation 2.4.13 to:

$$\mathbf{x} *_G \mathbf{h}_\theta = \theta_0 \mathbf{x} + \theta_1 (\mathbf{L}_{\text{ns}} - \mathbf{I})\mathbf{x} \tag{2.4.14}$$

$$= \theta_0 \mathbf{x} + \theta_1 \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}} \mathbf{x} \tag{2.4.15}$$

The symmetric normalization we introduced in Equation 2.4.1 is leveraged here: The transition in 2.4.15 is only possible due to the Laplacian being in its symmetric formalized form, as per Equation 2.4.3.

To further reduce the number of trainable parameters, the authors set $\theta = \theta_0 = -\theta_1$. The graph convolution then becomes:

$$\mathbf{x} *_G \mathbf{h}_\theta = \theta (\mathbf{I} + \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}}) \mathbf{x} \tag{2.4.16}$$

The authors note that the repeated application of this convolution may lead to numerical instabilities, so as a final addition they apply a *renormalization trick*:

$$\begin{aligned} \mathbf{I} + \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}} &\rightarrow (\mathbf{D} + \mathbf{I})^{-\frac{1}{2}} (\mathbf{I} + \mathbf{A}) (\mathbf{D} + \mathbf{I})^{-\frac{1}{2}} \\ &\rightarrow \tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}} = \tilde{\mathbf{A}}_{\text{ns}} \end{aligned} \tag{2.4.17}$$

The renormalization trick simply allows us to obtain the normalized symmetric form of $\mathbf{I} + \mathbf{A}$. This ties the Kipf and Welling GCN very smoothly to the MPNN framework. Recall the matrix form of the basic GNN in Equation 2.3.5:

$$\begin{aligned} \mathbf{H}^{(k)} &= \sigma\left((\mathbf{A} + \mathbf{I})\mathbf{H}^{(k-1)}\mathbf{W}^{(k)}\right) \\ &= \sigma\left(\tilde{\mathbf{A}}\mathbf{H}^{(k-1)}\mathbf{W}^{(k)}\right) \end{aligned}$$

Converting Equation 2.4.16 to graph-level matrix form and applying a nonlinearity σ gives us something very similar:

$$\mathbf{H}^{(k)} = \sigma\left(\tilde{\mathbf{A}}_{\text{ns}}\mathbf{H}^{(k-1)}\mathbf{W}^{(k)}\right) \tag{2.4.18}$$

We have recovered the original message-passing GNN (Equation 2.3.5), with the sole difference being that we use the symmetric normalized variant of the adjacency matrix with self-loops, $\tilde{\mathbf{A}}_{\text{ns}} = (\mathbf{I} + \mathbf{A})_{\text{ns}}$.

This very elegant formulation unifies the spectral and spatial views of the GCN into one: We have defined the graph convolution from a spectral-domain perspective as an approximation of a function operating on the graph Laplacian eigenvalues $\mathbf{\Lambda}$, yet the resulting convolutional layer simply aggregates information from spatially neighboring nodes and itself to update its state.

To motivate this GCN formulation through the spatial perspective, we can focus on the aggregation operation of the original GNN. In discussion of the shortcomings of the original GNN in Section 2.3, we had mentioned that the sum aggregation step (Equation 2.3.1) may lead to numerical instabilities in (2). A simple alternative to improve upon this is to compute

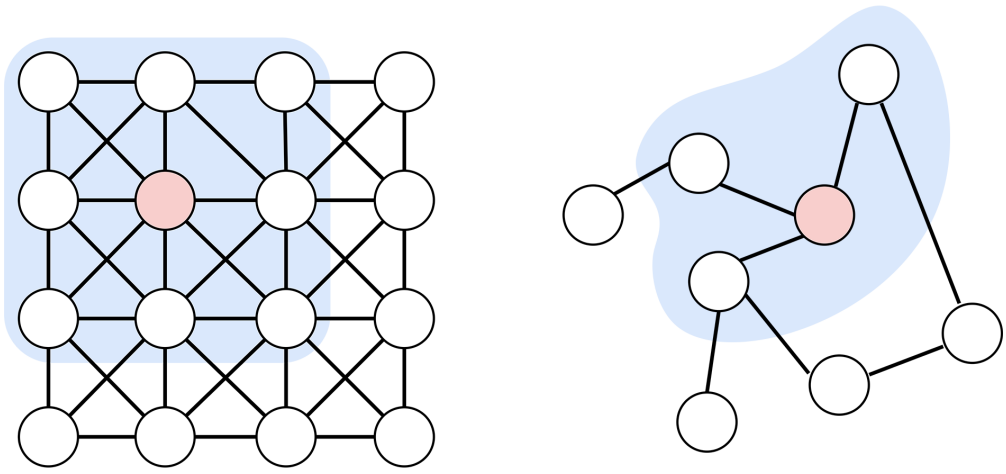


Figure 2.4 – A CNN convolution with a 3×3 kernel (left) vs spatial graph convolution (right). The pixel/vertex the convolution is computed for is highlighted in pink, while the convolved region is highlighted in blue. Each convolves a region equivalent to a 1-hop neighborhood of the center vertex, which is regular on a Euclidean grid and irregular for a general graph.

the average of the neighboring states:

$$\mathbf{m}_{\mathcal{N}(\nu)} = \frac{\sum_{u \in \mathcal{N}(\nu)} \mathbf{h}_u}{|\mathcal{N}(\nu)|} \tag{2.4.19}$$

The spectral formulation of Kipf and Welling [79] in fact performs a similar averaging operation through *symmetric normalization*:

$$\mathbf{m}_{\mathcal{N}(\nu)} = \sum_{v \in \mathcal{N}(\nu)} \frac{\mathbf{h}_u}{\sqrt{|\mathcal{N}(\nu)||\mathcal{N}(u)|}} \tag{2.4.20}$$

This is how the GCN unifies the two motivating points of view: Replacing the standard aggregation with the symmetric-normalized version in place of the standard GNN message-passing aggregation in Equation 2.3.3 is analogous to a first-order approximation of a spectral graph convolution!

2.4.2. Spatial convolutions on graphs

We can also motivate spatial graph convolutions without relying on the spectral derivation. Spatial convolutions form a direct analogy to convolutions in CNNs, as visually demonstrated in Figure 2.4. CNNs apply $n \times n$ filters over identically structured "patches" of the image; the size of the filter defines the "neighborhood": A 3×3 filter/kernel is equivalent to the 1-hop neighborhood of the center vertex on the 2D grid. The actual convolution operation is simply the sum of the elementwise multiplication of the $n \times n$ patch with the $n \times n$ kernel. The regularity of the grid once again guarantees that in vectorized form, the size of the filter and patch will be the same.

Spatial convolutions are similarly defined for each vertex based on the message passing framework: For a given vertex, the states of its neighborhood and itself can be vectorized and/or aggregated, and then multiplied with a filter of the same size. There are, however, two problems with this approach: (a) Unlike images data and CNNs, we cannot guarantee a fixed size of neighborhood for every vertex since the degrees may vary for each; it is impossible to directly apply a fixed-size filter directly on the neighborhood. (b) There are no predetermined orderings of vertices. Spatial GCNs are mostly distinguished by how they resolve these problems of non-regularity. The literature on spatial GCNs is quite vast, but the general approach is to aggregate the vertex neighborhood into a fixed size representation. Citing a few examples here will give us a better idea:

— PATCHY-SAN by Niepert et al. [102] orders the neighborhood nodes based on some criteria (e.g. degree or PageRank), and selects the first k nodes to keep the neighborhood size fixed. This framework is a direct analogy to the CNN convolution, as the MPNN aggregation operation is essentially a concatenation: a fixed size of neighborhood is selected and ordered to form a structured local "patch", which is then learnable by standard CNNs.

— Learnable graph convolutional network (LGCN) by Gao et al. [45] similarly transforms graph data into a 2D grid by using some node feature x themselves to order the nodes, and again picking k nodes with the largest x values to keep the neighborhood fixed. Afterwards, 1-D CNN kernels are used for learning on the 2-D patches.

— Diffusion-convolutional neural network (DCNN) by Atwood and Towsley [3] performs a k-step diffusion by computing powers of the transition matrix $\mathbf{P}^k$, where $\mathbf{P} = \mathbf{D}^{-1}\mathbf{A}$ (Definition 1.2.6):

$$\mathbf{H}^{(k)} = \sigma(\mathbf{P}^k\mathbf{X} \odot \mathbf{W}^{(k)}) \tag{2.4.21}$$

In DCNN, the hidden representations $\mathbf{H}^{(k)}$ are not functions of previous hidden representations $\mathbf{H}^{(k-1)}$; instead, each hidden representation $\mathbf{H}^{(k)}$ is a function of the kth-hop transitions of the input matrix $\mathbf{X}$. The hidden representations $\mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(k)}$ are then concatenated at the end.

— GraphSAGE by Hamilton et al. [60] presents a generalized neighborhood aggregation framework, in which a fixed-size set of neighbors are sampled uniformly for each vertex, aggregated to form a representation, and then concatenated with the node features themselves and fed to a fully-connected layer $\mathbf{W}$ and a nonlinear activation. This formulation naturally fits into the MPNN framework, with UPDATE step defined as:

$$\begin{aligned}\mathbf{h}_{\mathcal{N}_S(\nu)}^{(k)} &= \mathrm{AGG}^{(k)}(\{\mathbf{h}_u^{(k)}, \forall u \in \mathcal{N}_S(\nu)\}) \\ \mathbf{h}_\nu^{(k)} &= \sigma\left(\mathbf{W}^{(k)} \cdot \left[\mathbf{h}_\nu^{(k-1)} \oplus \mathbf{h}_{\mathcal{N}_S(\nu)}^{(k)}\right]\right)\end{aligned} \tag{2.4.22}$$

where $\mathcal{N}_S(\nu)$ denotes the sampled neighborhood for vertex ν, and $\oplus$ denotes concatenation. The AGG operator can be implemented by different approaches. The authors propose applying an element-wise max-pooling:

$$\mathrm{AGG}_{\mathrm{pool}}^{(k)} = \max\left(\left\{\sigma\left(\mathbf{W}_{\mathrm{pool}}\mathbf{h}_u^{(k)}\right), \forall u \in \mathcal{N}_S(\nu)\right\}\right) \tag{2.4.23}$$

Alternatively, one can simply take the mean of the neighborhood and concatenate with the current state. In fact, by further simplifying Equation 2.4.22 by averaging the current state as well in the update step, we can recover a form similar to the spectral GCN convolution with standard normalization (Eq. 2.4.19) as opposed to symmetric normalization (Eq. 2.4.20):

$$\mathbf{h}_\nu^{(k)} = \sigma\left(\mathbf{W}^{(k)} \cdot \mathrm{mean}\left(\left\{\mathbf{h}_\nu^{k-1}\right\} \cup \left\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}_S(\nu)\right\}\right)\right) \tag{2.4.24}$$

2.4.3. Addressing oversmoothing in GCNs

Graph convolutions quickly became the norm in GNNs after their inception, as they proved both faster and more accurate than pre-convolutional GNNs which were formulated more akin to generalization of recurrent neural networks to graphs [12]. However, initial GCNs were not able to fully address the problem of GNN oversmoothing we mentioned in Section 2.3: Stacking many GCN layers mimics many iterations of message-passing, leading to the individual node representations in the graph converging to a certain value and becoming indistinguishable. This problem renders very deep GCN models ineffective; Chen et al. [20] find that most GCN models are most effective in "shallow" forms of as little as two convolutional layers. This however limits the ability of these models to extract information from higher-order neighborhoods.

ResGatedGCN by Bresson and Laurent [12] adds residual connections between convolution layers to handle oversmoothing, a trick borrowed from the successful ResNet [62] family of CNN architectures. They also incorporate *edge gates* to modulate the information coming from each edge. The edge gates consist of learnable parameters, and enable the layers to learn what edges are important for the graph learning task at hand; we will see that similar approaches are further explored via graph attention in the next section.

Chen et al. [20] present GCN II, a more recent model that improves upon the original GCN through two additional techniques to handle oversmoothing. Recall the original GCN layer (Equation 2.4.18):

$$\mathbf{H}^{(k)} = \sigma\left(\tilde{\mathbf{A}}_{\text{ns}}\mathbf{H}^{(k-1)}\mathbf{W}^{(k)}\right)$$

We have mentioned that stacking K standard graph convolutions is equivalent to forming a Kth-order polynomial filter. Wu et al. [154] show that the standard graph convolution with the renormalization trick (Eq. 2.4.17) acts as a low-pass filter that applies smoothing over the graph. Stacking GCN layers is then equivalent to K applications of the smoothing operation. The analogue of this low pass-filter in the image domain is the *blurring* operation; one can then see the oversmoothing problem arising from the repeated application of such filters more intuitively. The authors define the GCN II layer as follows:

$$\mathbf{H}^{(k)} = \sigma\left(\left((1-\alpha_k)\,\tilde{\mathbf{A}}_{\text{ns}}\mathbf{H}^{(k-1)} + \alpha_k\mathbf{H}^{(0)}\right)\left((1-\beta_k)\,\mathbf{I}_n + \beta_k\mathbf{W}^{(k)}\right)\right) \tag{2.4.25}$$

The augmentations of GCN II upon the original GCN are two-fold:

(1) The smoothed representation $\tilde{\mathbf{A}}_{\text{ns}}\mathbf{H}^{(k-1)}$ is combined with an initial residual connection to the first layer $\mathbf{H}^{(0)}$, where α_k modulates the strength of this residual connection. This residual connection is different from the one proposed in ResGatedGCN among others in that it is connects the current layer to the initial node representations, not a previous convolutional layer. This means that even after many message-passing iterations, the final node representations retain at least a fraction of their initial states.

(2) The weight matrix $\mathbf{W}^{(k)}$ is combined with the identity matrix $\mathbf{I}_n$ (with the strength of the connection modulated by $(1-\beta_k)$), an idea borrowed from ResNet [62] again. This allows the weight matrices $\mathbf{W}^{(k)}$ to have small norms, and essentially acts as a regularizer. This regularizer prevents node representations from quickly converging after repeated convolution, and hence alleviates oversmoothing.

Many approaches have been introduced to manage oversmoothing in GNNs, two of which we have covered in this section as they relate to the construction of our taxonomy later on. Nonetheless, building efficient and performant deep GCNs (by stacking tens of convolutional layers similar to large CNNs) remains an open problem and a limiting factor on applying these learning algorithms to large graphs.

2.5. Graph Attention and Graph Transformers

A more recent line of work has focused on improving neighborhood aggregation via the *attention* mechanism Bahdanau et al. [4]. To compare with other message construction approaches, note that GraphSAGE considers the information flowing into a vertex from each neighbor equivalent through averaging (Eq. 2.4.19), while the Kipf and Welling uses non-parametric, pre-determined values computed through symmetric normalization (Eq. 2.4.20).

Graph attention replaces these values by learnable *attention weights* measuring the importances between a node and each of its neighbors:

$$\mathbf{m}_{\mathcal{N}(\nu)} = \sum_{u \in \mathcal{N}(\nu)} \alpha_{\nu,u} \mathbf{h}_u, \tag{2.5.1}$$

The next state weight for a node ν is then computed as (compare with Eq. 2.3.4):

$$\mathbf{h}_\nu^{(k)} = \sigma\left(\sum_{u \in \mathcal{N}(\nu) \cup \{\nu\}} \alpha_{\nu,u}^{(k)} \mathbf{W}^{(k)} \mathbf{h}_u^{(k-1)} \right) \tag{2.5.2}$$

The attention weight between ν and some neighbor u is computed by concatenating the hidden states linearly transformed by the weight matrix $\mathbf{W}^{(k)}$, and multiplying the concatenated matrix with a trainable attention vector $\mathbf{a}$. The output is then passed through a non-linearity (LeakyReLU is used in practice) and then normalized via softmax across the vertex neighborhood:

$$\begin{aligned} \alpha_{\nu,u} &= \text{softmax}\left(\sigma\left(\mathbf{a}^\top \left[\mathbf{W}^{(k)} \mathbf{h}_\nu^{(k-1)} \oplus \mathbf{W}^{(k)} \mathbf{h}_u^{(k-1)} \right] \right) \right) \\ &= \frac{\exp\left(\sigma\left(\mathbf{a}^\top \left[\mathbf{W}^{(k)} \mathbf{h}_\nu^{(k-1)} \oplus \mathbf{W}^{(k)} \mathbf{h}_u^{(k-1)} \right] \right) \right)}{\sum_{v \in \mathcal{N}(\nu)} \exp\left(\sigma\left(\mathbf{a}^\top \left[\mathbf{W}^{(k)} \mathbf{h}_\nu^{(k-1)} \oplus \mathbf{W}^{(k)} \mathbf{h}_v^{(k-1)} \right] \right) \right)} \end{aligned} \tag{2.5.3}$$

Furthermore, the authors propose using *multi-headed attention* inspired by the Transformer model [147] for better stability in the learning process. In multi-headed attention, M independent attention mechanisms are used at a GAT layer, and the resulting features are concatenated or averaged in each iteration (note that we have omitted the iteration superscripts k for notational clarity):

$$\mathbf{h}_\nu = \oplus_{m=1}^{M} \sigma\left(\sum_{u \in \mathcal{N}(\nu) \cup \{\nu\}} \alpha_{\nu,u}^{(m)} \mathbf{W}^{(m)} \mathbf{h}_u \right) \tag{2.5.4}$$

$$\mathbf{h}_\nu = \sigma\left(\frac{1}{M} \sum_{m=1}^{M} \sum_{u \in \mathcal{N}(\nu) \cup \{\nu\}} \alpha_{\nu,u}^{(m)} \mathbf{W}^{(m)} \mathbf{h}_u \right) \tag{2.5.5}$$

Transformer models and multi-headed attention can be particularly useful for relaxing the constraints put forth by graph structure. The standard Transformer self-attention on a graph is in fact equivalent to a form of weighted message-passing between all pairs of vertices (i.e. a fully connected graph). This results in an increased expressive power and can capture relationships between far away or even disconnected nodes if such relationships exist, at the cost of quadratic complexity of $O(|\mathcal{V}|)$. Furthermore, treating the graph as fully-connected essentially destroys structural information embedded in the edge relationships which may make training more difficult for even large transformer models. The GAT, on the other hand, provides a trade-off by essentially limiting the attention computation to local neighborhoods.

More recent work by Yun et al. [162] and Dwivedi and Bresson [33] have improved upon the GAT model by employing the fully-connected graph view and extending it to edge features to arrive at a more generalized Graph Transformer (GT). To not lose structural information in the fully-connected view, Dwivedi and Bresson [33] propose encoding positional information as node features. This is not trivial, however, since since graphs are non-Euclidean and don't have a standard coordinate system to "position" the nodes. The solution is a generalization of "positional encodings" (PE) from the original Transformer paper [147]. The intuition behind PEs is that nodes far apart in the graph should have substantially different PEs, while closer nodes should have similar ones. We can use the eigendecomposition of the symmetric normalized graph Laplacian (Equation 2.4.3) for this. An n-node graph will have $n \times n$ Laplacian, and taking k eigenvectors after decomposition will yield an $n \times k$ matrix, where each row corresponds to a node in the arbitrary ordering. The low frequency (i.e. smaller corresponding eigenvalue magnitude) eigenvectors of the Laplacian are well-localized: We can then select the k eigenvectors from the lowest frequency ones, and assign each node the corresponding $1 \times k$ vector as their PEs as an additional input feature. This approach has proved very successful in practice in transformer-based methods, but several issues still persist and is a very active area of research. For a more detailed overview of the recent developments, we refer the reader to Dwivedi and Bresson [33], Lim et al. [90] and Rampášek et al. [113].

2.6. Graph Isomorphism and GNNs

The concept of graph isomorphism aims to capture whether two graph objects *have the same structure*, in informal terms.

Definition 2.6.1 (Graph isomorphism (unlabeled))**.** *Two graphs G and H are isomorphic if there is a bijection between the nodes of the two graphs*

$$f : \mathcal{V}_G \to \mathcal{V}_H$$

such that two vertices ν and u in G are adjacent if and only if $f(\nu)$ and $f(u)$ are adjacent in H:

$$\{\nu, u\} \in \mathcal{E}_G \implies \{f(\nu), f(u)\} \in \mathcal{E}_H$$

In almost all cases, though, our nodes have features to distinguish them, functioning as node labels. For such labeled graphs we need to consider that the mapping preserves the node features as well as graph structure. This brings us to our definition of graph isomorphism for labeled graphs.

Definition 2.6.2 (Graph isomorphism (labeled))**.** *Two graphs G and H, with corresponding adjacency matrices $\mathbf{A}_G$, $\mathbf{A}_H$ and feature matrices $\mathbf{X}_G$, $\mathbf{X}_H$ are isomorphic if and only if there*

exists a permutation matrix $\mathbf{P}$ *such that*

$$\mathbf{P}\mathbf{A}_G\mathbf{P}^\top = \mathbf{A}_H$$
$$\mathbf{P}\mathbf{X}_G = \mathbf{X}_H$$

While conceptually simple, testing for graph isomorphism is a very difficult problem from an algorithmic perspective: The optimization procedure involves searching over all possible permutation matrices $\mathbf{P}$, which has a computational complexity of $O(|\mathcal{V}|!)$. The problem is not NP-complete, yet there are no known solutions in polynomial time either, and is therefore commonly referred to as an NP-intermediate (NPI) problem. There exist algorithms to approximately test for graph isomorphism though, the most commonly used one being the Weisfeiler-Lehman (WL) algorithm.

In the context of graph learning, graph isomorphism proves to be particularly useful in quantifying the *representational capacity* of learning algorithms. An intuitive way to think about this is whether a given algorithm (e.g. a GNN) is able to determine whether two graphs are isomorphic, i.e. outputs the same representation $\mathbf{z}_G = \mathbf{z}_H$ if G and H are isomorphic. Ideally, an more expressive GNN will be able to distinguish some non-isomorphic pairs of graphs that a less expressive GNN will not be able to.

In light of this, a natural baseline for GNN expressivity becomes the established standard in approximate isomorphism testing, namely the 1-WL algorithm. Whether a GNN is *as good as* the 1-WL test is the most common baseline of GNN representative power, and the test itself has some interesting relationships with GNNs. Before we delve into that, however, let us review the 1-WL algorithm.

2.6.1. The Weisfieler-Lehman algorithm

The base and most common variant of the WL algorithm is the 1-WL algorithm; colloquially any reference to the algorithm without a prefix usually refers to the 1-WL version. We can summarize the algorithm as follows; a visualization of the algorithm is also available in Figure 2.5:

(1) Given two graphs G and H, assign a label to each node in each graph, $l_G^{(0)}(\nu)$ for $\nu \in \mathcal{V}_G$ and likewise $l_H^{(0)}(\nu)$ for $u \in \mathcal{V}_H$. In many graphs, the node degree is used as the labels, but node features can be used as well.

(2) Assign a new label to each node iteratively by hashing a multi-set that is composed of (a) the current node label, and (b) the set of labels of its neighbors. W.l.o.g. for G, we have

$$l_G^{(i)}(\nu) = \text{HASH}\left(l_G^{(i-1)}(\nu), \left\{\left\{l_G^{(i-1)}(u) \quad \forall u \in \mathcal{N}(\nu)\right\}\right\}\right) \tag{2.6.1}$$

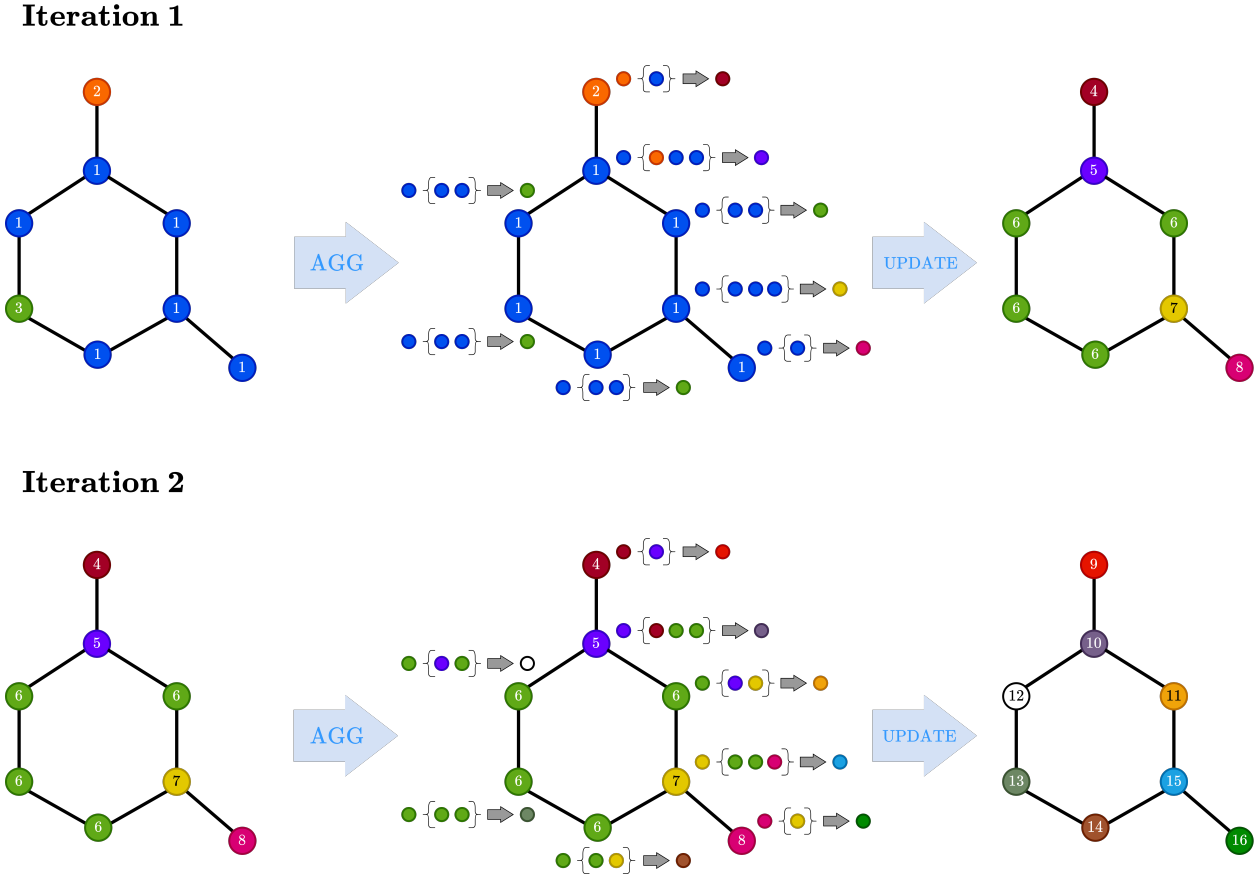


Figure 2.5 – Two iterations of the WL test on a graph. The AGG step aggregates the neighbor labels for each vertex with its own label, and the UPDATE step hashes the resulting multiset to a new value. The grey arrow represents the hash operation. Adapted from an example from Sato [125].

One can note the similarity this step bears to the message-passing step in MPNNs. The double-brace notation indicates a multi-set, and HASH maps the multi-set to a new label.

(3) Repeat step (2) for both graphs until some iteration K at which the labels for all vertices converge:

$$l_G^{(K)}(v) = l_G^{(K-1)}(v) \forall v \in \mathcal{V}_G \qquad l_H^{(K)}(u) = l_H^{(K-1)}(u) \forall u \in \mathcal{V}_H$$

Alternatively, the algorithm can terminate early if the resulting multi-sets are not equivalent at any point, meaning $WL(G) \neq WL(H)$ and the graphs not isomorphic.

(4) Finally, collect all node labels for each graph in multi-sets:

$$L_G = \left\{\left\{l_G^{(K)}(v) \forall v \in \mathcal{V}_G\right\}\right\} \qquad L_H = \left\{\left\{l_H^{(i-1)}(u) \forall u \in \mathcal{V}_H\right\}\right\}$$

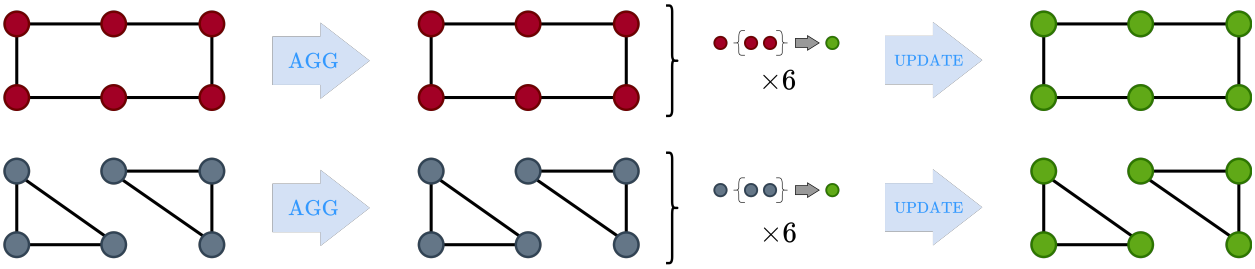


Figure 2.6 – A canonical example in which the WL test fails: The test cannot distinguish that these graphs are non-isomorphic.

> If the final multi-sets are equivalent, we have $WL(G) = WL(H)$ implying graph isomorphism between G and H.

The WL algorithm is highly efficient with convergence guarantees in $|\mathcal{V}|$ iterations, and guarantees that a pair of graphs are *not* isomorphic when the test fails. It also can guarantee isomorphism for certain families of graphs. However, there are well-known pairs of graphs where the WL test fails, one canonical example is the hexagon-versus-two-triangles shown in Figure 2.6. One thing to note is that the WL test failing here relies on the node labels being identical; the test does fail in the case labels are derived from node degrees, for example, but will succeed if they employ distinct node features.

2.6.2. The WL algorithm and GNNs

Two lines of work in 2019, namely Morris et al. [99] and Xu et al. [159], focused on formalizing the relationship between the 1-WL test and GNNs, based on the MPNN framework with the AGG and UPDATE operators introduced in Equation 2.2.1. Both papers arrived at the same two results through their analysis, which can be summarized as follows:

(1) Message-passing GNN architectures are *at most* as powerful as the WL test in distinguishing between non-isomorphic graphs.

(2) Under certain conditions, there exist message-passing GNNs *as powerful as* the WL test.

The following theorem from Xu et al. [159] puts forth the first point:

Theorem 2.6.3. *Let G_1 and G_2 be any two non-isomorphic graphs. If a graph neural network $\mathcal{A} : \mathcal{G} \to \mathbb{R}^d$ maps G_1 and G_2 to different embeddings, the WL isomorphism test also decides that they are non-isomorphic.*

We can provide a proof sketch as follows, and delegate the full proof to the original paper: The WL test always hashes different multi-sets of neighboring nodes into different labels, and the same multi-sets to the same label. The GNN is also guaranteed to map the same inputs to the same outputs at any iteration. This creates a valid mapping ϕ for the WL test and GNN states operating on any vertex $\nu \in G$ at some iteration i: $h_\nu^{(i)} = \phi(l_\nu^{(i)})$. Then, if

two graphs G_1 and G_2 have the same multi-set of WL neighborhood labels, they will also have identical sets of GNN neighborhood features, i.e. the same next states for each vertex. Assuming we use a permutation invariant graph-level readout function on the final states of the graphs to determine non-isomorphism, $\mathrm{WL}(G_1) = \mathrm{WL}(G_2) \implies \mathcal{A}(G_1) = \mathcal{A}(G_2)$.

This is followed by the second theorem, which establishes conditions for which an MPNN is as powerful as the WL test:

Theorem 2.6.4. *Let $\mathcal{A} : \mathcal{G} \to \mathbb{R}^d$ be a GNN. $\mathcal{A}$ maps any two graphs G_1 and G_2 that the WL test determines as non-isomorphic to different embeddings, if the following conditions hold:*

(1) The AGG *and* UPDATE *functions in the iterative update rule*

$$\mathbf{h}_\nu^{(k+1)} = \mathrm{UPDATE}^k\left(\mathbf{h}_\nu^{(k)}, \mathrm{AGG}^{(k)}(\{\mathbf{h}_u^{(k)}, \forall u \in \mathcal{N}(\nu)\})\right)$$

are injective*, i.e. they map distinct inputs to distinct outputs: $f(x_1) = f(x_2) \implies x_1 = x_2$.*

(2) $\mathcal{A}$'s graph-level readout operating on final states (in the form of multi-sets of node features) is injective.

Proof. Consider a GNN $\mathcal{A}$, with injective aggregation and update functions f and ϕ. Since the graph-level readout function is also injective, it is sufficient to show that $\mathcal{A}$'s iterative aggregation process results in different embeddings of node features at iteration k, $\mathbf{h}_\nu^{(k)}$. The WL test also applies an injective hash function g to update the node labels $l_\nu^{(k)}$:

$$l_\nu^{(k+1)} = g\left(l_\nu^{(k)}, \{l_u^{(k)}, \forall u \in \mathcal{N}(\nu)\}\right)$$

The proof shows through induction that for any iteration k, there exists an injective function φ such that $\mathbf{h}_\nu^{(k)} = \varphi\left(l_\nu^{(k)}\right)$. For $k = 0$, this already holds since the initial features are the same: $\mathbf{h}_\nu^{(0)} = l_\nu^{(0)}$.

Assuming this holds for iteration k, let us show that it holds for $k+1$. Substituting $\mathbf{h}_\nu^k = \varphi\left(l_v^k\right)$ gives us

$$\mathbf{h}_\nu^{(k+1)} = \phi\left(\varphi\left(l_\nu^{(k)}\right), f(\{\varphi\left(l_u^{(k)}\right), \forall u \in \mathcal{N}(\nu)\})\right)$$

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

$$l_\nu^{(k+1)} = \psi\left(l_\nu^{(k)}, \{l_u^{(k)}, \forall u \in \mathcal{N}(\nu)\}\right)$$

This gives us

$$\mathbf{h}_\nu^{(k+1)} = \psi \circ g^{-1} g\left(l_\nu^{(k)}, \{l_u^{(k)}, \forall u \in \mathcal{N}(\nu)\}\right) = \psi \circ g^{-1}\left(l_\nu^{(}k)\right)$$

where $\varphi = \psi \circ g^{-1}$ is injective as the composition of injective functions.

Recalling $\mathbf{h}_\nu^{(k)} = \varphi\left(l_\nu^{(k)}\right)$, consider two non-isomorphic graphs G_1 and G_2. If at some iteration K, the WL test distinguishes them as non-isomorphic, i.e. recognizes that the

multi-sets $\left\{l_\nu^{(K)}\right\}$ are different for G_1 and G_2. Since φ is injective, then we are guaranteed that the embeddings $\left\{\mathbf{h}_\nu^{(K)}\right\} = \left\{\varphi\left(l_\nu^{(K)}\right)\right\}$ are also different for G_1 and G_2. □

A related result from Morris et al. [99] shows that the MPNN formulation in Equation 2.3.3 is sufficient to match the 1-WL test, if the initial node features of the graphs are linearly independent. They also extend GNNs to match the k-WL test, an extension of the 1-WL test that is based on distinguishing graphs on tuples of vertices; this is much less common in practice so we deem both the k-WL test and k-GNNs out of scope, though a comprehensive review is provided in Morris et al. [99] for the interested reader.

The general problem with most GNN algorithms we have covered so far is that they are not guaranteed to be as expressive as the WL algorithm. Consider the spectral GCN; the spectral GCN aggregation rule is defined by the symmetric normalization we introduced in Equation 2.4.20, which is *not* an injective function. Similarly, the element-wise max-pooling in GraphSAGE aggregation (Eq. 2.4.22) is not injective.

Xu et al. [159] propose the Graph Isomorphism Network (GIN) which provably satisfies the conditions in 2.6.4. The theoretical guarantees of GIN rely on the fact that sum aggregation can represent injective (and in fact *universal*) functions over *multi-sets*:

Lemma 2.6.5. *Assume $\mathcal{X}$ is countable. There exists a function $f : \mathcal{X} \to \mathbb{R}^n$ such that $h(X) = \sum_{x \in X} f(x)$ is unique for each multi-set $X \subset \mathcal{X}$ of bounded size. Moreover, any multi-set function g can be decomposed as $g(X) = \phi\left(\sum_{x \in X} f(x)\right)$ for some function ϕ.*

Proof. Since the multi-set $\mathcal{X}$ is countable, there exists a mapping $Z : \mathcal{X} \to \mathbb{N}$ from $x \in \mathcal{X}$ to natural numbers. Since the cardinality of multi-sets of X is also bounded, there exists a number $N \in \mathbb{N} > |X| \quad \forall X$. Based on this information, one example of such a function is $f(x) = N^{-Z(x)}$, which provides an $h(X) = \sum_{x \in X} f(x)$ that is injective over multi-sets.

Any multi-set function g is required to be permutation invariant (since it operates on multi-sets, which are permutation invariant by construction) to be valid; $\phi\left(\sum_{x \in X} f(x)\right)$ is also permutation invariant by default since the inner sum is permutation invariant. Since $h(x) = \sum_{x \in X} f(x)$ is injective as well, any multi-set function g can be decomposed as $g(X) = \phi\left(\sum_{x \in X} f(x)\right)$. □

On the other hand, injective *set* functions (such as mean/max aggregation) are not injective when applied to multi-sets, leading to the deficiencies in GCN and GraphSAGE we have shown. Based on this lemma, the authors provide the following corollary:

Corollary 2.6.6. *Assume $\mathcal{X}$ is countable. There exists a function $f : \mathcal{X} \to \mathbb{R}^n$ such that for infinitely many choices of ϵ, including all irrational numbers, $h(c, X) = (1 + \epsilon) \cdot f(c) + \sum_{x \in X} f(x)$ is unique for each pair (c, X), where $c \in \mathcal{X}$ and $X \subset \mathcal{X}$ is a multi-set of bounded size. Moreover, any function g over such pairs can be decomposed as $g(c, X) = \varphi((1 + \epsilon) \cdot f(c) + \sum_{x \in X} f(x))$ for some function φ.*

PROOF. Consider $f(x) = N^{-Z(x)}$ from our proof of Lemma 2.6.5. Let $h(c, X) \equiv (1+\epsilon)\cdot f(c) + \sum_{x\epsilon X} f(x)$. We aim to show that for any $(c, X) \neq (c', X')$, we have $h(c, X) \neq h(c', X')$ where $c, c' \in \mathcal{X}$ and $X, X' \subset \mathcal{X}$, and ϵ is an irrational number.

We prove this by contradiction. We consider two cases:

(1) $c = c'$, $X \neq X'$. In this case, $h(c, X) = h(c', X') \implies \sum_{x\epsilon X} f(x) = \sum_{x\epsilon X'} f(x)$, which is not possible by Lemma 2.6.5 since $\sum_{x\epsilon X'} f(x)$ is injective; we have reached a contradiction.

(2) $c \neq c'$. We can rewrite the inequality as:

$$(1+\epsilon)\cdot f(c) + \sum_{x\epsilon X} f(x) = (1+\epsilon)\cdot f(c') + \sum_{x\epsilon X'} f(x) \tag{2.6.2}$$

$$\epsilon\cdot(f(c) - f(c')) = \left(f(c') + \sum_{x\epsilon X'} f(x)\right) - \left(f(c) + \sum_{x\epsilon X} f(x)\right) \tag{2.6.3}$$

Since $(f(c) - f(c'))$ is a non-zero rational number and ϵ is irrational, the L.H.S of Equation 2.6.3 is irrational. However, the R.H.S. of Equation 2.6.3 is rational as sums of rational numbers. Therefore, we have reached a contradiction.

For any function g over the pairs (c, X), φ can be constructed such that $g(c, X) = \varphi((1+\epsilon)\cdot f(c) + \sum_{x\epsilon X} f(x))$; φ is well-defined as $h(c, X) = (1+\epsilon)\cdot f(c) + \sum_{x\epsilon X} f(x)$ is injective. □

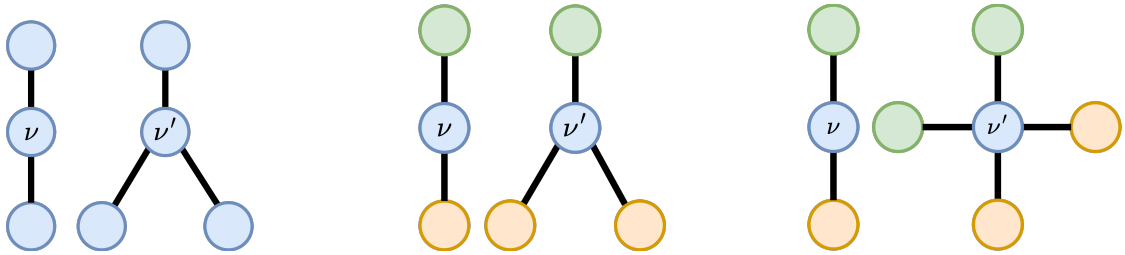
Xu et al. [159] then suggest using MLPs to model f and φ, since they can be used to approximate such function classes as proven by the universal approximation theorem [67]. As composition of injective functions are also injective, a single MLP is sufficient to model the composition of the two functions: $f^{(k+1)} \circ \varphi^{(k)}$. Using a learnable parameter or some scalar as ϵ, the GIN update rule becomes:

$$\mathbf{h}_\nu^{(k+1)} = \text{MLP}^{(k)}\left(\left(1+\epsilon^{(k)}\right)\cdot \mathbf{h}_\nu^{(k)} + \sum_{u\epsilon\mathcal{N}(\nu)} \mathbf{h}_u^{(k)}\right) \tag{2.6.4}$$

GIN now serves as one of the most reliable GNN models in benchmarking due to its provably better expressivity over other benchmark GNNs. We have briefly discussed the shortcomings of GCN and GraphSAGE over GIN, but it is still helpful to provide some pointers on *how* such models come short.

— As mentioned previously, the mean or max-pooling of neighborhood features we see in other GNN models are not injective over multi-sets. The authors provide several pairs of graph structures that mean and/or max-pooling aggregation cannot distinguish but summation can, shown in Figure 2.7. The general trend is that a neighborhood representation fails when there are repeating node features.

— Many GNN approaches we discussed use a linear mapping $\mathbf{W}$ followed by a nonlinearity σ and possibly no bias, i.e. a 1-layer-perceptron; the original GNN (Eq. 2.3.4),



(a) Mean and max fail **(b)** Max fails **(c)** Mean and max fail

Figure 2.7 – Thee pairs of graphs that mean and/or max-pooling cannot distinguish, where colors indicate different node features. Between the two graphs, nodes v and v' get the same embedding even though their corresponding graph structures differ. Figure adapted from Xu et al. [159].

GCN (Eq. 2.4.18), DCNN (Eq. 2.4.21), GraphSAGE (Eq. 2.4.22) are all examples of such networks. The authors prove Lemma 2.6.7 (proof delegated to the original paper), which indicates that there exists graph neighborhoods (represented by multi-sets) that these 1-layer perceptrons cannot distinguish; The intuition is that the 1-layer perceptron is not a universal approximator of multi-set functions.

Lemma 2.6.7. *There exists finite multi-sets $X_1 \neq X_2$ such that for any linear mapping* $\mathbf{W}$*,*

$$\sum_{x \in X_1} \text{ReLU}(\mathbf{W}x) = \sum_{x \in X_2} \text{ReLU}(\mathbf{W}x)$$

This sums up our discussion of the development of GNN models from a variety of approaches. We will now proceed with an overview of graph data, and survey attempts and current work on benchmarking in graph learning.

Chapter 3

Graph Datasets and Benchmarking: A Survey

3.1. Motivation for Benchmarking

As with any other machine learning domain, data form the foundations of graph learning research; graph learning as a field is as useful as its predictive power on graph data. In consideration of graph learning and machine learning in general, as a scientific field, the process of benchmarking plays an important role.

Benchmarking in ML is the practice of comparing the predictive performance of ML algorithms with state-of-the-art (SOTA) over established datasets (called "benchmarks"). Each benchmark typically consists of a dataset, along with associated task(s) (e.g. sentiment analysis, node classification, image recognition) and quantitative metrics. Improvements over benchmarks is considered a noisy but reliable indicator of progress in a given line of ML research, subfield or even the field itself, thus rendering it as the *de facto* paradigm for scientific ML research [80, 128, 112].

Benchmarking is also essential to compare the current success and future potential of distinct lines of research. For example, the success of artificial neural networks (ANN) over kernel learning through hand-crafted features or regression models happened long after their invention. Despite their theoretical capabilities, interest in neural networks were limited for almost half a century due to computational limitations. Only when these computational capabilities were partly overcome in the 2000s their theoretical abilities were validated "in the wild" through surpassing competing approaches in benchmark tests in a variety of fields (e.g. speech recognition, image classification, sequence prediction) the interest in neural networks were revived. A more recent example is the success of attention-based models in sequence or image-based tasks. Attention mechanisms are used for more than a decade under a variety of names, but have gained immense popularity only in the last few years when the Transformer architecture matched or even surpassed RNN and CNN-based models in respective benchmarks in natural language processing (NLP) and computer vision.

In short, benchmarks in ML have a much wider impact on the field itself than measuring the performance of algorithms; they shape the trajectory of machine learning field itself by determining how different learning paradigms are viewed, validated or rejected in both research and industry. This on its own is a sufficient argument for the construction of benchmarking procedures that are generalizable, fair and robust. Whether we as the machine learning community have achieved or even have striven for these objectives, however, is highly debatable.

3.2. State of ML Benchmarking: An Overview

There is little doubt that the benchmarking procedures in machine learning "work" in a broad sense, since the rapid progress in ML we have seen in the last decade does depend on our ability to set appropriate benchmarks and improve upon them. It is very difficult if not outright impossible to argue that machine learning and in particular deep learning has *not* been progressing: Putting aside any discussion of the epistemological merits of the field, deep learning has redefined the state-of-the-art in the wild (and beyond the scope of benchmarking data) across almost all data-driven domains; in most cases multiple times through algorithms that learn better, scale better and generalize better in each iteration, with solid theoretical and empirical justifications. Such progress in a short span of about a mere decade indicates that we are *probably approximately correct* – we must be doing at least some things right in how we define the SOTA across multiple domains and improving upon them.

Nevertheless, benchmarking is an inherently flawed process. Benchmark datasets are meant to be representative of their data domain, but it is impossible to capture the complexity of data and problems spanning a domain to just a few datasets. Benchmarks therefore are meant to be good approximators instead, with the aim of maximal alignment with the "real world". For a set of benchmarks to be good approximators, we conjecture the following criteria:

— **Reliability:** Benchmark datasets must be reliable in that they are derived from credible and verifiable sources, and not subject to tampering to knowingly induce data biases.

— **Accuracy:** Benchmark datasets must accurately capture the information, relationships and distributions they aim to represent. This implicitly justifies the emergence of different subfields of machine learning as different types of data (i.e. text, image, graph) capture different information. Graphs are essentially relational models, for example, as a result relational data (e.g. social networks, molecular data in cheminformatics and biochemistry, citation graphs) make more suitable graph learning benchmarks.

— **Fairness:** Benchmark datasets need to align well with objectives of "ethical ML & AI", in particular concerning ML research on fields with high social impact. "Unfair" datasets may lead to skewed interpretations of data and render models unsafe for deployment in the wild.
— **Data Coverage:** The set of benchmarks must be selected in order to maximally cover the *data and task space*. Selecting multiple datasets that cover overlapping data and task space leads to quickly diminishing marginal returns, as they all test for the same elements of a given ML algorithm and naturally reach the same conclusions. Benchmarking with little coverage may erroneously highlight algorithms that *overfit* to small regions of the data space as SOTA.

Verifying the reliability and accuracy of datasets is a well-established process and is usually done manually. Most high-quality datasets are released with extensive information on how they are sourced and designed, which can be independently verified to ensure reliability and accuracy. These are later on empirically validated through usage; the datasets that do not meet the criteria tend to gradually go out of use. This is not to say these issues are fully resolved; dataset documentation and standardization is often left to the incentive of the researchers to self-enforce, leading to questionable practices far too common to ignore. There is however significant past and ongoing work to address these issues [49, 127, 48].

Evaluating the fairness of datasets is a more complex problem, but is a very active area of research. ML & AI ethics have branched into its own field; multiple lines of work are dedicated to evaluate datasets on whether they induce hidden biases that favour or hurt communities, or lead to models that do not generalize in the wild. The development of ML & AI ethics has in turn triggered an increased and justified scrutiny on ML datasets. Various AI fairness studies have covered data bias on race and gender [18, 46, 164], disabilities [73] and geodiversity [132] among others, particularly in text [46, 30, 164, 73] and image [132, 25] domains. Expansive surveys on dataset biases and applications in the real world are provided by Mehrabi et al. [94] and Paullada et al. [108].

Data coverage of datasets, on the other hand, is much more difficult to evaluate. We therefore dedicate this next subsection to investigating the nature of data coverage, and how it relates to the state of benchmarking in graph learning.

3.2.1. Data space and coverage

In this thesis, we consider an abstract "data and task space" over which we define the notion of coverage. We should therefore first present a brief (and mostly informal) discussion of what we mean by these terms, as they will be fundamental to our reasoning and discussions in the coming sections. Our construction of the data and task space is analogous to the concepts of *GNN design/task space* in You et al. [161], only applied to graph datasets here

instead of GNN models. Additionally, a similar notion of *benchmark dataset space* is also used in Palowitch et al. [106] where data dimensions are constructed using graph metrics. The reader is therefore referred to those works as well for a more holistic discussion of these concepts.

Task space. Task space is an *abstract space* composed of the Cartesian product of the dimensions (where the term *dimension* refers to a quantifiable attribute of an object) over which we define graph learning problems. The task space in graph learning is well-established; It basically represents the space of different prediction "tasks" we may want to perform over a given graph. Drawing from existing literature, we can think of two main dimensions over which the task space is built: (a) Prediction type (e.g. classification vs. regression tasks), and (b) Prediction level, i.e. what type of structure we are making a prediction about (node, edge or graph-level). A more detailed discussion on graph tasks is also available in Section 1.2.3.

Data space. A graph object can also be thought of as being composed of multiple dimensions: Number of nodes (i.e. size), graph density, longest path, symmetry, existence of node features, *types* of node features, application domain are all examples of dimensions upon which a graph can be defined. The Cartesian product of these dimensions can then be thought of as a hyperspace over which graphs (or aggregations of them in the form of datasets) can be defined. Unlike task space, the dimensions of data space are not well-established in the literature, and remain somewhat abstract.

Note that by this definition, the dimensions of the data space can be arbitrarily large, making complete coverage over this abstract hyperspace intractable using real-world datasets. However, we can pick and choose dimensions that are *causally relevant* towards our goal of quantifying GNN performance: A data dimension is causally relevant with respect to a task if changing a graph along that dimension "causes" GNN models to behave differently over them. The name of a graph dataset for example is completely irrelevant to GNN performance over said dataset; we can therefore safely assume "dataset name" is not a relevant dimension. On the other hand, the existence/lack of node features is likely to be a relevant dimension: A GNN will probably behave differently on a pair of graphs that share the same graph topology, but with only one possessing meaningful node features. In turn, we can think of the data space as the set of dimensions over which GNNs behave differently, i.e. those that are causally relevant.

Ideally, a set of benchmark datasets need to have good coverage over causally relevant dimensions in order to test the performance of a GNN maximally. However, we will see that the challenge here lies not necessarily in selecting/creating datasets to cover these dimensions,

but rather in defining and constructing dimensions that are causally relevant. Evaluating data space coverage of benchmarks is particularly difficult for a number of reasons:

(1) Unlike previous criteria, we do not have established metrics or methods to evaluate data coverage. To compare, even if mostly through empirical observations, researchers are able to measure the reliability and accuracy of individual datasets accurately as explained above. Similarly, there is an ever-growing body of work on how to detect and address problems with data fairness. Data coverage, on the other hand, is defined over an abstract "data and task space" that is not well-defined.

(2) Data coverage cannot be evaluated for individual datasets as each dataset constitutes a single data point in the data space. Rather, coverage is a measure of to what extent a *collection* of datasets (that are selected as benchmarks) can represent the variety of data in the domain. This requires each dataset to cover distinct areas of the data and task space, and careful consideration of the interactions and similarities between distinct datasets is required.

Due to the difficulty of appropriate evaluation procedures and lack of standardized methods or metrics, the notion of data coverage has often been overlooked in the selection of benchmarks until recently.

Lack of consideration for data coverage in the selection of benchmarks is closely related to many of the problems that plague machine learning today. Addressing these benchmarking problems is of utmost importance for a number of reasons. Koch et al. [80] presents an insightful dissection of this importance of setting good benchmarks: Firstly, establishing appropriate benchmarks pave the way to safe and effective machine learning models, particularly in mission-critical applications. They also lead to a more accurate measurement of scientific process in the field. Furthermore, they heavily influence the behavior of researchers and industry practitioners. Since benchmark performance is accepted as an indicator of progress, researchers and practitioners tend to align their work to maximize their performance over these benchmarks. A misalignment of the benchmarks with the real world means the models that maximize performance over benchmarks are suboptimal in the wild, compromising the goals laid out in the previous statement and misdirecting the field of machine learning in general.

To gain a better understanding of these problems before we address them, we will first inspect coverage issues in computer vision benchmarking. As a historically more mature but arguably similar domain compared to geometric deep learning, a vaster literature is available on benchmarking issues in CV, serving as a useful guide to analogous problems in graph learning.

3.2.2. Data coverage in computer vision benchmarking

Appropriate coverage of the data space has garnered considerable attention in recent years within the field of computer vision, particularly in the context of transfer learning [32, 114]. Transfer learning is arguably the current norm in deep learning on image and text data. In transfer learning, researchers rely on models that have been extensively trained on large datasets, usually released by technology industry giants with access to large amounts of computing power. Researchers then fine-tune these "pretrained" models on their own data, leveraging the features learned by the pretrained models to learn quicker attain desired metrics faster, with relatively small compute effort.

Nevertheless, there is a recent line of questioning on whether these pretrained models harm generalization to datasets with substantially different distributions due to the biases induced by the datasets used in pretraining. In image domains, the largest and best-known such dataset is ImageNet [28, 123], which has set the standard in CV in transfer learning, image classification, object detection [71] and image segmentation [63, 19] since its inception in 2009 due to its sheer size of >1 million images covering 1000 object classes.

The benefits of curation and release of such large datasets and pretrained models are plenty; they make machine learning research more accessible to the ML community and individual researchers by delegating the cost of both (a) collection and annotation of massive datasets and (b) pretraining of large models on these datasets to well-endowed institutions, and significantly increasing the development speed and rate of discovery in ML research.

There is no free lunch, though [153], and these benefits come at a cost. The size, availability and industry support behind these large datasets establish them as benchmarks in rapid fashion, to the point that the evaluation procedures for whole ML subfields become reliant on them, as in the case of ImageNet. This reliance in turn confines ML research into a narrow focus of attaining maximal performance on a narrow set of benchmarks, at the cost of discarding more comprehensive evaluation processes that take generalization and data biases into account. This may induce biases in the form of *architectural overfitting*, where model architecture design decisions are adapted to *fit* the datasets they are tested on, invalidating these datasets as reliable benchmarks and possibly hurting model generalization [106, 115]. D'Amour et al. [26] argue that most generalization issues in deep learning can be attributed to *underspecification*: In the limited scope of evaluation on a few benchmarks, there are many different solutions (e.g. different weight configurations of a neural network) that will perform well over them. However, when these models are released into the real world, they may behave erratically when the data encountered do not resemble the distributions the model was evaluated on. The authors support these claims by an extensive empirical study that explores underspecification with examples drawn from computer vision, medical imaging, NLP and electronic health data. Torralba and Efros [142] measure cross-dataset

generalization between six well-known benchmarking image datasets, including ImageNet, Caltech-101 [37] and PASCAL VOC [36] by training a classifier on one dataset and testing it on the other five. Their results show dramatic drops in accuracy for distinct pairs of datasets, indicating significant low generalization due to dataset bias.

ImageNet is subject to particular scrutiny: Tsipras et al. [143] show that labeling processes in ImageNet (such as an image with multiple objects having a single class label) introduces systematic errors which propagates into the models pretrained on them, potentially hurting generalization performance. Recht et al. [116] test generalization of ImageNet-trained models by replicating the dataset creation process of the original dataset to produce ImageNet-v2 and find out accuracy drops of more than 10%; they surprisingly also find that models that perform better on ImageNet also perform better on ImageNet-v2, implying ImageNet-trained models may not be extremely overfitting even though they cannot account for the remarkable drops in performance. Engstrom et al. [35] in turn claim the drops in performance is caused by statistical bias introduced in the creation of ImageNet-v2. In a similar study, Kornblith et al. [83] test whether models that perform better on ImageNet perform better on transfer learning tasks. They find that better model architectures are also better for transfer learning, but many of the reguralization techniques used to improve ImageNet performance (such as label smoothing [139] and dropout [136]) are detrimental for transfer learning. Beyer et al. [7] propose an improved labeling framework for ImageNet to fix the label biases, and find that more recent SOTA models are overfitting to the label biases in ImageNet, and in turn are less generalizable.

This amount of auditing for a *single* benchmark dataset underlines the fact that *there is no perfect benchmark*. More importantly, these studies warn us of the dangers of low data coverage in benchmarking by relying in one or few benchmark datasets in order to define SOTA. Unfortunately, convergence to a few large datasets as benchmarks is the current trend in ML: Koch et al. [80] and Barbosa-Silva et al. [5] find that despite a steady increase in the number of available datasets, ML research communities in CV and NLP are becoming increasingly concentrated on a tiny fraction of these datasets over time for benchmarking, most of which are top-down introduced by a handful of institutions. Furthermore, many of these datasets are originally created for a different task than they are used for (e.g. object detection vs. image classification), likely introducing hidden biases in the adoption process.

As we have stated before, benchmarking is a flawed process; every dataset used for benchmarking will inevitably have certain data defects. However, it is possible to mitigate the effects of these flaws on the ML benchmarking process through increased data coverage. Testing model performance on a set of sufficiently numerous and distinct datasets makes it possible to better approximate how these models will behave in the real world.

3.3. Benchmarking in Graph Learning

Historically, graph representation learning has in particular suffered from reliance on a few datasets for benchmarking. This reliance arguably has different underlying reasons: Primarily, graph learning is a younger domain than its text and image-based counterparts, and therefore the associated body of work, including work on datasets, is comparatively less.

Another important distinction is that graph data creation is a very arduous task. Image and text data is commonplace in the wild and data gathering is usually simpler; more effort goes into the annotation process which is usually delegated to crowdsourcing services. Graph data, on the other hand, is a representational and abstract form of data, and hence is not encountered in the wild. Graph data is usually either (a) created manually by application domain experts (e.g. bio/chemical graphs), or relies on specialized tools to extract relationships between objects (e.g. crawlers for web graphs).

The citation datasets Cora [92], CiteSeer [50] and Pubmed [130] have been the go-to benchmarks since the inception of graph learning. All three are transductive node-level classification datasets (each sample is concerned with classifying an individual vertex in a single shared graph). For inductive graph-level classification, Yanardag and Vishwanathan [160] have proposed numerous datasets drawn from bioinformatics, social networks and collaboration networks, which have been established as benchmarks since. Morris et al. [100] later on incorporated these into the TUDatasets framework, which is a popular graph dataset collection preferred for its ease of use.

However, recent studies in re-evaluating GNN benchmarking by Dwivedi et al. [34] and Hu et al. [68] have shown that these datasets are insufficient to serve as standalone benchmarks. Firstly, the small sizes of a majority of these datasets have proven to be a limitation: The largest node-level classification dataset (Pubmed) consists of ~20,000 nodes; in real-life applications such as GNNs on social networks, the constructed graphs involve *millions* of nodes.

For graph-level tasks, we encounter even more extreme examples: In TUDatasets, IMDB-BINARY consists of 1,000 graphs with an average of 20 nodes per graph. Bioinformatics datasets on average also have ~1,000 graphs per dataset (with MUTAG consisting of just 188 (!) graphs) and about 30 nodes/graph. This is not to invalidate these datasets: after all, molecular graphs are not bound to have more than a few hundred nodes at most, so many of these do resemble distributions of their application domains. However, the development of models almost exclusively on these small datasets rendered them less generalizable and less scalable to larger graphs due to the induced architectural overfitting [34, 68].

In recent years, several new node-level datasets have emerged to address the small-size bias in graph data. For node-level prediction, PPI, Reddit [58] and Amazon2M [22] have been proposed, all of which are significantly larger than the "standard" node-level classification

datasets. However, it was noted that these datasets have comparatively small test sets that lead to artificially small generalization gaps [68]. For graph-level tasks, recent studies [70, 76] have started incorporating large-scale cheminformatics datasets from MoleculeNet [157].

Secondly, these benchmarks do not have standardized experimental protocols, meaning every paper that uses these datasets set their own train/test splits and cross-validation parameters. This lack of uniformity renders direct comparison of models on the same dataset difficult due to the discrepancies in the respective benchmarking protocols. The large datasets that were brought on to alleviate size issues in graph learning also lacked standardized evaluation methods, leaving this problem unresolved.

Finally, these datasets also induce data biases which do not stem from graph size: Ivanov et al. [75] show that many graph datasets (including some we have mentioned so far) suffer from *isomorphism bias*, where datasets have high percentages of isomorphic examples corresponding to different target classes; models can then incorporate additional structures to check for isomorphism that increases benchmark performance but do not translate to better generalization. Furthermore, node and link-level prediction datasets usually consist of sparse graphs drawn from recommendation, citation and social domains, while many bio/cheminformatics graphs (e.g. protein or drug pair interaction graphs) the graphs are very dense. This domain-specific graph structures may induce architectural overfitting and biases that hurt generalization.

The reader may notice that the issues we have covered are closely related to the problem of coverage, apart from those concerning standardization of experiment protocols. We have also demonstrated that coverage may be composed of multiple dimensions: graph size, task type, application domain, graph density etc. Indeed, the studies by Dwivedi et al. [34] and Hu et al. [68] are the first to actively address data coverage in graph learning.

In order to address these problems, Hu et al. [68] released Open Graph Benchmark (OGB), a codebase for benchmarking with a diverse set of datasets focused on scalability, robustness and reproducibility in graph learning. They tackle data coverage from three dimensions:

— Graph size: Most pre-OGB graph benchmarks are small-to-medium sized; OGB includes large graphs with >100 million nodes to cover a large spectrum of graph sizes.

— Application domain: OGB broadly categorizes graphs into three domains: *nature*, *society* and *information*. Additionally, each dataset is assigned domain-specific data splits that are more appropriate than random splitting.

— Task categories: In addition to data diversity, OGB takes into account task diversity, and covers node, edge and graph-level tasks.

The authors then conduct benchmarking experiments and ablation studies on these datasets using a variety of graph learning models, with a focus on scalability and *out-of-distribution* generalization.

Dwivedi et al. [34] take a different approach and define *appropriateness* (i.e. data coverage) of a set of datasets through their ability to statistically separate the performance of GNNs. The "usual" benchmarks of Cora, CiteSeer and TUDatasets as deemed not appropriate as most GNNs perform "almost statistically the same" on them. In turn, they propose seven medium-to-small scale datasets that cover multiple tasks and application domains, including synthetic (algorithmically generated) datasets. They proceed with benchmarking several GNN models and present their findings on how these GNN models are statistically separated.

These two papers have since proved pivotal in improving the state of benchmarking in graph representation learning. They are also important in that they explicitly try to address the data coverage deficiency in graph data. Nevertheless, while both papers empirically show that their selection of datasets are valid in that they can statistically separate different GNN models, their selections are based on basic criteria (e.g. task type, application domain, graph size & density) that do not sufficiently explain GNN behavior on the datasets. Even though such criteria are strong indicators of GNN behavior on data, they do not necessarily dictate how a GNN will behave on a given dataset, or whether one GNN model will outperform another on it. Two datasets that vary greatly in graph size/density or application domain may simply be very close in the *data space*, and therefore provide no insight into the separability of GNNs.

More recent papers have taken additional steps forward to arrive at more complete definitions of graph data spaces. Palowitch et al. [106] explicitly construct a *benchmark dataset space* by synthetic graph generation: They define the dimensions through the parameters of the graph generation algorithm, which involves both standard graph metrics such as graph size as well as continuous structural variables, e.g. the in/out-cluster edge probabilities). They also show that most benchmarking datasets including the OGB datasets (which were introduced to improve data diversity) are concentrated in a small portion of their data space. However, their work relies on small synthetic datasets, which may not generalize well to real-world application domains. Furthermore, the graph generator parameters that define the data space still rely on graph metrics that do not always translate to GNN separability.

You et al. [161] measure how a variety of "anchor" models perform over a collection of benchmarking datasets, and try to identify datasets that GNNs behave similarly on. They then demonstrate that best-performing GNNs have similar architectures for tasks that are similar. The methodology is similar to our work in that its data space implicitly accounts for information flow, albeit is much limited in application and scope as it is focused on exploration of the GNN design space rather than the data space itself.

This work therefore aims to address data coverage from the perspective of *information flow* that has been overlooked in previous literature. We claim that the real driving force dictating how GNNs perform on graph data is information flow, i.e. how information is

embedded and distributed in a graph through GNNs. We then construct a benchmarking method for graph *datasets* (as opposed to models) to profile their information flow, and propose a taxonomy of existing benchmark datasets based on our method. The information flow perspective provides more insights into *how* different GNNs will behave on a certain dataset, and aims to serve as a guide in making informed decisions about benchmarking data selection.

Chapter 4

Taxonomization of Graph Benchmarking Datasets

4.1. Motivation

As we have covered in the previous chapter, recent work in graph benchmarking [34, 68, 106] have focused on compiling a set of large(r) benchmarking datasets across diverse graph domains, with the objective of reducing data bias and improving data coverage. These developments have improved the *modus operandi* of graph learning benchmarking, but still only provide "educated guesses" on whether a collection of datasets (or the ones put forward by them) attain sufficient coverage. The *dimensions* of data and task space considered in previous work can be grouped into the following categories:

— Tasks: Node, edge and graph-level tasks; regression vs. classification tasks; transductive vs. inductive tasks

— Application domain: Social networks, bio/cheminformatics, citation graphs, financial graphs, synthetic graphs

— Graph metrics: Average number of nodes/edges/triangles, graph density, min/max/average degree, average clustering coefficient

These recent work mostly operate under the assumption that sufficient coverage over these dimensions will translate to a sufficient coverage of the abstract graph data space, i.e. result in a collection of graphs that test for different aspects of graph learning models, and provide maximal separability of different GNNs. While certainly an improvement on its precedents, this viewpoint is not entirely reliable either. In fact, Dwivedi et al. [34] and Hu et al. [68]'s criticism of the TUDataset collection verify this unreliability: The TUDataset collection both leverages many application domains, and incorporates datasets with a large spectrum of graph metrics [100]; yet both papers have found that most GNNs perform almost identically over them. The authors improve upon these by making better educated guesses (like stating TUDataset graphs don't have sufficient coverage in the size dimension and in turn proposing

larger graphs). Better educated guesses naturally lead to better coverage of the data space; but this approach is evidently suboptimal: The large body of evidence we have presented so far indicates that the well-understood dimensions we associate with application domains and measurable graph metrics fail to efficiently capture the data space, an assertion we validate further with our supplemental study in Appendix C.1.

4.1.1. Solution formulation

In this work, we argue that the causally relevant dimensions of the data space (see Section 3.2.1) is better captured by the notion of *information flow* in graph data: Information flow refers to *how* task-related information is encoded and propagated in graph datasets via GNNs. GNNs make predictions on graphs by leveraging the information flow: Altering this information flow will change how a GNN leverages it, which will then be reflected in a change in its predictive ability. This has three important implications:

(1) All causally relevant data dimensions that a GNN model leverages in order to make a prediction are captured by the information flow induced in the graph by the model.

(2) Datasets with distinct information flows (i.e. a selection that has good coverage of the relevant data space) have a better chance of statistically separating distinct GNN algorithms, as different GNN models leverage the information flow in different ways (see Chapter 2).

(3) By altering different elements of information flow in graph datasets and measuring how GNN performance changes, we can test for the causal relevance of corresponding data dimensions. For example, if removing a specific node feature from a graph dataset reduces GNN performance on a task, then that node feature is causally relevant for that dataset and task.

Our formulation characterizes a clear relationship between the graph data space that maximizes GNN separability and the notion of information flow: Distinguishing datasets by their information flow is equivalent to distinguishing them in the corresponding data space. Consequently, maximizing data coverage in terms of information flow is a proxy to generating a data space that is optimal for GNN separability.

Based on this formulation, we aim to provide a structured framework to better characterize the information flow in graph datasets: Where is information embedded in a graph? Do long-term dependencies exist? How important is graph structure and/or node features? We propose to use the lens of empirical transformation sensitivity to answer these questions for graph datasets, and subsequently taxonomize their use as benchmarks in graph representation learning. Our approach is illustrated in Figure 4.1. Namely, we list our contributions in this study as:

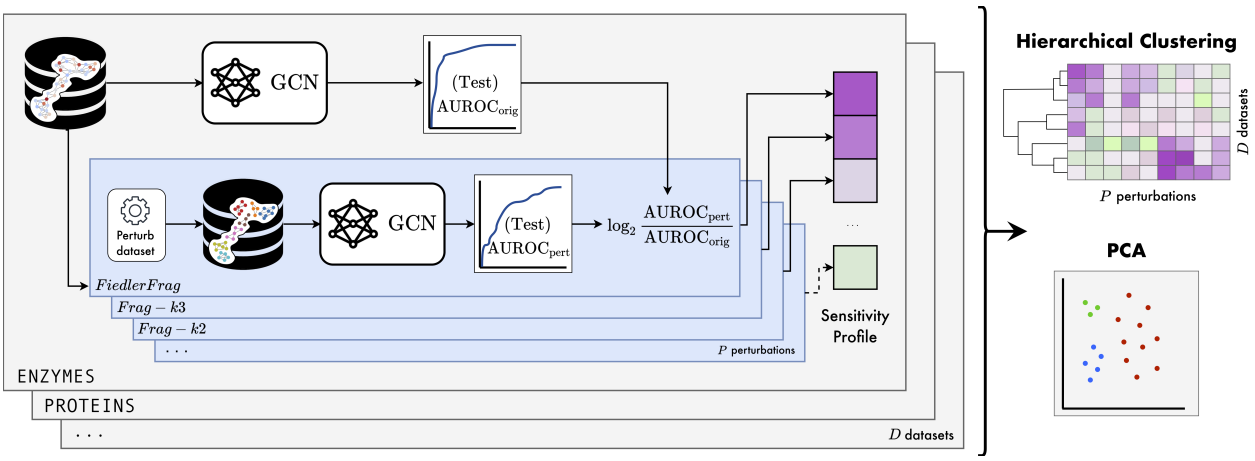


Figure 4.1 – Overview of our pipeline to taxonomize graph learning datasets.

(1) We develop a graph dataset taxonomization framework that is extendable to both new datasets and evaluation of additional graph/task properties,

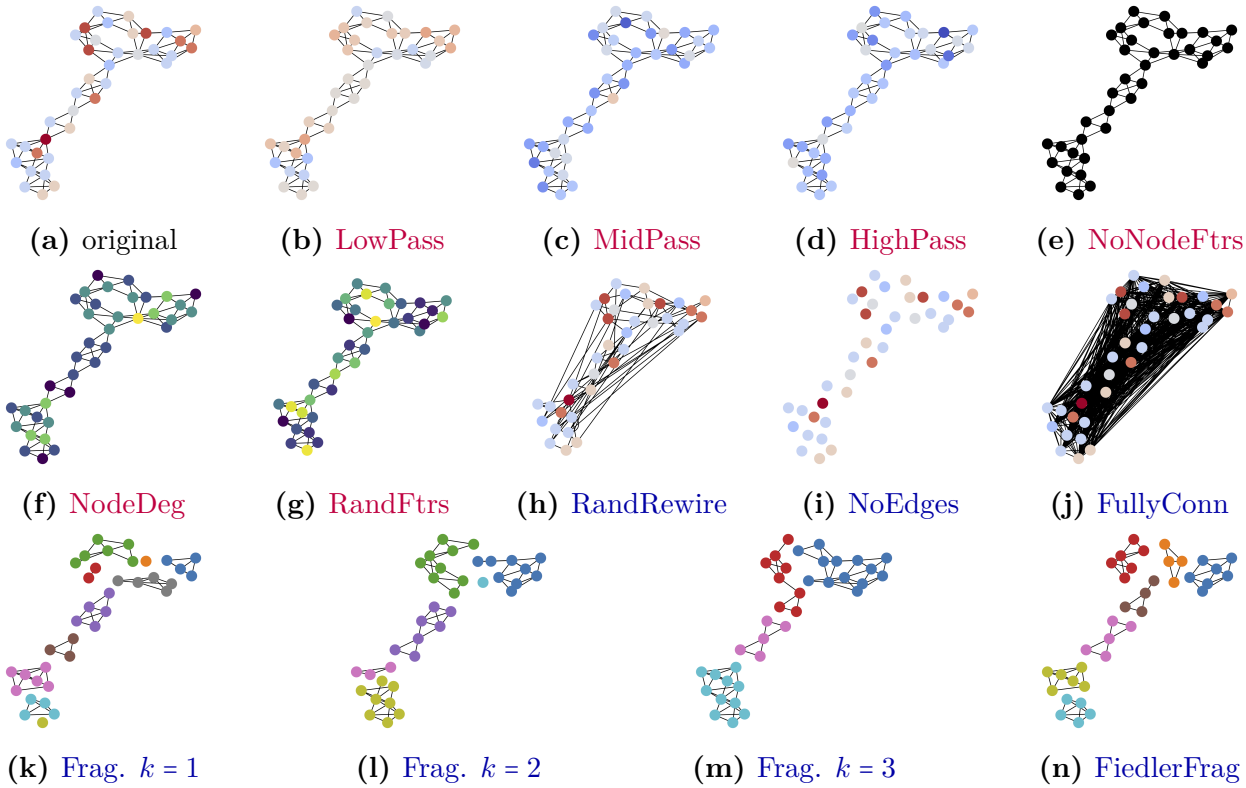
(2) Using this framework, we provide the first taxonomization of GNN (and GRL) benchmarking datasets, collected from TUDatasets [100], OGB [68] and other sources,

(3) Through the resulting taxonomy, we provide insights about existing datasets and guide better dataset selection in future benchmarking of GNN models.

4.2. Method

As a proxy for invariance or sensitivity to graph perturbations, we study the changes in GNN performance on perturbed versions of each dataset. These perturbations are designed to eliminate or emphasize particular types of information embedded in the graphs. We define an empirical *sensitivity profile* of a dataset as a vector where each element is the performance of a GNN after a given perturbation, reported as a percentage of the network's performance on the original dataset. In particular, we use a set of 13 perturbations, visualized in Figure 4.2. Of these perturbations, 6 are designed to perturb node features, while keeping the graph structure intact, whereas the remaining 7 keep the node attributes the same, but manipulate the graph structure.

For the purpose of these perturbations, we consider all graphs to be undirected and unweighted, and assume they all have node features, but not edge features. These assumptions hold for most datasets we use in this study. However, if necessary, we preprocess the data by symmetrizing each graph's adjacency matrix and dropping any edge attributes. With these assumptions in place, we also focus on classification tasks on node and graph-level, and do not consider edge prediction tasks or regression tasks in general. Nevertheless, we underline that our framework can encapsulate such tasks by extending it to datasets with edge features and/or applying appropriate metrics when evaluating regression tasks. Formally, let



(a) original **(b)** LowPass **(c)** MidPass **(d)** HighPass **(e)** NoNodeFtrs

(f) NodeDeg **(g)** RandFtrs **(h)** RandRewire **(i)** NoEdges **(j)** FullyConn

(k) Frag. $k = 1$ **(l)** Frag. $k = 2$ **(m)** Frag. $k = 3$ **(n)** FiedlerFrag

Figure 4.2 – Node feature and graph structure perturbations of the first graph in ENZYMES. The color coding of nodes illustrates their feature values, except (k-n) where the fragment assignment is shown.

$G = (\mathcal{V}, \mathcal{E}, \mathbf{X})$ be an undirected, unweighted, attributed graph with node set $\mathcal{V}$ of cardinality $|\mathcal{V}| = n$, edge set $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$, and a matrix of d-dimensional node features $\mathbf{X} \in \mathbb{R}^{n \times d}$. We let $\mathbf{M} \in \mathbb{R}^{n \times n}$ denote the adjacency matrix of each graph, where $\mathbf{M}(u,\nu) = 1$ if $(u,\nu) \in \mathcal{E}$ and zero otherwise.

Several of our perturbations are based on spectral graph theory, which represents graph signals in a spectral domain analogous to classical Fourier analysis. We define the graph Laplacian $\mathbf{L} \coloneqq \mathbf{D} - \mathbf{M}$ and the symmetric normalized graph Laplacian $\mathbf{N} \coloneqq \mathbf{D}^{-\frac{1}{2}}\mathbf{L}\mathbf{D}^{-\frac{1}{2}} = \mathbf{I} - \mathbf{D}^{-\frac{1}{2}}\mathbf{M}\mathbf{D}^{-\frac{1}{2}}$, where $\mathbf{D}$ is the diagonal degree matrix. Both $\mathbf{L}$ and $\mathbf{N}$ are positive semi-definite and have an orthonormal eigendecompositions $\mathbf{L} = \mathbf{\Phi}\mathbf{\Lambda}\mathbf{\Phi}^\top$ and $\mathbf{N} = \tilde{\mathbf{\Phi}}\tilde{\mathbf{\Lambda}}\tilde{\mathbf{\Phi}}^\top$. By convention, we order the eigenvalues and corresponding eigenvectors $\{(\lambda_i, \phi_i)\}_{0 \le i \le n-1}$ of $\mathbf{L}$ (and similarly in the case of $\mathbf{N}$) in ascending order $0 = \lambda_0 \le \lambda_1 \le \cdots \le \lambda_{n-1}$. The eigenvectors $\{\phi_i\}_{0 \le i \le n-1}$ constitute a basis of the space of graph signals and can be considered as generalized Fourier modes. The eigenvalues $\{\lambda_i\}_{0 \le i \le n-1}$ characterize the variation of these Fourier modes over the graph and can be interpreted as (squared) frequencies.

4.2.1. Node Feature Perturbations

We first consider two perturbations that alter local node features, setting them either to a fixed constant (w.l.o.g., one) for all nodes, or to a one-hot encoding of the degree of the node. We refer to these perturbations as *NoNodeFtrs* (since constant node features carry no additional information) and *NodeDeg*, respectively. In addition, we consider a random node feature perturbation (*RandFtrs*) by sampling a one-dimensional feature for each node uniformly at random within [−1,1]. Sensitivity to these perturbations, exhibited by a large decrease in predictive performance, may indicate that a dataset (or task) is dominated by highly informative node features.

We also develop spectral node feature perturbations. As in Euclidean settings, the Fourier decomposition can be used to decompose graph signals into a set of canonical signals, called Fourier modes, which are organized according to increasing variation (or frequency). In Euclidean Fourier analysis, these modes are sinusoidal waves oscillating at different frequencies. A standard practice in audio signal processing is to remove noise from a signal by identifying and removing certain Fourier modes or *frequency bands*. We generalize this technique to graph datasets and systematically remove certain graph Fourier modes to probe the importance of the corresponding frequency bands.

In this perturbation, we use the frequencies derived from the symmetric normalized graph Laplacian $\mathbf{L}_{\text{ns}}$ and split them into three roughly equal-sized frequency bands (*low*, *mid*, *high*), i.e., bins of subsequent eigenvalues. To assess the importance of each of the frequency bands, we then apply *hard* band-pass filtering to the graph signals (node feature vectors), i.e., we project the signals on the span of the selected Fourier modes. More specifically, for each band, we let $\mathbf{I}_{\text{band}}$ be a diagonal matrix with diagonal elements equal to one if the corresponding eigenvalue is in the band, and zero otherwise. Then, the hard band-pass filtered signal is computed as

$$\mathbf{X}_{\text{band}} = \tilde{\mathbf{\Phi}}\mathbf{I}_{\text{band}}\tilde{\mathbf{\Phi}}^{\top}\mathbf{X}. \tag{4.2.1}$$

The above band-pass filtering perturbation enables a precise selection of the frequency bands. However, it requires a full eigendecomposition of the normalized graph Laplacian, which is impractical for large graphs. We therefore provide an alternative approach based on wavelet bank filtering [23]. This leverages the fact that polynomial filters h of the normalized graph Laplacian directly transform the spectrum via $h(\mathbf{L}_{\text{ns}}) = \tilde{\mathbf{\Phi}}h(\tilde{\mathbf{\Lambda}})\tilde{\mathbf{\Phi}}^{\top}$, yielding the *frequency response* $h(\lambda)$ for any eigenvalue λ of $\mathbf{L}_{\text{ns}}$. This is usually done by taking the symmetrized diffusion matrix

$$\mathbf{T} = \frac{1}{2}(\mathbf{I} + \mathbf{D}^{-\frac{1}{2}}\mathbf{M}\mathbf{D}^{-\frac{1}{2}}) = \frac{1}{2}\left(2\mathbf{I} - \mathbf{L}_{\text{ns}}\right). \tag{4.2.2}$$

By construction, $\mathbf{T}$ admits the same eigenbasis as $\mathbf{L}_{\text{ns}}$ but its eigenvalues are mapped from [0,2] to [0,1] via the frequency response $h(\lambda) = 1 - \lambda/2$. As a result, large eigenvalues are mapped to small values (and vice versa). Next, we construct *diffusion wavelets* [27] that consist of differences of dyadic powers $2^k, k \in \mathbb{N}_0$ of $\mathbf{T}$, i.e., $\Psi_k = \mathbf{T}^{2^{k-1}} - \mathbf{T}^{2^k}$, which act as bandpass filters on the signal. Intuitively, this operator "compares" two neighborhoods of different sizes (radius 2^{k-1} and 2^k) at each node. Diffusion wavelets are usually maintained in a wavelet bank $\mathcal{W}_K = \{\mathbf{\Psi}_k, \mathbf{\Phi}_\mathbf{K}\}_{k=0}^K$, which contains additional highpass $\mathbf{\Psi}_0 = \mathbf{I} - \mathbf{T}$ and lowpass $\mathbf{\Psi}_\mathbf{K} = \mathbf{T}^K$ filters. In our experiments, we choose $K = 1$, resulting in the following low, mid, and highpass filtered node features:

$$\mathbf{X}_{\text{high}} = (\mathbf{I} - \mathbf{T})\mathbf{X}, \quad \mathbf{X}_{\text{mid}} = (\mathbf{T} - \mathbf{T}^2)\mathbf{X}, \quad \mathbf{X}_{\text{low}} = \mathbf{T}^2\mathbf{X}. \tag{4.2.3}$$

These filters correspond to frequency responses $h_{\text{high}}(\lambda) = \lambda/2$, $h_{\text{mid}}(\lambda) = (1-\lambda/2) - (1-\lambda/2)^2$ and $h_{\text{low}}(\lambda) = (1-\lambda/2)^2$. Therefore, the low-pass filtering preserves low-frequency information while suppressing high-frequency information whereas high-pass filtering does the opposite. The mid-pass filtering suppresses all frequencies. However, it preserves much more middle-frequency information than it does high- or low-frequency information.

Therefore, this filtering may be interpreted as approximation of the hard band-pass filtering discussed above. From the spatial message passing perspective, low-pass filtering is equivalent to local averaging of the node features, which has a profound implication on homophilic and heterophilic characteristics of the datasets (Sec. 4.4.2). Finally, since the computations needed in (4.2.3) can be carried out via sparse matrix multiplications, they have the advantage of scaling well to large graphs. Therefore, we utilize the wavelet bank filtering for the datasets with larger graphs considered in Sec. 4.4.2, while for the smaller graphs, considered in Sec. 4.4.1, we employ the direct band-pass filtering approach.

4.2.2. Graph Structure Perturbations

The following perturbations act on the graph structure by altering the adjacency matrix. By removing all edges (*NoEdges)* or making the graph fully-connected (*FullyConn*), we can eliminate the structural information completely and essentially turn the graph into a set. The difference between the two perturbations lies in whether all nodes are processed independently or all nodes are processed together. However, *FullyConn* is only applied to inductive datasets in Sec. 4.4.1 due to computational limitations. Furthermore, we consider a degree-preserving random edge rewiring perturbation (*RandRewire*). In each step, we randomly sample a pair of edges and randomly exchange their end nodes. We then repeat this process without replacement until 50% of the edges have been randomly rewired.

To inspect the importance of local vs. global graph structure, we designed the fragmentation perturbation (*Frag-k*), which randomly partitions the graph into connected components consisting of nodes whose distance to a seed node is less than k. Specifically, we randomly

draw one seed node at a time from the graph and extract its k-hop neighborhood by eliminating all edges between this new fragment and the rest of the graph; we repeat this process on the remaining graph until the whole graph is processed. A smaller k implies smaller components, and hence discards the global structure and long-range interactions.

Graph fragmentations can also be constructed using spectral graph theory. In our taxonomization, we adopt one such method, which we refer to as Fiedler fragmentation (*FiedlerFrag*) (see [74] and the references therein). In the case when the graph G is connected, ϕ_0, the eigenvector of the graph Laplacian $\mathbf{L}$ corresponding to $\lambda_0 = 0$, is constant. The eigenvector ϕ_1 corresponding to the next smallest eigenvalue, λ_1, is known as the *Fiedler vector* [41]. Since ϕ_0 is constant, it follows that ϕ_1 has zero average. This motivates partitioning the graph into two sets of vertices, one where ϕ_1 is positive and the other where ϕ_1 is negative. We refer to this process as binary Fiedler fragmentation. This heuristic is used to construct the ratio cut for a connected graph [57]. The ratio cut partitions a connected graph into two disjoint connected components $V = U \cup W$, such that the objective $|E(U,W)|/(|U| \cdot |W|)$ is minimized, where $E(U,W) \coloneqq \{(u,w) \in E : u \in U, w \in W\}$ is the set of removed edges when fragmenting G accordingly. This can be seen as a combination of the min cut objective (numerator), while encouraging a balanced partition (denominator).

FiedlerFrag is based on iteratively applying binary Fiedler fragmentation. In each step, we separate out the graph into its connected components and apply binary Fiedler fragmentation to the largest component. We repeat this process until either we reach 200 iterations, or the size of the largest connected component falls below 20. In contrast to the random fragmentation *Frag-k*, this perturbation preserves densely connected regions of the graph and eliminates connections between them. Thus, *FiedlerFrag* tests the importance of *inter community message flow*. Due to computational limits, we only apply *FiedlerFrag* to inductive datasets in Sec. 4.4.1 for which this computation is feasible.

4.2.3. Data-driven Taxonomization by Hierarchical Clustering

To study a systematic classification of the graph datasets, we use Ward's method [151] for hierarchical clustering analysis of their *sensitivity profiles*. The *sensitivity profiles* are established empirically by contrasting the performance of a GNN model on a perturbed dataset and on the original dataset. To quantify this performance change, we use $\log_2$-transformed ratio of test AUROC (area under the ROC curve). Thus a sensitivity profile for a dataset is a 1-D vector with as many elements as we have perturbation experiments. A visual representation of our taxonomy method is presented in Figure 4.1.

As we consider a large number of datasets in our taxonomy, we first construct a perturbation sensitivity matrix where each row represents a dataset and each column represents a perturbation. An entry in this matrix is computed by taking the ratio between the test score

achieved with the perturbed dataset and the test score achieved with the original dataset. As our performance metric we use the area under the receiver operating characteristic (AUROC) averaged over 10 random seed runs or 10 cross-validation folds, depending on whether a dataset has predefined data splits or not. Row-wise hierarchical clustering provides us a data-driven taxonomization of the datasets.

Using AUROC as our metric, the values of the perturbation sensitivity matrix range from 0.5 to 1 when a perturbation causes a loss in predictive performance, and from 1 to 2 when it improves it. Therefore we element-wise $\log_2$-transform the matrix to balance the two ranges and map the values onto $[-1, 1]$ before hierarchical clustering. Yet, for a more intuitive presentation, we show the original ratio values as percentages in our plots.

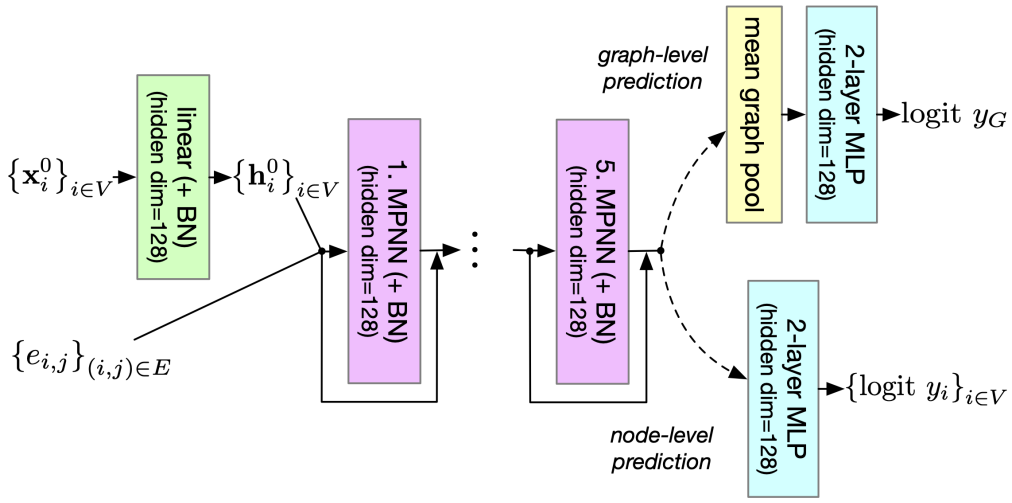


Figure 4.3 – MPNN model blueprint used for all datasets.

MPNN Hyperparameter Selection. We keep the model hyperparameters, illustrated in Figure 4.3, identical for each dataset and perturbation combination. We use a linear node embedding layer, 5 graph convolutional layers with residual connections and batch normalization (only for inductive datasets), followed by global mean pooling (in case of graph-level prediction tasks), and finally a 2-layer MLP classifier. For training we use Adam optimizer [78] with learning rate reduction by 0.5 factor upon reaching a validation loss plateau. Early stopping is done based on validation split performance.

GNN Model Selection. In order to generate *sensitivity profiles*, we must select suitable GNN models based on several practical considerations: (i) The model has to be expressive enough to efficiently leverage aspects of the node features and graph structure that we perturb. Otherwise, our analysis will not be able to uncover reliance on these properties. (ii) The model needs to be general enough to be applicable to a wide variety of datasets, avoiding the need for dataset-specific adjustments that may lead to perturbation profiling that is not comparable between datasets. Therefore, we did not aim for specialized models that

maximize performance, but rather models that (i) achieve at least baseline performance comparable to published works over all datasets, (ii) have manageable computational complexity to facilitate large-scale experimentation, and (iii) use well-established and theoretically well-understood architectures.

With these criteria in mind, we focused on two popular MPNN models in our analysis: GCN [79] and GIN [159]. The original GCN serves as an ideal starting point as its abilities and limitations are well-understood. However, we also wanted to perform taxonomization through a provably more expressive and recent method, which motivated our selection of GIN as the second architecture. We emphasize that the main focus here is not to provide a benchmarking of GNN models *per se*, but rather to address the taxonomization of *graph datasets* (and accompanying tasks) used in such benchmarks. Nevertheless, we have also generated sensitivity profiles by additional models in order to comparatively demonstrate the robustness of our approach: 2-Layer GIN, ChebNet [27], GatedGCN [12] and GCN II [20]; see Figure 4.6.

Implementation. Our pipeline is built using PyTorch [107] and PyG [39] with GraphGym [161] (provided under MIT License). Its modular & scalable design facilitated here one of the most extensive experimental evaluation of graph datasets to date.

4.3. Graph Learning Benchmarks

4.3.1. Inductive Datasets

`MNIST` and `CIFAR10` [34] are derived from the well-known image classification datasets. The images are converted to graphs by SLIC superpixelization; node features are the average pixel coordinates and intensities; edges are constructed based on kNN criterion.

`PATTERN` and `CLUSTER` [34] are node-level inductive datasets generated from SBMs [65]. In PATTERN, the task is to identify nodes of a structurally specific subgraph; CLUSTER has a semi-supervised clustering task of predicting the true cluster assignment of nodes while observing only one labelled node per cluster.

`IMDB-BINARY` [160] is a dataset of ego-networks, where nodes represent actors/actresses and an edge between two nodes means that the two artists played in a movie together. The task is to determine which genre (action or romance) each ego-network belongs to.

`D&D` [31] is a protein dataset where each protein is represented by a graph with rich node feature set. The task is to classify proteins as enzymes or non-enzymes.

`ENZYMES` [10] is a dataset of tertiary structures from six enzymatic classes (determined by Enzyme Commission numbers). Each node represents a secondary structure element (SSE), and has an edge between its three spatially closest nodes. Node features are the type of SSE, and the physical and chemical information.

PROTEINS [10] is a modification of the D&D [31]; the task is the same but the protein graphs are generated as in ENZYMES.

NCI1 and NCI109 [149] consist of graph representations of chemical compounds; each graph represents a molecule in which nodes represent atoms and edges represent atomic bonds. Atom types are one-hot encoded as node features. The tasks are to determine whether a given compound is active or inactive in inhibiting non-small cell lung cancer (NCI1) or ovarian cancer (NCI109).

COLLAB [160] is an ego-network dataset of researchers in three different fields of physics. Each graph is a researcher's ego-network, where nodes are researchers and an edge between two nodes means the two researchers have collaborated on a paper. The task is to determine which field a given researcher ego-network belongs to.

REDDIT-BINARY and REDDIT-MULTI-5K [160] graphs are derived from Reddit communities (subreddits). These subreddits are Q&A based or discussion-based. Each graph represents a set of interactions between users through posts and comments; nodes represent users while an edge implies an interaction between two users. The task for REDDIT-BINARY is to determine whether the given interaction graph belongs to a Q&A or discussion subreddit. In REDDIT-MULTI-5K, the graphs are drawn from 5 specific subreddits instead, and the task is to predict the subreddit a graph belongs to.

MUTAG [1] is a dataset of Nitroaromatic compounds. Each compound is represented by a graph in which nodes represent atoms with their types one-hot encoded as node features, and edges represent atomic bonds. The task is to determine whether a given compound has mutagenic effects on Salmonella typhimurium bacteria.

MalNet-Tiny [43] is a smaller version of MalNet dataset, consisting of function call graphs of various malware on Android systems using Local Degree Profiles as node features. In MalNet-Tiny, the task is constrained to classification into 5 different types of malware.

ogbg-molhiv, ogbg-molpcba, ogbg-moltox21 [68] datasets, adopted from MoleculeNet [155], are composed of molecular graphs, where nodes represent atoms and edges represent atomic bonds in-between. Node features include atom type and physical/chemical information such chirality and charge. The task is to classify molecules on whether they inhibit HIV replication (ogbg-molhiv) or their toxicity on on 12 different targets such as receptors and stress response pathways in a multilabel classification setting (ogbg-moltox21). In ogbg-molpcba the task is 128-way multi-task binary classification derived from 128 bioassays from PubChem BioAssay.

PCQM4Mv2-subset is our derivative of the OGB-LSC PCQM4Mv2 [69] molecular dataset. The original task is a regression of a quantum physical property – the HOMO-LUMO gap. For compatibility with our analysis, we quantized the regression task into 20-way classification task based on quantils of the training set. As true labels of the original "test-dev" and "test-challange" dataset splits are kept private by the OGB-LSC challenge organizers,

and for efficiency of our analysis, we created a custom reduced splits as follows: *train set*: random 10% of the original train set; *validation set*: another random 50,000 graphs from the original train set; *test set*: the original validation set. The molecular graphs are featurized the same way as in `ogbg-mol*` datasets.

`PPI` [166, 60] dataset contains a collection of 24 tissue-specific protein-protein interaction networks derived from the STRING database [140] using tissue-specific gold-standards from [54]. 20 of the networks are used for training, 2 used for validation, and 2 used for testing. In each network, each protein (node) is associated with 50 different gene signatures as node features. The multi-label node classification task was to classify each gene (node) in a graph based on its gene ontology terms.

`SYNTHETICnew` [38] is a dataset where each graph is based on a random graph G with scalar node features drawn from the normal distribution. Two classes of graphs are generated from G by randomly rewiring edges and permuting node attributes; the number of rewirings and permuted attributes are distinct for the two classes. Noise is added to the node features to make the tasks more difficult. The task is to determine which class a given graph belongs to.

`Synthie` [98] dataset is generated from two Erdös-Rényi graphs $G_{1,2}$: Two sets of graphs $S_{1,2}$ are then generated by randomly adding and removing edges from $G_{1,2}$. Then, 10 graphs were sampled from these sets and connected by randomly adding edges, resulting in a single graph. Two classes of these graphs, $C_{1,2}$ are generated by using distinct sampling probabilities for the two sets. The two classes are then in turn split into two by generating two sets of vectors A and B; nodes from a given graph were appended a vector from A as node features if they were sampled from S_1, and B for S_2 for one class, and vice versa for the other. The task is to classify which of these four classes a given graph belongs to.

`Small-world and Scale-free` [161] datasets are generated by tweaking graph generation parameters for the real-world-derived small-world [152] and scale-free [66] graphs. Graphs are generated using a range of Averaging Clustering Coefficient and Average Path Length parameters. In our experiments, clustering coefficients and PageRank scores constitute node features while task is to classify graphs based on average path length, where the continuous path length variable is rendered discrete by 10-way binning.

4.3.2. Transductive Node-level Datasets

`WikiNet` [109] contains two networks of Wikipedia pages, where edges indicate mutual links between pages, and node features are bag-of-words (BoW) of informative nouns. The task is to classify the web pages based on their average monthly traffic bins.

Table 4.1 – Inductive benchmarks. All datasets are equipped with graph-level classification tasks, except PATTERN and CLUSTER which are equipped with inductive node-level classification tasks.

Dataset	# Graphs	Avg # Nodes	Avg # Edges	# Features	# Classes	Predef. split	Ref.
MNIST	70,000	70.57	564.53	3	10	Yes	[34]
CIFAR10	60,000	117.63	941.07	5	10	Yes	[34]
PATTERN	14,000	118.89	6,078.57	3	2	Yes	[34]
CLUSTER	12,000	117.20	4,301.72	7	6	Yes	[34]
IMDB-BINARY	1,000	19.77	96.53	–	2	No	[160]
D&D	1,178	284.32	715.66	89	2	No	[31]
ENZYMES	600	32.63	62.14	21	6	No	[10]
PROTEINS	1,113	39.06	72.82	4	2	No	[10]
NCI1	4,110	29.87	32.3	37	2	No	[149]
NCI109	4,127	29.68	32.13	38	2	No	[149]
COLLAB	5,000	74.49	2,457.78	–	3	No	[160]
REDDIT-BINARY	2,000	429.63	497.75	–	2	No	[160]
REDDIT-MULTI-5K	4,999	508.52	594.87	–	5	No	[160]
MUTAG	188	17.93	19.79	7	2	No	[1]
MalNet-Tiny	5,000	1,410.3	2,859.94	5	5	No	[43]
ogbg-molhiv	41,127	25.5	27.5	9 sets	2	Yes	[68]
ogbg-molpcba	437,929	26.0	28.1	9 sets	128x binary	Yes	[68]
ogbg-moltox21	7,831	18.6	19.3	9 sets	12x binary	Yes	[68]
PCQM4Mv2-subset	446,405	14.1	14.6	9 sets	quantized to 20	Custom	[69]
PPI	24	2,372.67	66,136	50	121	Yes	[166]
SYNTHETICnew	300	100	196	1	2	No	[38]
Synthie	400	95	196.25	15	4	No	[98]
Small-world	256	64	694	2	10	No	[161]
Scale-free	256	64	501.56	2	10	No	[161]

`WebKB` [109] contains networks of web pages from different universities, where an (directed) edge is a hyperlink between two web pages, with BoW node features. The task is to classify the web pages into five categories: student, project, course, staff, and faculty.

`Actor` [109] is a network of actors, where an edge indicate co-occurrence of two actors on a same Wikipedia page, with node features represented by keywords about the actor on Wikipedia. The task is to classify the actor into one of five categories.

`WikiCS` [95] is a network of Wikipedia articles related to Computer Science, where edges represent hyperlinks between them, with 300-dimensional word embeddings of the articles. The task is to classify the articles into one of ten branches of the field.

`Flickr` [163] is a network of images, where the edges represent common properties between images, such as locations, gallery, and comments by the same users. The node features are BoW of image descriptions, and the task is to predict one of 7 tags for an image.

`CF` (CitationFull) [9] contains citation networks where nodes are papers and edges represent citations, with node features as BoW of papers. The task is to classify the papers based on their topics.

`DzEu` (DeezerEurope) [120] is a network of Deezer users from European countries where nodes are the users and edges are mutual follower relationships. The task is to predict the gender of users.

LFMA (LastFMAsia) [120] is a network of LastFM users from Asian countries where edges are mutual follower relationships between them. The task is to predict the location of users.
Amazon [133] contains Amazon Computers and Amazon Photo. They are segments of the Amazon co-purchase graph, where nodes represent goods, edges indicate that two goods are frequently bought together, node features are bag-of-words encoded product reviews, and class labels are given by the product category.
Coau (Coauthor) [133] contains Coauthor CS and Coauthor Physics. They are co-authorship graphs based on the Microsoft Academic Graph from the KDD Cup 2016 challenge 3. Nodes are authors, and are connected by an edge if they co-authored a paper; node features represent paper keywords for each author's papers, and class labels indicate most active fields of study for each author.
Twitch [119] contains Twitch user-user networks of gamers who stream in a certain language where nodes are the users themselves and the edges are mutual friendships between them. The task is to to predict whether a streamer uses explicit language. Due to low baseline performance even after a thorough hyperparameter search, we excluded `Twitch-RU` and `Twitch-FR` from our main analysis.
Github [119] is a network of GitHub developers where nodes are developers who have starred at least 10 repositories and edges are mutual follower relationships between them. The task is to predict whether the user is a web or a machine learning developer.
FBPP (FacebookPagePage) [119] is a network of verified Facebook pages that liked each other, where nodes correspond to official Facebook pages, edges to mutual likes between sites. The task is multi-class classification of the site category.

4.4. Results

Each of the 49 datasets we consider is equipped with either a node classification or graph classification task. In the case of node classification, we further differentiate between the *inductive* setting, in which learning is done on a set of graphs and the generalization occurs from a training set of graphs to a test set, and the *transductive* setting, in which learning is done in one (large) graph and the generalization occurs between subsets of nodes in this graph. Graph classification tasks, by contrast, always appear in an *inductive* setting. The only major difference between graph classification and inductive node classification is that prior to final prediction, the hidden representations of all nodes are pooled into a single graph-level representation. In the following two subsections, we provide an analysis of the sensitivity profiles for datasets with inductive and transductive tasks.

4.4.1. Taxonomy of Inductive Benchmarks

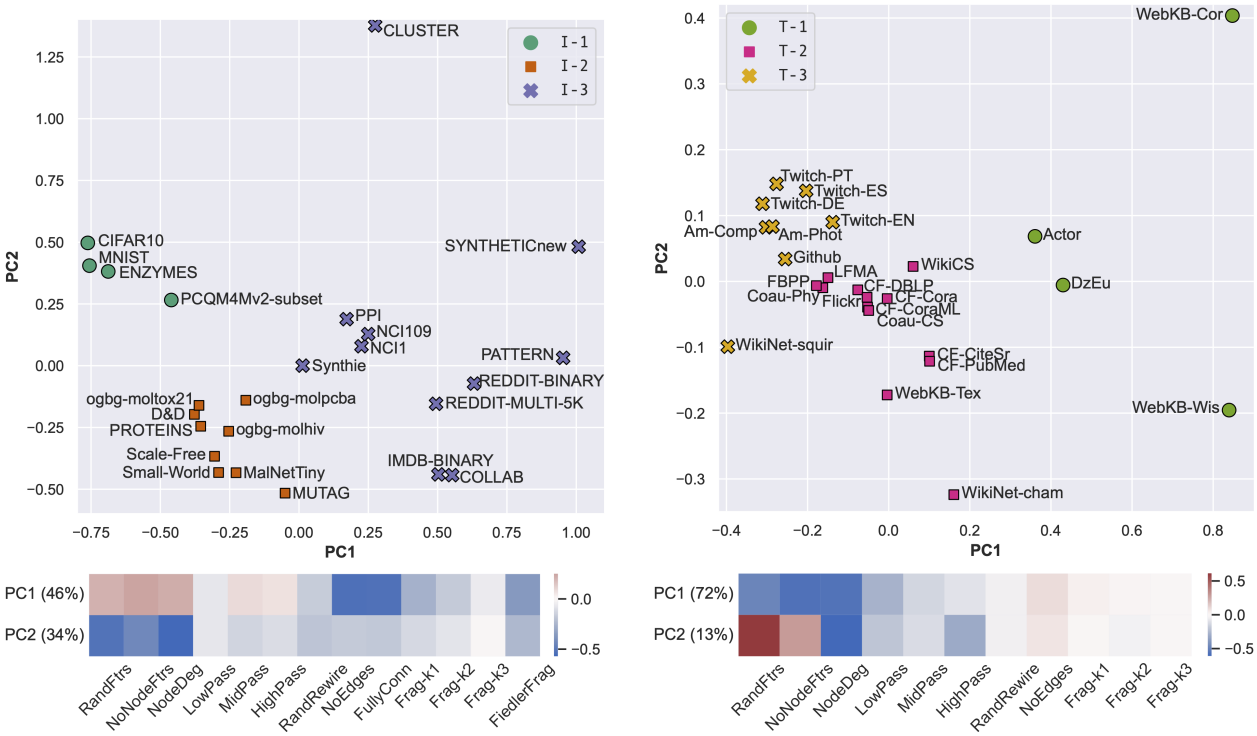
Table 4.2 – Transductive benchmarks with node-level classification tasks.

Dataset	# Nodes	# Edges	# Node feat.	# Pred. classes	Predef. split	Ref.
WikiNet-cham	2,277	72,202	128	5	Yes	[109]
WikiNet-squir	5,201	434,146	128	5	Yes	[109]
WebKB-Cor	183	298	1,703	10	Yes	[109]
WebKB-Wis	251	515	1,703	10	Yes	[109]
WebKB-Tex	183	325	1,703	10	Yes	[109]
Actor	7,600	30,019	932	10	Yes	[109]
WikiCS	11,701	297,110	300	10	Yes	[95]
Flickr	89,250	899,756	500	7	Yes	[163]
CF-Cora	19,793	126,842	8,710	70	No	[9]
CF-CoraML	2,995	16,316	2,879	7	No	[9]
CF-CiteSeer	4,230	10,674	602	6	No	[9]
CF-DBLP	17,716	105,734	1,639	4	No	[9]
CF-PubMed	19,717	88,648	500	3	No	[9]
DzEu	28,281	185,504	128	2	No	[120]
LFMA	7,624	55,612	128	18	No	[120]
Am-Comp	13,752	491,722	767	10	No	[133]
Am-Phot	7,650	238,162	745	8	No	[133]
Coau-CS	18,333	163,788	6,805	15	No	[133]
Coau-Phy	34,493	495,924	8,415	5	No	[133]
Twitch-EN	7,126	77,774	128	2	No	[119]
Twitch-ES	4,648	123,412	128	2	No	[119]
Twitch-DE	9,498	315,774	128	2	No	[119]
Twitch-PT	1,912	64,510	128	2	No	[119]
Github	37,700	578,006	128	2	No	[119]
FBPP	22,470	342,004	128	4	No	[119]

Datasets. We examine a total of 24 datasets, 21 of which are equipped with a graph-classification task (inductive by nature) and the other three are equipped with an inductive node-classification task. Of these datasets, 18 are derived from real-world data, while the other six are synthetically generated.

For real-world data, we consider several domains. Biochemistry tasks are the most ubiquitous, including compound classification based on effects on cancer or HIV inhibition (`NCI1` & `NCI109` [149], `ogbg-molhiv` [68]), protein-protein interaction `PPI` [166, 60], multilabel compound classification based on toxicity on biological targets (`ogbg-moltox21` [68]), and multiclass classification of enzymes (`ENZYMES` [68]). We also consider superpixel-based graph classification as an extension of image classification (`MNIST` & `CIFAR10` [34]), collaboration datasets (`IMDB-BINARY` & `COLLAB` [160]), and social graphs (`REDDIT-BINARY` & `REDDIT-MULTI-5K` [160]).

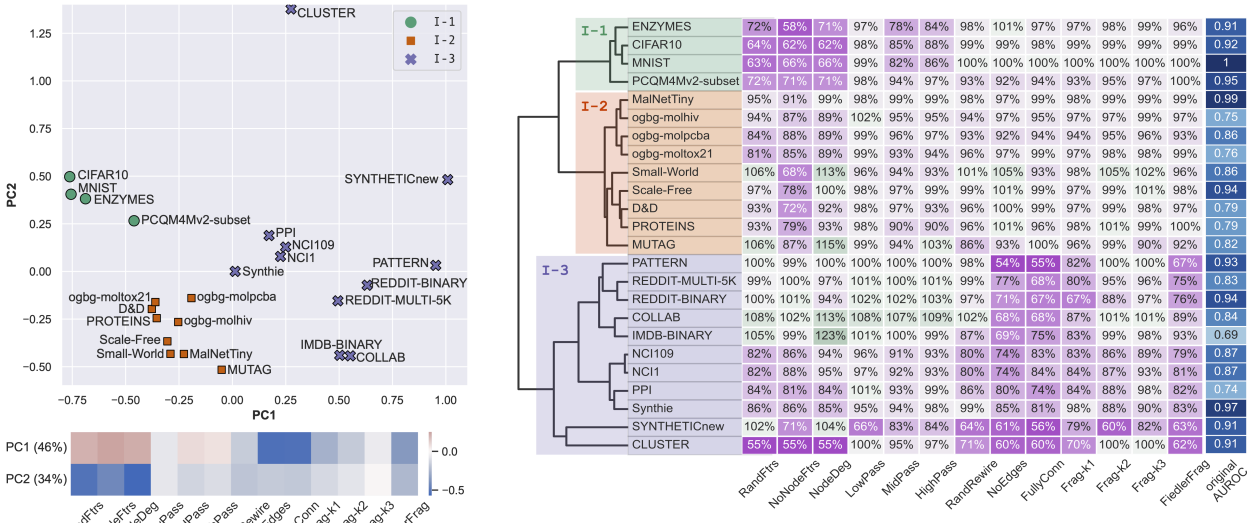
For synthetic data, we have a concrete understanding of their graph domain properties and how these properties relate to their respective prediction tasks. This allows us to derive a deeper understanding of their *sensitivity profiles*. The six synthetic datasets in our study make use of a varied set of graph generation algorithms. `Small-world` [161] is based on graph generation with the Watz-Strogatz (WS) model; the task is to classify graphs based on average path length. `Scale-free` [161] retains the same task definition, but the graph generation algorithm is an extension of the Barabási-Albert (BA) model proposed by Holme



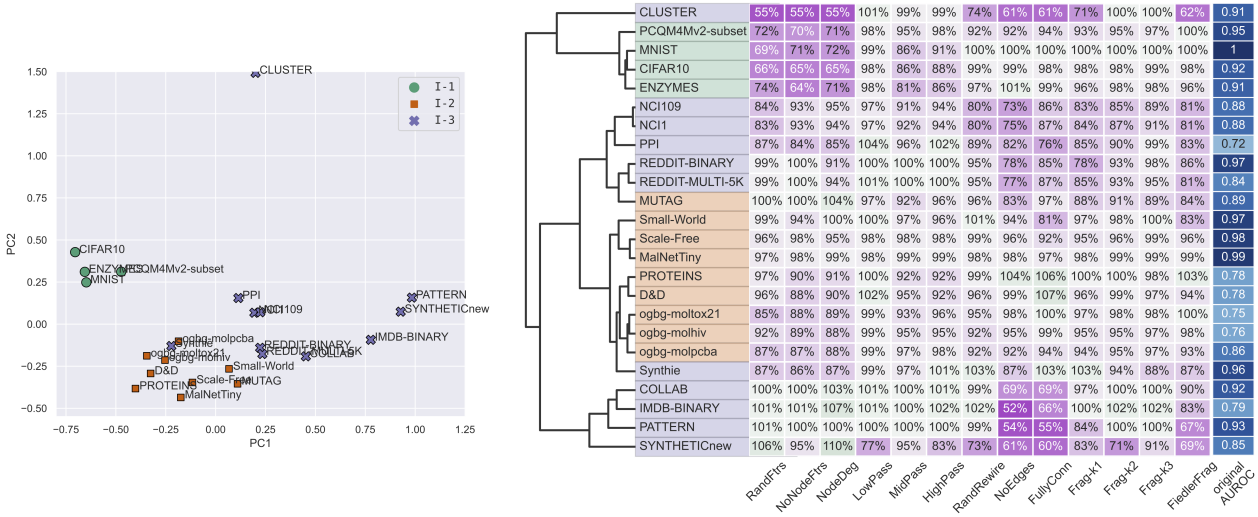
(a) Inductive graph datasets **(b)** Transductive graph datasets

Figure 4.4 – Visualization of (a) inductive and (b) transductive datasets based on PCA of their perturbation *sensitivity profiles* according to a GCN model. The datasets are labeled according to their taxonomization by hierarchical clustering, shown in Figure 4.5 and 4.7, which corroborates with the emerging clustering in the PCA plots. In the bottom part are shown the loadings of the first two principal components and (in parenthesis) the percentage of variance explained by each of them.

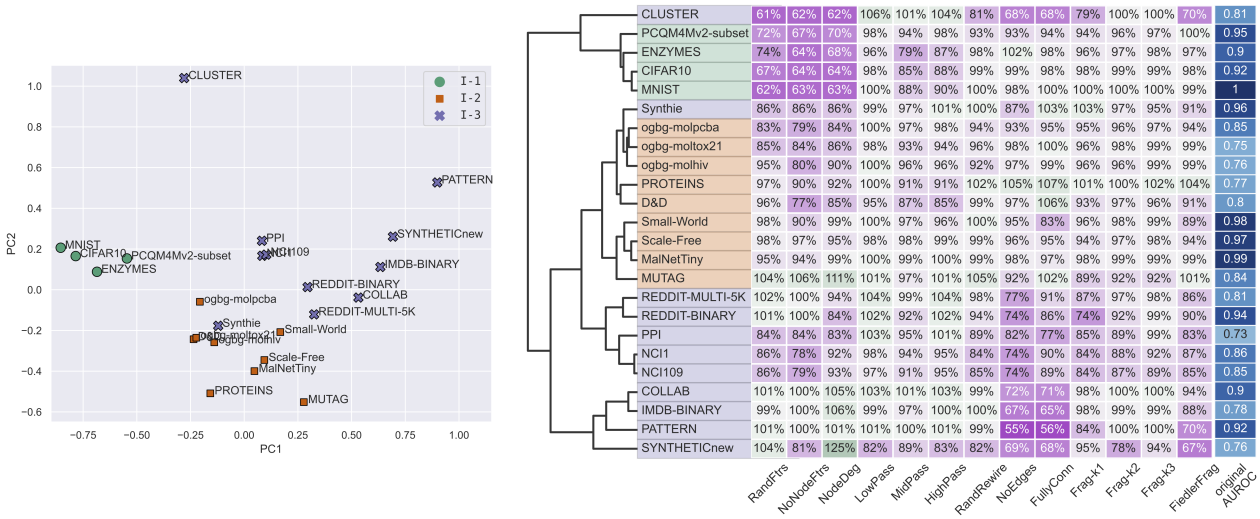
and Kim [66]. `PATTERN` and `CLUSTER` are node-level classification tasks generated with stochastic block models (SBM) [65]. `Synthie` [98] graphs are derived by first sampling graphs from the well-known Erdös-Rényi (ER) model, then deriving each class of graphs by a specific graph surgery and sampling of node features from a distinct distribution per each class. Similarly, `SYNTHETICnew` [38] graphs are generated from a random graph, where different classes are formed by specific modifications to the original graph structure and node features. Further details of dataset definitions and synthetic graph generation algorithms are provided in Section 4.3.



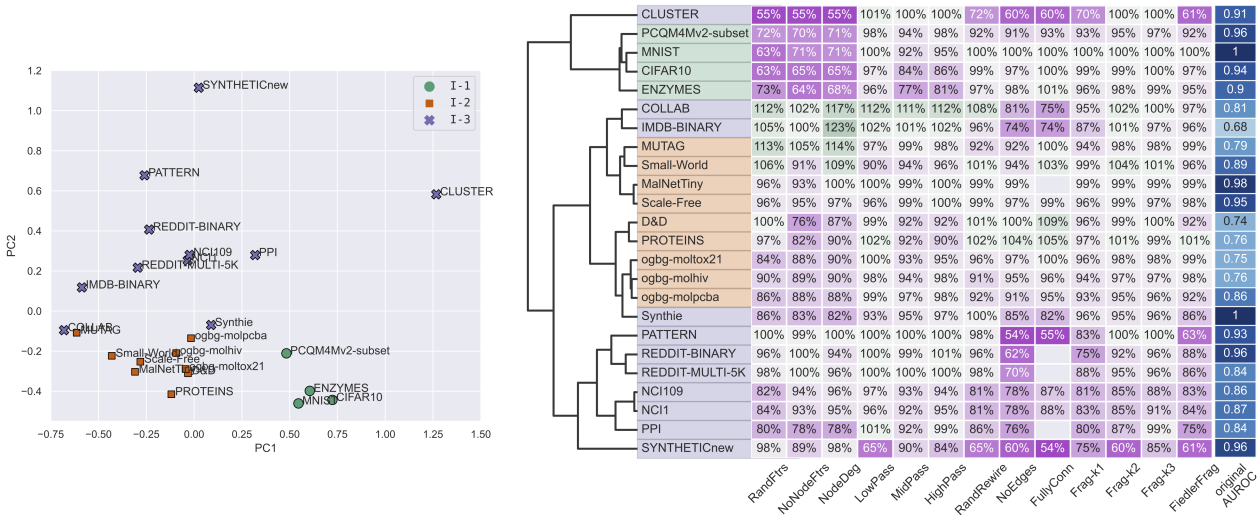
(a) Sensitivity profiles by **GCN** model.



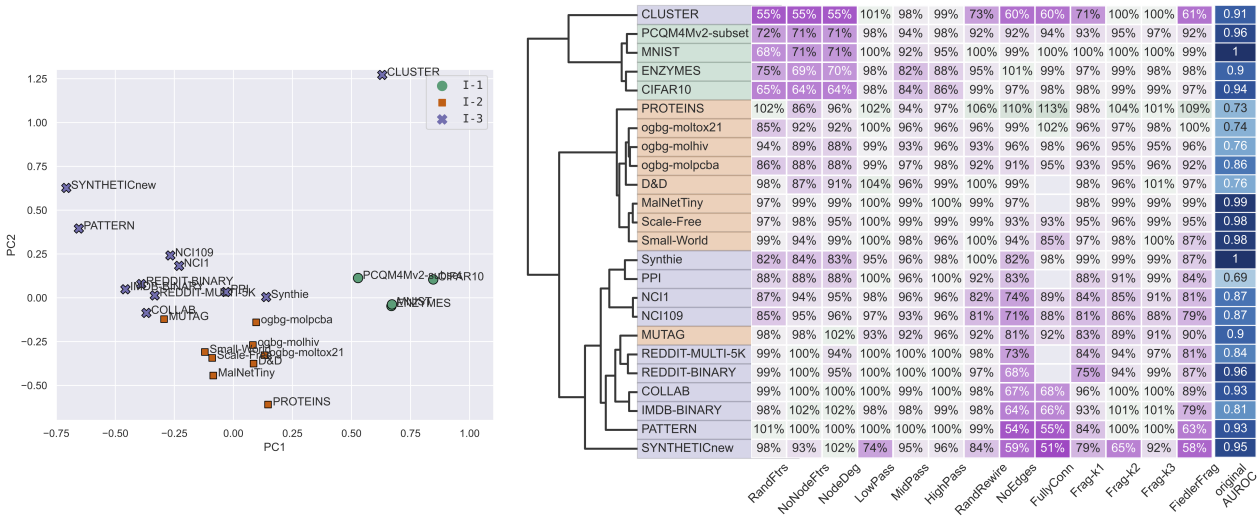
(b) Sensitivity profiles by **GIN** model; annotated by cluster assignment w.r.t. GCN model.



(c) Sensitivity profiles by **2-Layer GIN** model; annotated by cluster assignment w.r.t. GCN model.

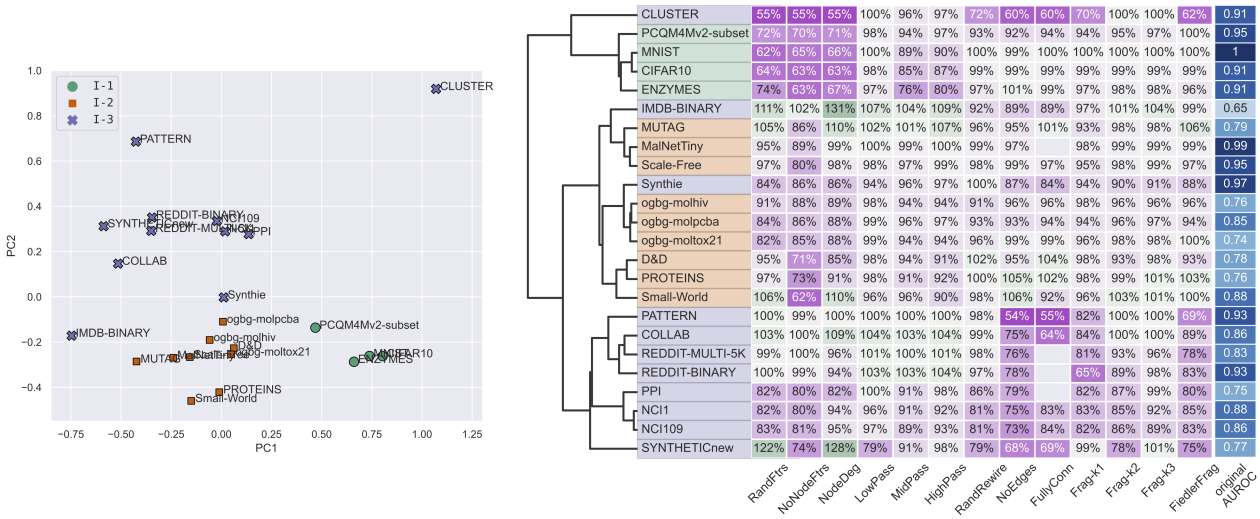


(d) Sensitivity profiles by **ChebNet** model; annotated by cluster assignment w.r.t. GCN model.



	RandFtrs	NoNodeFtrs	NodeDeg	LowPass	MidPass	HighPass	RandRewire	NoEdges	FullyConn	Frag-k1	Frag-k2	Frag-k3	FiedlerFrag	original AUROC
CLUSTER	55%	55%	55%	101%	98%	99%	73%	60%	60%	71%	100%	100%	61%	0.91
PCQM4Mv2-subset	72%	71%	71%	98%	94%	98%	92%	92%	94%	93%	95%	97%	92%	0.96
MNIST	68%	71%	71%	100%	92%	95%	100%	99%	100%	100%	100%	100%	99%	1
ENZYMES	75%	69%	70%	98%	82%	88%	95%	101%	99%	97%	99%	98%	98%	0.9
CIFAR10	65%	64%	64%	98%	84%	86%	99%	97%	98%	98%	99%	99%	97%	0.94
PROTEINS	102%	86%	96%	102%	94%	97%	106%	110%	113%	98%	104%	101%	109%	0.73
ogbg-moltox21	85%	92%	92%	100%	96%	96%	96%	99%	102%	96%	97%	98%	100%	0.74
ogbg-molhiv	94%	89%	88%	99%	93%	96%	93%	96%	98%	96%	95%	95%	96%	0.76
ogbg-molpcba	86%	88%	88%	99%	97%	98%	92%	91%	95%	93%	95%	96%	92%	0.86
D&D	98%	87%	91%	104%	96%	99%	100%	99%		98%	96%	101%	97%	0.76
MalNetTiny	97%	99%	99%	100%	99%	100%	99%	97%		98%	99%	99%	99%	0.99
Scale-Free	97%	98%	95%	100%	99%	99%	99%	93%	93%	95%	96%	99%	95%	0.98
Small-World	99%	94%	99%	100%	98%	96%	100%	94%	85%	97%	98%	100%	87%	0.98
Synthie	82%	84%	83%	95%	96%	98%	100%	82%	98%	99%	99%	99%	87%	1
PPI	88%	88%	88%	100%	96%	100%	92%	83%		88%	91%	99%	84%	0.69
NCI1	87%	94%	95%	98%	96%	96%	82%	74%	89%	84%	85%	91%	81%	0.87
NCI109	85%	95%	96%	97%	93%	96%	81%	71%	88%	81%	86%	88%	79%	0.87
MUTAG	98%	98%	102%	93%	92%	96%	92%	81%	92%	83%	89%	91%	90%	0.9
REDDIT-MULTI-5K	99%	100%	94%	100%	100%	100%	98%	73%		84%	94%	97%	81%	0.84
REDDIT-BINARY	99%	100%	95%	100%	100%	100%	97%	68%		75%	94%	99%	87%	0.96
COLLAB	99%	100%	100%	100%	99%	100%	98%	67%	68%	96%	100%	100%	89%	0.93
IMDB-BINARY	98%	102%	102%	98%	98%	99%	98%	64%	66%	93%	101%	101%	79%	0.81
PATTERN	101%	100%	100%	100%	100%	100%	99%	54%	55%	84%	100%	100%	63%	0.93
SYNTHETICnew	98%	93%	102%	74%	95%	96%	84%	59%	51%	79%	65%	92%	58%	0.95

(e) Sensitivity profiles by **GatedGCN** model; annotated by cluster assignment w.r.t. GCN model.



	RandFtrs	NoNodeFtrs	NodeDeg	LowPass	MidPass	HighPass	RandRewire	NoEdges	FullyConn	Frag-k1	Frag-k2	Frag-k3	FiedlerFrag	original AUROC
CLUSTER	55%	55%	55%	100%	96%	97%	72%	60%	60%	70%	100%	100%	62%	0.91
PCQM4Mv2-subset	72%	70%	71%	98%	94%	97%	93%	92%	94%	94%	95%	97%	100%	0.95
MNIST	62%	65%	66%	100%	89%	90%	100%	99%	100%	100%	100%	100%	100%	1
CIFAR10	64%	63%	63%	98%	85%	87%	99%	99%	99%	99%	99%	99%	99%	0.91
ENZYMES	74%	63%	67%	97%	76%	80%	97%	101%	99%	97%	98%	98%	96%	0.91
IMDB-BINARY	111%	102%	131%	107%	104%	109%	92%	89%	89%	97%	101%	104%	99%	0.65
MUTAG	105%	86%	110%	102%	101%	107%	96%	95%	101%	93%	98%	98%	106%	0.79
MalNetTiny	95%	89%	99%	100%	99%	100%	99%	97%		98%	99%	99%	99%	0.99
Scale-Free	97%	80%	98%	98%	97%	99%	98%	99%	97%	95%	98%	99%	97%	0.95
Synthie	84%	86%	86%	94%	96%	97%	100%	87%	84%	94%	90%	91%	88%	0.97
ogbg-molhiv	91%	88%	89%	98%	94%	94%	91%	96%	96%	98%	96%	96%	96%	0.76
ogbg-molpcba	84%	86%	88%	99%	96%	97%	93%	93%	94%	94%	96%	97%	94%	0.85
ogbg-moltox21	82%	85%	88%	99%	94%	94%	96%	99%	99%	96%	98%	98%	100%	0.74
D&D	95%	71%	85%	98%	94%	91%	102%	95%	104%	98%	93%	98%	93%	0.78
PROTEINS	97%	73%	91%	98%	91%	92%	100%	105%	102%	98%	99%	101%	103%	0.76
Small-World	106%	62%	110%	96%	96%	90%	98%	106%	92%	96%	103%	101%	100%	0.88
PATTERN	100%	99%	100%	100%	100%	100%	98%	54%	55%	82%	100%	100%	69%	0.93
COLLAB	103%	100%	109%	104%	103%	104%	99%	75%	64%	84%	100%	100%	89%	0.86
REDDIT-MULTI-5K	99%	100%	96%	101%	100%	101%	98%	76%		81%	93%	96%	78%	0.83
REDDIT-BINARY	100%	99%	94%	103%	103%	104%	97%	78%		65%	89%	98%	83%	0.93
PPI	82%	80%	82%	100%	91%	98%	86%	79%		82%	87%	99%	80%	0.75
NCI1	82%	80%	94%	96%	91%	92%	81%	75%	83%	83%	85%	92%	85%	0.88
NCI109	83%	81%	95%	97%	89%	93%	81%	73%	84%	82%	86%	89%	83%	0.86
SYNTHETICnew	122%	74%	128%	79%	91%	98%	79%	68%	69%	99%	78%	101%	75%	0.77

(f) Sensitivity profiles by **GCNII** model; annotated by cluster assignment w.r.t. GCN model.

Figure 4.5 – Taxonomy of inductive graph learning datasets via graph perturbations. The categorization into 3 dataset clusters is stable across the following models with only minor deviations: (a) GCN, (b) GIN, (c) 2-Layer GIN, (d) ChebNet, (e) GatedGCN, (f) GCNII. Missing performance ratios (due to out-of-memory error) are shown in gray.

General Insights. Here we first itemize the main insights into inductive datasets, and then proceed with the analysis of individual clusters. Our full taxonomy is shown in Figures 4.5 and 4.4a.

— **Three distinct groups of datasets.** We identify a categorization into three dataset clusters `I-{1,2,3}` that emerge from both the hierarchical clustering and PCA. The datasets in `I-{1,2}` exhibit stronger node feature dependency and do not appear to contain crucial information encoded in the graph structure. For them, the node features are sufficient, even though the structure itself may be non-trivial. The main differentiating factor between `I-1` and `I-2` is their relative sensitivity to node feature perturbations – in particular, how well *NodeDeg* can substitute the original node features. On the other hand, datasets in `I-3` rely considerably more on graph structure for correct task prediction. This is also reflected by the first two principal components (Figure 4.4a), where PC1 approximately corresponds to structural perturbations and PC2 to node feature perturbations.

— **No clear clustering by dataset domain.** While datasets that are derived in a similar fashion cluster together (e.g., `REDDIT-*` datasets), in general, each of the three clusters contains datasets from a variety of application domains. Not all molecular datasets behave alike; e.g., `ogbg-mol*` datasets in `I-2` considerably differ from `NCI*` datasets in `I-3`.

— **Synthetic datasets do not fully represent real-world scenarios.** `CLUSTER`, `SYNTHETICnew`, and `PATTERN` lie at the periphery of the PCA embeddings, suggesting that existing synthetic datasets do not resemble the type of complexity encountered in real-world data. Hence, one should use synthetic datasets in conjunction with real-world datasets to comprehensively evaluate GNN performance rather than solely relying on synthetic ones. Nevertheless, the closest real-world datasets to `PATTERN` are `REDDIT-BINARY` and `REDDIT-MULTI-5K`. This proximity makes intuitive sense as all three datasets rely on finding substructures in graphs that infer the labels. We also note that the sensitivity profiles of all synthetic datasets are well-accounted for w.r.t. their respective design criteria which validate our approach; we refer the reader to Sec. 4.4.1 for a more detailed analysis.

— **Representative set.** One can now select a representative subset of all datasets to cover the observed heterogeneity among the datasets. Our recommendation: `PCQM4Mv2-subset`, `CIFAR10` from `I-1`; `D&D`, `ogbg-molpcba` from `I-2`; `NCI1`, `COLLAB`, `REDDIT-MULTI-5K`, `CLUSTER` from `I-3`.

— **Robustness w.r.t. GNN choice.** In addition to GCN, we have performed our perturbation analysis w.r.t. GIN [159], 2-Layer GIN, ChebNet [27], GatedGCN [12] and GCN II [20] models as well. These models were selected to cover a variety of inductive model biases: GIN is a provably 1-WL expressive GNN, ChebNet uses higher-order approximation of the Laplacian, GatedGCN employs gating akin to attention, and GCN II leverages skip connections and identity mapping to alleviate oversmoothing

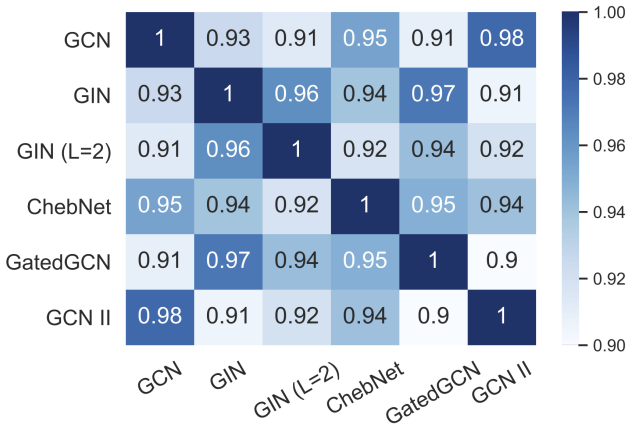


Figure 4.6 – Pearson correlation between perturbation profiles derived by six GNN models.

of the original GCN. We have also tested a 2-layer GIN to probe the robustness to number of message-passing layers.

The taxonomies w.r.t. other models (Figure 4.5) are congruent with that of GCN shown here. Given the differing inductive biases and representational capacity, some difference in the sensitivity profiles are not only expected but desired to validate their functions in benchmarking. The resulting profiles can be used for a detailed comparative analysis of these models, but the overall conclusions remain consistent. This consistency is further validated in Figure 4.6, in which we have conducted correlation analysis amongst these models. The Pearson correlation coefficients of all pairs are above 90%, implying that our taxonomy is sufficiently robust w.r.t. different GNNs and the number of layers.

`I-1`: Node-feature reliance. The top-most cluster `I-1`, while indifferent to structural perturbations, is highly sensitive to node feature perturbations that comprise the left-hand-side columns in Figure 4.5. The presence of image-based datasets `MNIST` and `CIFAR10` in this cluster is not surprising, as for superpixel graphs the structure loosely follows a grid layout for all classes, meaning determining class solely based on structure is difficult. Additionally, the coordinate information of superpixels is encoded also in the node features, together with average pixel intensities. A model with powerful enough classifier component is then sufficient for achieving high accuracy using these node features alone. Furthermore, the sensitivity of these datasets to *MidPass* and *HighPass* indicates that the overall shape of the signals encoded by low-frequencies is more informative for classifying the image content than sharp superpixel transitions encoded by high-frequencies. The presence of `ENZYMES` in `I-1` is likely due to the fact that some of the node features are precomputed using graph kernels, and therefore are sufficient to distinguish the enzyme classes in the dataset when structural information is removed. Last but not least, `PCQM4Mv2-subset` dataset appears to have a complex task that is dominated by the node feature information, yet the graph

structure encodes non-negligible information as well. Out of all datasets in the I-1 cluster, PCQM4Mv2-subset is the most sensitive one to structural perturbations. This corroborates with the expectation that predicting the HOMO-LUMO gap, which is the energy difference between the highest occupied molecular orbital (HOMO) and lowest unoccupied molecular orbital (LUMO), is a complex task that heavily depends on atom types, their bonds, and relative distances.

I-2: Node features contain majority of necessary structural information. For datasets in I-2, the graph structural information is again not necessary for achieving the baseline performance if the original node features are present, while the performance deteriorates noticably if *NoNodeFtrs* is applied. However, unlike I-1, these datasets are much less affected overall by the perturbations on node features. Many of the node features on these datasets are themselves derived from the graph's geometry, and it seems MPNNs are able to use either the graph structure or the node features to compensate for the absence of the other when encountering perturbed graphs. It appears that the low/mid/high-pass filterings in particular are able to retain a significant amount of geometric information.

The synthetic graphs of Scale-Free and Small-world (both I-2 datasets) are generated through different algorithms (WS and BA, respectively), but the node features and tasks are equivalent: The features are the local clustering coefficient and PageRank score of each node and the task is to classify graphs based on average path length. Since the encoded features are derived from graph structure itself, MPNNs are still able to exploit them when the original graph structure is perturbed. When the MPNNs are forced to rely on graph structure instead, they are still able to attain AUROCs above random despite some decrease.

For many of the I-2 datasets, *NodeDeg* allows one to replace geometric information of original node features with new geometric information, the degree of each vertex, to large success – for some of them the original AUROC scores are recovered and even surpassed, possibly due to *NodeDeg* reinforcing the existing structural signal. This trend is not as pronounced when the GIN-based model is used, since GIN achieves a comparatively high level of performance even in the face of *NoNodeFtrs*, likely due to the higher expressiveness of GIN compared to GCN in distinguishing of structural patterns.

On the other hand, there are datasets of biochemical origin in this cluster, whose node features encode chemical and physical attributes, such as atom or amino acid type. Except MUTAG, there appears to be some information encoded in these node features that is irreplaceable by graph structure or node degree information.

I-3: Graph-structure reliance. The I-3 cluster is characterized by strong structural dependencies, and can be further divided into two subgroups based on their sensitivities to node feature perturbations.

The first subgroup, which consists of PATTERN, COLLAB, IMDB-BINARY and REDDIT, is not affected by node feature perturbations. These datasets do not have any original informative node features and their tasks appear to be purely structure-based. Indeed, in the case of PATTERN the task is to detect structural patterns in graphs, rendering node features irrelevant for the task. On the other hand, structural perturbations such as *NoEdges* and *FullyConn* cause drastic performance drops in this group, since most of its task signals are sourced from graph structures. This group also exhibits limited to no sensitivity towards *Frag-k2* and *Frag-k3* perturbations, which test for degrees of reliance on longer range interactions by limiting information propagation to $\{2, 3\}$ hops. We still see prominent sensitivity to *Frag-k1*, though, implying reliance on information from immediate neighbors. We can attribute the insensitivity for $k > 1$ to inherent graph properties for some of these datasets: For dense networks like PATTERN or ego-nets such as IMDB-BINARY and COLLAB, just 1 or 2 hops recover the original graph – for these graphs, the notion of long-range information does not exist.

The second I-3 subgroup, formed by NCI datasets and Synthie, are the datasets that are notably affected by *all* perturbations. For Synthie, this sensitivity stems from its construction. The four synthetic classes in Synthie are formed by combinations of two distributions of graph structures and two distributions of node features – elimination of either leads to a *partial* collapse in the distinguishability of two classes. The NCI classification tasks, similarly to related bioinformatics datasets in I-2, show a degree of reliance on the high-dimensional node features, but additionally, they are also dependent on non-local structure as they are among the datasets most adversely affected by *Frag-k2* and *Frag-k3*.

Synthetic datasets CLUSTER and SYNTHETICnew are also adversely affected by both structural and node feature perturbations. However, they stand out due to the magnitude of this effect. Many of the perturbations lead to a major decrease in AUROC and close-to-random performance. A closer inspection can provide an explanation. The task of CLUSTER is semi-supervised clustering of unlabeled nodes into six clusters, and the true cluster labels are given as node features in only a single node per cluster. *NoEdges* and *FullyConn* remove the cluster structure altogether, while *NoNodeFtrs* and *NodeDeg* remove the given cluster labels, rendering the task unsolvable in either case. In SYNTHETICnew, the two classes are derived from a "base" graph by a class-specific edge rewiring and node feature permutation, hence either graph structure or node features should differentiate the classes. Despite such expectation, we observe that the original node features alone are not sufficient, as structure perturbations have detrimental impact on the prediction performance. On the other hand GIN and GCN with *NodeDeg* can learn to distinguish the two classes even without the original node features. Thus, the original node features appear to be unnecessary, while after bandpass-filtering even provide misleading signal.

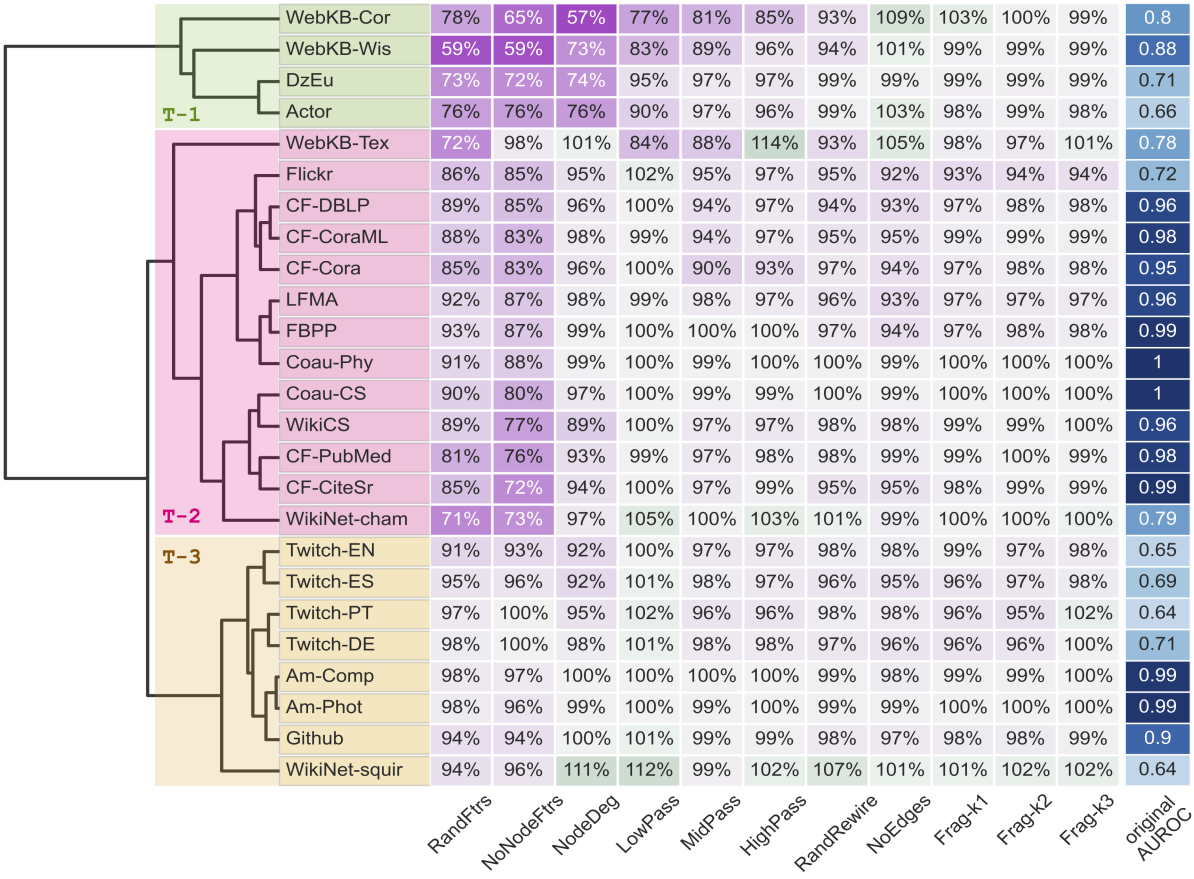


Figure 4.7 – Taxonomization of transductive datasets into 3 clusters based on sensitivity profiles w.r.t. a GCN-based model.

4.4.2. Taxonomy of Transductive Benchmarks

Datasets. We selected a wide variety of 25 transductive datasets with node classification task, including citation networks, social networks, and other web page derived networks (see Section 4.3).

In citation networks, such as CitationFull (`CF`) [9], nodes and edges correspond to papers that are linked via citation. In web page derived networks, like `WikiNet` [109], `Actor` [109], and `WikiCS` [95], they correspond to hyperlinks between pages. In social networks, like Deezer (`DzEu`) [120], LastFM (`LFMA`) [120], `Twitch` [119], Facebook (`FBPP`) [119], `Github` [119], and `Coau` [133], nodes and edges are based on a type of relationship, such as mutual-friendship and co-authorship. `Flickr` [163] and `Amazon` [133] are constructed based on other notions of similarity between entities, such as co-purchasing and image property similarities. `WebKB` [109] contains networks of university web pages connected via hyperlinks. It is an example of a *heterophilic* dataset [101], since immediate neighbor nodes do not necessarily share the same labels (which correspond to a user's role such as faculty or graduate student). By

contrast, Cora, CiteSeer, and PubMed are known to be *homophilic* datasets where nodes within a neighborhood are likely to share the same label. In fact, no less than 60% of nodes in these networks have neighborhoods that share the same node label as the central node [95].

General Insights. Below we list the main insights into transductive graph datasets and their taxonomy (Figures 4.7 and 4.4b). We then again proceed with the analysis of individual clusters.

— **Transductive datasets are uniformly insensitive to structural perturbations.** Sensitivity profiles of *all* transductive datasets show high robustness to *all* graph structure perturbations. This is in stark contrast with the inductive datasets, where the largest cluster I-3 is defined by high sensitivity to structural perturbations. The lowest performance achieved for a transductive dataset due to graph structure removal is still as high as 92% (Flickr). Furthermore, on average, considering only the neighborhoods of up to 3-hops (*Frag-k3*) nearly retains the full potential of the model (99% $\pm$ 1.6%), revealing the lack of long-range dependencies in these node-level datasets. Graph connectivity may not be vital to every dataset/task, e.g., in WikiCS word embeddings of Wikipedia pages may be sufficient for categorization without hyperlinks. Additionally, negligence of the full graph structure might be attributed to the limitations of the GCN expressivity and issues such as oversquashing [141]. While these limitations are fundamentally true, our observation of long-range dependencies on some graph-level tasks like NCI, coupled with our architecture being 5 layers deep with residual connections, indicate that our GCN model is capable of capturing non-local information in the 3-hop neighborhoods. Furthermore, our observed long-range *independence* in transductive node-level datasets is consistent with the promising results presented by recent development of scalable GNNs that operate on subgraphs [21, 163, 40], breaking or limiting long-range connections. While the observation that no dataset significantly depends on structural information is startling, it corroborates with the reported strong performance of MLP or similar models augmented with label propagation to outperform GNNs in several of these transductive datasets. [47, 72].

— **Three distinct groups of datasets.** The transductive datasets are also categorized into three clusters as T-{1,2,3}. T-1 consists of heterophilic datasets, such as WebKB and Actor [101, 91]. These are well-separated from others, as seen in the right half of the PCA plot (Figure 4.4b), primarily via PC1, which is characterized by performance drop due to removal of the original node features (*NoNodeFtrs*, *RandFtrs*) and their replacement by node degrees (*NodeDeg*). T-3 is indifferent to both node and structure removal, implying redundancies between node features and graph structure for their tasks. T-2 datasets, on the other hand, experience significant performance degradation on *NoNodeFtrs* and *RandFtrs*, yet these drops are recovered in *NodeDeg*.

This indicates that T-2 datasets have tasks for which structural summary information is sufficient, perhaps due to homophily.

— **Representative set.** Many datasets have very close sensitivity profiles, thus factoring in also the graph size and original AUROC (avoiding saturated datasets), we make the following recommendation: WebKB-Wis, Actor from T-1; WikiNet-cham, WikiCS, Flickr from T-2; WikiNet-squir, Twitch-EN, Github from T-3.

T-3: Indifference to node and structure removal. The datasets in T-3 are relatively insensitive to perturbations of graph structure and also to the removal of node features (*NoNodeFtrs* and *NodeDeg*). For example, the Amazon datasets (Am-Phot and Am-Comp) always achieve near perfect classification performance regardless of the perturbations applied, suggesting redundancy between node features and graph structure for the corresponding tasks. For these datasets, in particular, GitHub, Am, and Twitch, more sophisticated, or combinations of, perturbations might be needed to gauge their essential characteristics.

T-2: Rich node features but substitutable for structural (summary) information. T-2 contains a broad spectrum of datasets from citation networks (CF), social networks (Coau, FBPP, LFMA), to web pages (WikiNet, WikiCS). The considerable performance decrease due to node feature removal suggests the relevance of the node features for their tasks. For example, it is not surprising that the binary bag-of-words features of CF datasets provide relevant information to classify papers into different fields of research, as one might expect some keywords to appear more likely in one field than in another. Furthermore, using the one-hot encoded node degrees (*NodeDeg*) always results in better performance over *NoNodeFtrs*. And in many cases such as Facebook (FBPP), *NodeDeg* nearly retains the baseline performance, suggesting the relevance of node degree information, as a form of structural summary, for the respective tasks.

WebKB-Tex, although clustered into T-2 is more of an outlier that does not clearly fit into any of the existing clusters. As we will discuss more in T-1, WebKB-Tex considerably benefits from *HighPass*, while *LowPass* and *MidPass* severely decrease its performance.

T-1: Heterophilic datasets. Three of the four datasets in T-1 (Actor, WebKB-Cor, and WebKB-Wis) are commonly referred to as heterophilic datasets [101, 91]. While WebKB-Tex (T-2) is also known to be heterophilic, it is isolated from T-1 mainly due to its insensitivity to node feature removal, suggesting the structure alone is sufficient for its prediction task.

Our results show that in heterophilic datasets such as T-1 and WebKB-Tex, *LowPass* node feature filtering, realized by local aggregation (Eq. 4.2.3), significantly degrades the performance, unlike other homophilic datasets. By contrast, *HighPass* results in better performance than *LowPass*. In the case of WekbKB-Tex, *HighPass* significantly improves the performance over the baseline. This observation is related to recent findings [91] that in

the case of extreme heterophily, local information, this time in form of the neighborhood patterns, may suffice to infer the correct node labels.

Finally, despite heterophilic datasets [91, 2, 141, 101] attracting much recent attention, this type of datasets (`T-1` and `WebKB-Tex`) is lacking in availability compared to the others (`T-{2,3}`), which exhibit homophily but with different levels of reliance on node features. Thus, there is a need to collect and generate more real-world heterophilic datasets.

4.5. Discussion

Our results quantify the extent to which graph features or structures are more important for the downstream tasks, an important question brought up in classical works on graph kernels [84, 129]. We observed that more than half of the datasets contain rich node features. On average, excluding these features reduces GNN prediction performance more than excluding the entire graph structures, especially for transductive node-level tasks. Furthermore, low-frequency information in node features appears to be essential in most datasets that rely on node features. Historically, most graph data aimed to capture closeness among entities, which has prompted development of local aggregation approaches, such as label propagation, personalized page rank, and diffusion kernels [81, 24], all of which share a common principle of low pass filtering. High-frequency information, on the other hand, may be important in recently emerging application areas, such as combinatorial optimization, logical reasoning or biochemical property prediction, which require complex non-local representations.

Further, despite the recent interest in development of new methods that could leverage long-range dependencies and heterophily, the availability of adequate benchmarking datasets remains lacking or less readily accessible. Meanwhile, some recent efforts such as GraphWorld [106] aim to comprehensively profile a GNN's performance using a collection of synthetic datasets that cover an entire parametric space. Notably, our analysis demonstrates that synthetic tasks do not fully resemble the complexity of real-world applications. Hence, benchmarking made purely by synthetic datasets should be taken with caution, as the behavior might not be representative of real-world scenarios.

As a comprehensive benchmarking framework, our work provides several potential use cases beyond the taxonomy analysis presented here. One such usage is understanding the characteristics of any new datasets and how they are related to existing ones. For example, DeezerEurope (`DzEu`) is a relatively new dataset [120] that is less commonly benchmarked and studied than the other datasets we consider. The inclusion of `DzEu` in `T-1` suggested its heterophilic nature, which indeed has been recently demonstrated [89]. On the other hand, since the sensitivity profiles naturally suggest the invariances that are important for different datasets from a practical standpoint, they could provide valuable guidance to the development of self-supervised learning and data augmentations for GNNs [158].

Finally, we observed that overall patterns in *sensitivity profiles* remain similar regardless whether we used GCN, GIN or the other 4 models to derive them. Subtle differences in sensitivity profiles w.r.t. different GNN models are not only expected but also desired when comparing models that have distinct levels of expressivity. While we expect overall patterns to be similar, more expressive models should provide enhanced resolution. One could then contrast taxonomization w.r.t. first-order GNNs (such as those we used) with provably more expressive higher-order GNNs, Transformer-based models with global attention, and others. We hope our work will also inspire future work to empirically validate expressivity of new graph learning methods in this vein, beyond classical benchmarking.

Conclusion

As the field of graph representation learning rapidly grows, the importance of setting up comprehensive and informative benchmarking processes that evaluate these developments increases in tandem. Nevertheless, it is still difficult today to make informed decisions about benchmarking data selection in the relatively short history of graph learning, primarily due to a limited understanding of how datasets differ in their encoding and propagation of information.

The first contribution of this thesis is to present a comprehensive overview of how we have arrived at graph neural networks as the presiding paradigm in geometric deep learning, from a broadly historical perspective. We showed that development of graph learning as a field has relied on a unique amalgamation of research areas in computer science, drawing from graph theory, signal processing and stochastic models in addition to machine learning. In our further enquiry into machine learning on graphs, we have covered traditional machine learning algorithms such as kernel methods in addition to discussing the influences of pre-GNN deep learning methods of recurrent, recursive and convolutional neural networks.

We then reviewed a number of GNN algorithms that have proven both influential and performative, and thus (a) commonly serve as points of reference for researchers developing new GNN algorithms, and (b) some of which we have in turn used when constructing our taxonomy. In doing so, we also took care to provide models that cover a spectrum of inductive biases such as spatial and spectral convolutions, edge gates, attention and graph isomorphism. We have also made a conscious attempt to present these models from the lens of neural message passing, in the spirit of Gilmer et al. [51] and Hamilton [59].

Our second contribution is to provide a much-needed analysis of current shortcomings of benchmarking processes in graph representation learning. We follow a top-down approach and conjecture criteria on what constitutes a set of benchmark datasets appropriate in machine learning, and identify the issue of "insufficient data coverage", i.e. seemingly different benchmarks testing for similar properties of a machine learning algorithm and offer no differentiability. We argue that data coverage is not a well-understood problem, yet its harmful effects on generalization performance of deep learning algorithms are unmistakable. We then zero in on benchmarking problems in GRL, where most traditional graph benchmarks consist

of small graphs and lack unified experimental protocols which lead to noisy benchmarking procedures. We then shift our focus to how data coverage issues manifest themselves in GRL benchmarking and how these issues have been partially addressed to date. Our observation is that the criteria for "variety" and coverage in graph datasets, as proposed by recent literature, is usually limited to application domains and basic graph data metrics such as graph size, which are insufficient to ensure that benchmarks processes are able to statistically separate different GNN models.

As our third and main contribution in this work, we present a solution to this problem by constructing a method for defining and analyzing the information flow in graph data, and come up with the most complete taxonomy of graph datasets in literature based on the analysis of this information flow on a multitude of datasets, covering graph and node prediction tasks, on inductive and transductive datasets. The core principle of our method is to gauge the essential characteristics of a given dataset with respect to its accompanying prediction task by inspecting the downstream effects caused by perturbing its graph data. The resulting sensitivities to the diverse set of perturbations serve as "fingerprints" that allow to identify datasets with similar characteristics.

Finally, using our taxonomy, we derive several insights into the current common benchmarks used in the field of graph representation learning, and make recommendations on selection of representative benchmarking suits. Our analysis also puts forward a foundation for evaluating new benchmarking datasets that will likely emerge in the field. With this in mind we have taken care to ensure that our methodology provides not just a snapshot of the current state of benchmarking in graph representation learning, but also a dynamic framework that can grow as the field progresses further by its application to future graph datasets and models. We also facilitate the dynamic nature of our methodology by providing the source code for our taxonomy; we hope that this will make it easier for researchers to both use our framework in developing their benchmarks, and further extend it to suit their needs.

Limitations and Future Work. Our perturbation-based approach has a fundamental limitation in the sense that we cannot test the significance of a property that we cannot systematically perturb or that the reference GNN model cannot capture. Therefore, designing more sophisticated perturbation strategies for gauging specific characteristics and relations could bring further insight into the datasets and GNN models alike. New perturbations may gauge the usefulness of geometric substructures such as cycles [6] or the effects of graph bottlenecks, e.g., by rewiring graphs to modify their "curvatures" [141]. Other interesting perturbations could include graph sparsification (edge removal) [135] and graph pooling/coarsening (edge contraction) [16, 8].

A number of OGB node-level datasets are not included in this study due to memory cost of typical MPNNs. Conducting an analysis based on recent scalable GNN models [40]

would be an interesting avenue of future research. Further, we only considered classification tasks, omitting regression tasks, as their evaluation metrics are not easily comparable. One way to circumvent this issue would be to quantize regression tasks into classification tasks by binning their continuous targets. Additionally, we disregarded edge features in two OGB molecular datasets we used. In a future work, edge features could be leveraged by an edge-feature aware generalization of MPNNs. The importance of edge features can then be analyzed by introducing new edge-feature perturbations. We also limited our analysis to node-level and graph-level tasks, but this framework could be further extended to link-prediction or edge-level tasks. While our perturbations could be used in this new scenario as well, new perturbations, such as the above-mentioned graph sparsification, would need to be considered. Similarly, hallmark models for link and relation predictions, outside MPNNs, should be considered.

References

[1] A.K.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 orbital energies and hydrophobicity. *Journal of medicinal chemistry*, 34(2):786–797, 1991.

[2] U. Alon and E. Yahav. On the bottleneck of graph neural networks and its practical implications, 2021.

[3] James Atwood and Don Towsley. Diffusion-convolutional neural networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL `https://proceedings.neurips.cc/paper/2016/file/390e982518a50e280d8e2b535462ec1f-Paper.pdf`.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014. URL `https://arxiv.org/abs/1409.0473`.

[5] Adriano Barbosa-Silva, Simon Ott, Kathrin Blagec, Jan Brauner, and Matthias Samwald. Mapping global dynamics of benchmark creation and saturation in artificial intelligence. *arXiv preprint arXiv: Arxiv-2203.04592*, 2022.

[6] B. Bevilacqua, F. Frasca, D. Lim, B. Srinivasan, C. Cai, G. Balamurugan, M.M. Bronstein, and H. Maron. Equivariant subgraph aggregation networks, 2021.

[7] Lucas Beyer, Olivier J. Hénaff, Alexander Kolesnikov, Xiaohua Zhai, and Aäron van den Oord. Are we done with imagenet? *arXiv preprint arXiv: Arxiv-2006.07159*, 2020.

[8] C. Bodnar, C. Cangea, and P. Liò. Deep graph mapper: Seeing graphs through the neural lens. *Frontiers in Big Data*, 4, June 2021.

[9] A. Bojchevski and S. Günnemann. Deep gaussian embedding of graphs: Unsupervised inductive learning via ranking. In *Proc. of ICLR*, 2018.

[10] K.M. Borgwardt, C.S. Ong, S. Schönauer, SVN Vishwanathan, A.J. Smola, and H. Kriegel. Protein function prediction via graph kernels. *Bioinformatics*, 21:i47–i56, 2005.

[11] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, pages 144–152, 1992.

[12] Xavier Bresson and Thomas Laurent. Residual gated graph convnets. *ICLR*, 2018.

[13] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1):107–117, 1998. ISSN 0169-7552. doi: https://doi.org/10.1016/S0169-7552(98)00110-X. URL `https://www.sciencedirect.com/science/article/pii/S016975529800110X`. Proceedings of the Seventh International World Wide Web Conference.

[14] M.M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4): 18–42, Jul 2017. ISSN 1558-0792.

[15] M.M. Bronstein, J. Bruna, T. Cohen, and P. Veličković. Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges, 2021.

[16] N. Brugnone, A. Gonopolskiy, M.W. Moyle, M. Kuchroo, D. Dijk, K.R. Moon, D. Colon-Ramos, G. Wolf, M.J. Hirn, and S. Krishnaswamy. Coarse graining of data via inhomogeneous diffusion condensation. *IEEE Big Data*, Dec 2019.

[17] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs, 2013. URL `https://arxiv.org/abs/1312.6203`.

[18] Joy Buolamwini and Timnit Gebru. Gender shades: Intersectional accuracy disparities in commercial gender classification. In Sorelle A. Friedler and Christo Wilson, editors, *Proceedings of the 1st Conference on Fairness, Accountability and Transparency*, volume 81 of *Proceedings of Machine Learning Research*, pages 77–91. PMLR, 23–24 Feb 2018. URL `https://proceedings.mlr.press/v81/buolamwini18a.html`.

[19] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 40(4): 834–848, 2018. doi: 10.1109/TPAMI.2017.2699184. URL `https://doi.org/10.1109/TPAMI.2017.2699184`.

[20] M. Chen, Z. Wei, Z. Huang, B. Ding, and Y. Li. Simple and deep graph convolutional networks. In *Proceedings of the 37th International Conference on Machine Learning*, 2020.

[21] W. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C. Hsieh. Cluster-gcn. *Proc. of 25th SIGKDD*, 2019.

[22] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. *arXiv preprint arXiv: Arxiv-1905.07953*, 2019.

[23] R.R. Coifman and M. Maggioni. Diffusion wavelets. *Applied and computational harmonic analysis*, 21(1):53–94, 2006.

[24] L. Cowen, T. Ideker, B.J. Raphael, and R. Sharan. Network propagation: a universal amplifier of genetic associations. *Nat. Rev. Gene.*, 18(9):551–562, 2017.

[25] Kate Crawford and Trevor Paglen. Excavating ai: the politics of images in machine learning training sets. *AI & SOCIETY*, 2021. doi: 10.1007/s00146-021-01162-8. URL `https://link.springer.com/article/10.1007/s00146-021-01162-8/fulltext.html`.

[26] Alexander D'Amour, Katherine Heller, Dan Moldovan, Ben Adlam, Babak Alipanahi, Alex Beutel, Christina Chen, Jonathan Deaton, Jacob Eisenstein, Matthew D. Hoffman, Farhad Hormozdiari, Neil Houlsby, Shaobo Hou, Ghassen Jerfel, Alan Karthikesalingam, Mario Lucic, Yian Ma, Cory McLean, Diana Mincu, Akinori Mitani, Andrea Montanari, Zachary Nado, Vivek Natarajan, Christopher Nielson, Thomas F. Osborne, Rajiv Raman, Kim Ramasamy, Rory Sayres, Jessica Schrouff, Martin Seneviratne, Shannon Sequeira, Harini Suresh, Victor Veitch, Max Vladymyrov, Xuezhi Wang, Kellie Webster, Steve Yadlowsky, Taedong Yun, Xiaohua Zhai, and D. Sculley. Underspecification presents challenges for credibility in modern machine learning. *arXiv preprint arXiv: Arxiv-2011.03395*, 2020.

[27] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in NeurIPS*, volume 29, pages 3844–3852, 2016.

[28] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, K. Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[29] John Denker, W. Gardner, Hans Graf, Donnie Henderson, R. Howard, W. Hubbard, L. D. Jackel, Henry Baird, and Isabelle Guyon. Neural network recognizer for handwritten zip code digits. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 1. Morgan-Kaufmann, 1988. URL `https://proceedings.neurips.cc/paper/1988/file/a97da629b098b75c294dffdc3e463904-Paper.pdf`.

[30] Lucas Dixon, John Li, Jeffrey Sorensen, Nithum Thain, and Lucy Vasserman. Measuring and mitigating unintended bias in text classification. 2018.

[31] P.D. Dobson and A.J. Doig. Distinguishing enzyme structures from non-enzymes without alignments. *J. of Mol. Bio.*, 330(4):771–783, 2003.

[32] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 647–655, Bejing, China, 22–24 Jun 2014. PMLR. URL

https://proceedings.mlr.press/v32/donahue14.html.

[33] Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs. *CoRR*, abs/2012.09699, 2020. URL https://arxiv.org/abs/2012.09699.

[34] V.P. Dwivedi, C.K. Joshi, T. Laurent, Y. Bengio, and X. Bresson. Benchmarking Graph Neural Networks. *arXiv:2003.00982*, 2020.

[35] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Jacob Steinhardt, and Aleksander Madry. Identifying statistical bias in dataset replication. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 2922–2932. PMLR, 2020. URL http://proceedings.mlr.press/v119/engstrom20a.html.

[36] Mark Everingham, Luc Van Gool, Christopher Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88:303–338, 06 2010. doi: 10.1007/s11263-009-0275-4.

[37] Li Fei-Fei, Rob Fergus, and Pietro Perona. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. *Computer Vision and Image Understanding*, 106(1):59–70, 2007. ISSN 1077-3142. doi: https://doi.org/10.1016/j.cviu.2005.09.012. URL https://www.sciencedirect.com/science/article/pii/S1077314206001688. Special issue on Generative Model Based Vision.

[38] A. Feragen, N. Kasenburg, J. Petersen, M. de Bruijne, and K. Borgwardt. Scalable kernels for graphs with continuous attributes. In *Adv. in NeurIPS*, volume 26, 2013.

[39] M. Fey and J.E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Repr. Learning on Graphs and Manifolds*, 2019.

[40] M. Fey, J. E. Lenssen, F. Weichert, and J. Leskovec. Gnnautoscale: Scalable and expressive graph neural networks via historical embeddings, 2021.

[41] M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak mathematical journal*, 25(4):619–633, 1975.

[42] P. Frasconi, M. Gori, and A. Sperduti. A general framework for adaptive processing of data structures. *IEEE Transactions on Neural Networks*, 9(5):768–786, September 1998. ISSN 10459227. doi: 10.1109/72.712151. URL http://ieeexplore.ieee.org/document/712151/.

[43] S. Freitas, Y. Dong, J. Neil, and D.H. Chau. A large-scale database for graph representation learning. In *Adv. in NeurIPS*, 2021.

[44] Kunihiko Fukushima. Visual feature extraction by a multilayered network of analog threshold elements. *IEEE Transactions on Systems Science and Cybernetics*, 5(4): 322–333, 1969. doi: 10.1109/TSSC.1969.300225.

[45] Hongyang Gao, Zhengyang Wang, and Shuiwang Ji. Large-scale learnable graph convolutional networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, page 1416–1424, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355520. doi: 10.1145/3219819.3219947. URL https://doi.org/10.1145/3219819.3219947.

[46] Nikhil Garg, Londa Schiebinger, Dan Jurafsky, and James Zou. Word embeddings quantify 100 years of gender and ethnic stereotypes. *Proceedings of the National Academy of Sciences*, 115(16):E3635–E3644, 2018.

[47] J. Gasteiger, A. Bojchevski, and S. Günnemann. Predict then propagate: Graph neural networks meet personalized pagerank. In *International Conference on Learning Representations*, 2018.

[48] Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna M. Wallach, Hal Daumé III, and Kate Crawford. Datasheets for datasets. *Commun. ACM*, 64(12):86–92, 2021. doi: 10.1145/3458723. URL https://doi.org/10.1145/3458723.

[49] R. Geiger, Kevin Yu, Yanlai Yang, Mindy Dai, Jie Qiu, Rebekah Tang, and Jenny Huang. Garbage in, garbage out? do machine learning application papers in social computing report where human-labeled training data comes from? *fat**, 2019. doi: 10.1145/3351095.3372862.

[50] C. Lee Giles, Kurt D. Bollacker, and Steve Lawrence. Citeseer: an automatic citation indexing system. In *INTERNATIONAL CONFERENCE ON DIGITAL LIBRARIES*, pages 89–98. ACM Press, 1998.

[51] J. Gilmer, S.S. Schoenholz, P.F. Riley, O. Vinyals, and G.E. Dahl. Neural message passing for quantum chemistry, 2017.

[52] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[53] M. Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. volume 2, pages 729 – 734 vol. 2, 01 2005. ISBN 0-7803-9048-2. doi: 10.1109/IJCNN.2005.1555942.

[54] C.S. Greene, A. Krishnan, A.K. Wong, E. Ricciotti, R.A. Zelaya, D.S. Himmelstein, R. Zhang, B.M. Hartmann, E. Zaslavsky, S.C. Sealfon, et al. Understanding multicellular function and disease with human tissue-specific networks. *Nature genetics*, 47(6): 569–576, 2015.

[55] Thomas Gärtner. A survey of kernels for structured data. *ACM SIGKDD Explorations Newsletter*, 5(1):49–58, July 2003. ISSN 1931-0145, 1931-0153. doi: 10.1145/959242.959248. URL https://dl.acm.org/doi/10.1145/959242.959248.

[56] A. Hagberg, P. Swart, and s.D. Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los

Alamos, NM (United States), 2008.

[57] L. Hagen and A.B. Kahng. New spectral methods for ratio cut partitioning and clustering. *IEEE transactions on computer-aided design of integrated circuits and systems*, 11(9):1074–1085, 1992.

[58] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv: Arxiv-1709.05584*, 2017.

[59] W.L. Hamilton. *Graph Representation Learning*. Morgan & Claypool, 2020.

[60] W.L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1025–1035, 2017.

[61] David K. Hammond, Pierre Vandergheynst, and Rémi Gribonval. Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis*, 30(2):129–150, 2011. ISSN 1063-5203. doi: https://doi.org/10.1016/j.acha.2010.04.005. URL `https://www.sciencedirect.com/science/article/pii/S1063520310000552`.

[62] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv: Arxiv-1512.03385*, 2015.

[63] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2980–2988, 2017. doi: 10.1109/ICCV.2017.322.

[64] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, nov 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL `https://doi.org/10.1162/neco.1997.9.8.1735`.

[65] P.W. Holland, K.B. Laskey, and S. Leinhardt. Stochastic blockmodels: First steps. *Social Networks*, 5(2):109–137, 1983. ISSN 0378-8733.

[66] P. Holme and B.J. Kim. Growing scale-free networks with tunable clustering. *Physical Review E*, 65(2), Jan 2002. ISSN 1095-3787. doi: 10.1103/physreve.65.026107.

[67] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(89)90020-8. URL `https://www.sciencedirect.com/science/article/pii/0893608089900208`.

[68] W. Hu, M.s Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *Adv. in NeurIPS 33*, 2020.

[69] W. Hu, M. Fey, H. Ren, M. Nakata, Y. Dong, and J. Leskovec. OGB-LSC: A large-scale challenge for machine learning on graphs. In *35th Conference on Neural Information Processing Systems: Datasets and Benchmarks Track*, 2021.

[70] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay S. Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=HJlWWJSFDH.

[71] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3296–3297, 2017. doi: 10.1109/CVPR.2017.351.

[72] Q. Huang, H. He, A. Singh, S. Lim, and A. Benson. Combining label propagation and simple models out-performs graph neural networks. In *International Conference on Learning Representations*, 2020.

[73] Ben Hutchinson, Vinodkumar Prabhakaran, Emily Denton, Kellie Webster, Yu Zhong, and Stephen Denuyl. Social biases in NLP models as barriers for persons with disabilities. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 5491–5501. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.acl-main.487. URL https://doi.org/10.18653/v1/2020.acl-main.487.

[74] J. Irion and N. Saito. Efficient approximation and denoising of graph signals using the multiscale basis dictionaries. *IEEE Transactions on Signal and Information Processing over Networks*, 3(3):607–616, 2016.

[75] Sergei Ivanov, Sergei Sviridov, and Evgeny Burnaev. Understanding isomorphism bias in graph data sets. *arXiv preprint arXiv: Arxiv-1910.12091*, 2019.

[76] Shengli Jiang and Prasanna Balaprakash. Graph neural network architecture search for molecular property prediction. In Xintao Wu, Chris Jermaine, Li Xiong, Xiaohua Hu, Olivera Kotevska, Siyuan Lu, Weija Xu, Srinivas Aluru, Chengxiang Zhai, Eyhab Al-Masri, Zhiyuan Chen, and Jeff Saltz, editors, *2020 IEEE International Conference on Big Data (IEEE BigData 2020), Atlanta, GA, USA, December 10-13, 2020*, pages 1346–1353. IEEE, 2020. doi: 10.1109/BigData50022.2020.9378060. URL https://doi.org/10.1109/BigData50022.2020.9378060.

[77] Hisashi Kashima, Koji Tsuda, and Akihiro Inokuchi. Marginalized kernels between labeled graphs. In Tom Fawcett and Nina Mishra, editors, *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA*, pages 321–328. AAAI Press, 2003. URL http://www.aaai.org/Library/ICML/2003/icml03-044.php.

[78] D.P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[79] T.N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *Proc. of ICLR*, 2017.

[80] Bernard Koch, Emily Denton, Alex Hanna, and Jacob G. Foster. Reduced, reused and recycled: The life of a dataset in machine learning research. *CoRR*, abs/2112.01716, 2021. URL `https://arxiv.org/abs/2112.01716`.

[81] S. Köhler, S. Bauer, D. Horn, and P.N. Robinson. Walking the interactome for prioritization of candidate disease genes. *The American Journal of Human Genetics*, 82(4): 949–958, April 2008.

[82] Risi Imre Kondor and John D. Lafferty. Diffusion kernels on graphs and other discrete input spaces. In *Proceedings of the Nineteenth International Conference on Machine Learning*, ICML '02, page 315–322, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc. ISBN 1558608737.

[83] Simon Kornblith, Jonathon Shlens, and Quoc V. Le. Do better imagenet models transfer better? In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 2661–2671. Computer Vision Foundation / IEEE, 2019. doi: 10.1109/CVPR.2019.00277. URL `http://openaccess.thecvf.com/content_CVPR_2019/html/Kornblith_Do_Better_ImageNet_Models_Transfer_Better_CVPR_2019_paper.html`.

[84] N.M. Kriege, F.D. Johansson, and C. Morris. A survey on graph kernels. *Applied Network Science*, 5(1):1–42, 2020.

[85] H. W. Kuhn and A. W. Tucker. Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability, 1950*, pages 481–492, Berkeley and Los Angeles, 1951. University of California Press.

[86] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989. doi: 10.1162/neco.1989.1.4.541.

[87] Yann Lecun. A theoretical framework for back-propagation. In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School, CMU, Pittsburg, PA*, pages 21–28. Morgan Kaufmann, 1988.

[88] K. Lemont, L.B. Kier, L.H. Hall, and Elsevier Science & Technology (Firm). *Molecular Connectivity in Chemistry and Drug Research*. Medicinal chemistry. Academic Press, 1976. ISBN 9780124065604. URL `https://books.google.com.tr/books?id=gQfwAAAAMAAJ`.

[89] D. Lim, F. Hohne, X. Li, S. Linda H., V. Gupta, O. Bhalerao, and S. Lim. Large scale learning on non-homophilous graphs: New benchmarks and strong simple methods, 2021.

[90] Derek Lim, Joshua Robinson, Lingxiao Zhao, Tess Smidt, Suvrit Sra, Haggai Maron, and Stefanie Jegelka. Sign and basis invariant networks for spectral graph representation learning, 2022.

[91] Y. Ma, X. Liu, N. Shah, and J. Tang. Is homophily a necessity for graph neural networks?, 2021.

[92] Andrew Mccallum, Kamal Nigam, and Jason Rennie. Automating the construction of internet portals. 03 2000.

[93] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943. ISSN 1522-9602. doi: 10.1007/BF02478259. URL `https://doi.org/10.1007/BF02478259`.

[94] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. A survey on bias and fairness in machine learning. *arXiv preprint arXiv: Arxiv-1908.09635*, 2019.

[95] P. Mernyei and C. Cangea. Wiki-cs: A wikipedia-based benchmark for graph neural networks, 2020.

[96] Y. Min, F. Wenkel, and G. Wolf. Scattering GCN: Overcoming Oversmoothness in Graph Convolutional Networks. In *Adv. in NeurIPS 33*, pages 14498–14508, 2020.

[97] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969.

[98] C. Morris, N.M. Kriege, K. Kersting, and P. Mutzel. Faster kernels for graphs with continuous attributes via hashing. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 1095–1100, 2016.

[99] 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 *Proceedings of the AAAI Conference on AI*, volume 33, pages 4602–4609, 2019.

[100] 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. In *ICML 2020 GRL+ Workshop*, 2020.

[101] H. Mostafa, M. Nassar, and S. Majumdar. On local aggregation in heterophilic graphs. *arXiv:2106.03213*, 2021.

[102] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2014–2023, New York, New York, USA, 20–22 Jun 2016. PMLR. URL `https://proceedings.mlr.press/v48/niepert16.html`.

[103] Antonio Ortega. *Introduction to Graph Signal Processing*. Cambridge University Press, 2022. doi: 10.1017/9781108552349.

[104] Antonio Ortega, Pascal Frossard, Jelena Kovačević, José M. F. Moura, and Pierre Vandergheynst. Graph signal processing: Overview, challenges, and applications. *Proceedings of the IEEE*, 106(5):808–828, 2018. doi: 10.1109/JPROC.2018.2820126.

[105] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. URL `http://ilpubs.stanford.edu:8090/422/`. Previous number = SIDL-WP-1999-0120.

[106] J. Palowitch, A. Tsitsulin, B. Mayer, and B. Perozzi. GraphWorld: Fake graphs bring real insights for GNNs. *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022.

[107] 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 H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[108] Amandalynne Paullada, Inioluwa Deborah Raji, Emily M. Bender, Emily Denton, and Alex Hanna. Data and its (dis)contents: A survey of dataset development and use in machine learning research. *Patterns*, 2(11):100336, 2021. ISSN 2666-3899. doi: https://doi.org/10.1016/j.patter.2021.100336. URL `https://www.sciencedirect.com/science/article/pii/S2666389921001847`.

[109] H. Pei, B. Wei, K.C. Chang, Y. Lei, and B. Yang. Geom-GCN: Geometric graph convolutional networks. In *Proc. of ICLR*, 2020.

[110] Fernando J Pineda. Dynamics and architecture for neural computation. *Journal of Complexity*, 4(3):216–245, September 1988. ISSN 0885064X. doi: 10.1016/0885-064X(88)90021-0. URL `https://linkinghub.elsevier.com/retrieve/pii/0885064X88900210`.

[111] Dejan Plavšić, Sonja Nikolić, Nenad Trinajstić, and Zlatko Mihalić. On the harary index for the characterization of chemical graphs. *Journal of Mathematical Chemistry*, 12(1):235–250, Dec 1993. doi: 10.1007/BF01164638. URL `https://doi.org/10.1007/BF01164638`.

[112] Inioluwa Deborah Raji, Emily Denton, Emily M. Bender, Alex Hanna, and Amandalynne Paullada. AI and the everything in the whole wide world benchmark. In Joaquin Vanschoren and Sai-Kit Yeung, editors, *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, 2021. URL `https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/084b6fbb10729ed4da8c3d3f5a3ae7c9-Abstract-round2.html`.

[113] Ladislav Rampášek, Mikhail Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. Recipe for a general, powerful, scalable graph transformer, 2022.

[114] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: An astounding baseline for recognition. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 512–519, 2014. doi: 10.1109/CVPRW.2014.131.

[115] Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do cifar-10 classifiers generalize to cifar-10? *arXiv preprint arXiv: Arxiv-1806.00451*, 2018.

[116] Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do imagenet classifiers generalize to imagenet? In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 5389–5400. PMLR, 2019. URL `http://proceedings.mlr.press/v97/recht19a.html`.

[117] F. Rosenblatt. The perceptron - a perceiving and recognizing automaton. Technical Report 85-460-1, Cornell Aeronautical Laboratory, Ithaca, New York, January 1957.

[118] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL `https://networkrepository.com`.

[119] B. Rozemberczki and R. Sarkar. Characteristic functions on graphs: Birds of a feather, from statistical descriptors to parametric models. In *Proc. of 29th ACM Int'l Conf. on Information & Knowledge Management*, pages 1325–1334, 2020.

[120] B. Rozemberczki, C. Allen, and R. Sarkar. Multi-scale attributed node embedding. *Journal of Complex Networks*, 9(2):cnab014, 2021.

[121] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv: Arxiv-1609.04747*, 2016.

[122] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, Oct 1986. doi: 10.1038/323533a0. URL `https://doi.org/10.1038/323533a0`.

[123] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vis.*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y. URL `https://doi.org/10.1007/s11263-015-0816-y`.

[124] Aliaksei Sandryhaila and José M. F. Moura. Discrete signal processing on graphs: Frequency analysis. *IEEE Trans. Signal Process.*, 62(12):3042–3054, 2014. doi: 10.1109/TSP.2014.2321121. URL `https://doi.org/10.1109/TSP.2014.2321121`.

[125] Ryoma Sato. A survey on the expressive power of graph neural networks. *arXiv preprint arXiv: Arxiv-2003.04078*, 2020.

[126] F. Scarselli, M. Gori, Ah Chung Tsoi, M. Hagenbuchner, and G. Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, January 2009. ISSN 1045-9227, 1941-0093. doi: 10.1109/TNN.2008.2005605. URL http://ieeexplore.ieee.org/document/4700287/.

[127] Morgan Klaus Scheuerman, Alex Hanna, and Emily Denton. Do datasets have politics? disciplinary values in computer vision dataset development. *Proc. ACM Hum.-Comput. Interact.*, 5(CSCW2), oct 2021. doi: 10.1145/3476058. URL https://doi.org/10.1145/3476058.

[128] David Schlangen. Targeting the benchmark: On methodology in current natural language processing research. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 2: Short Papers), Virtual Event, August 1-6, 2021*, pages 670–674. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.acl-short.85. URL https://doi.org/10.18653/v1/2021.acl-short.85.

[129] T. Schulz and P. Welke. On the necessity of graph kernel baselines. In *ECML-PKDD, GEM workshop*, volume 1, page 6, 2019.

[130] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93, Sep. 2008. doi: 10.1609/aimag.v29i3.2157. URL https://ojs.aaai.org/index.php/aimagazine/article/view/2157.

[131] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, USA, 2014. ISBN 1107057132.

[132] Shreya Shankar, Yoni Halpern, Eric Breck, James Atwood, Jimbo Wilson, and D. Sculley. No classification without representation: Assessing geodiversity issues in open data sets for the developing world. *arXiv preprint arXiv: Arxiv-1711.08536*, 2017.

[133] O. Shchur, M. Mumme, A. Bojchevski, and S. Günnemann. Pitfalls of graph neural network evaluation. *NeurIPS 2018 R2L workshop*, 2018.

[134] A. Sperduti and A. Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, May 1997. ISSN 1045-9227, 1941-0093. doi: 10.1109/72.572108. URL https://ieeexplore.ieee.org/document/572108/.

[135] D.A. Spielman and S. Teng. Spectral sparsification of graphs, 2010.

[136] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL http:

//jmlr.org/papers/v15/srivastava14a.html.

[137] Ljubisa Stankovic, Danilo Mandic, Milos Dakovic, Milos Brajovic, Bruno Scalzo, and Anthony G. Constantinides. Graph signal processing - part ii: Processing and analyzing signals on graphs. *arXiv preprint arXiv: Arxiv-1909.10325*, 2019.

[138] Ljubisa Stankovic, Danilo Mandic, Milos Dakovic, Milos Brajovic, Bruno Scalzo, and Tony Constantinides. Graph signal processing - part i: Graphs, graph spectra, and spectral clustering. *arXiv preprint arXiv: Arxiv-1907.03467*, 2019.

[139] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, 2016. doi: 10.1109/CVPR.2016.308.

[140] D. Szklarczyk, A. Franceschini, S. Wyder, K. Forslund, D. Heller, J. Huerta-Cepas, M. Simonovic, A. Roth, A. Santos, K.P. Tsafou, et al. STRING v10: protein–protein interaction networks, integrated over the tree of life. *Nucleic acids research*, 43(D1): D447–D452, 2015.

[141] J. Topping, F.D. Giovanni, B.P. Chamberlain, X. Dong, and M.M. Bronstein. Understanding over-squashing and bottlenecks on graphs via curvature, 2021.

[142] Antonio Torralba and Alexei A. Efros. Unbiased look at dataset bias. In *CVPR 2011*, pages 1521–1528, 2011. doi: 10.1109/CVPR.2011.5995347.

[143] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Andrew Ilyas, and Aleksander Madry. From ImageNet to image classification: Contextualizing progress on benchmarks. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 9625–9635. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/tsipras20a.html.

[144] Ah Chung Tsoi, Gianni Morini, Franco Scarselli, Markus Hagenbuchner, and Marco Maggini. Adaptive ranking of web pages. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, page 356–365, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136803. doi: 10.1145/775152.775203. URL https://doi.org/10.1145/775152.775203.

[145] Ah Chung Tsoi, Markus Hagenbuchner, and Franco Scarselli. Computing customized page ranks. *ACM Trans. Internet Technol.*, 6(4):381–414, nov 2006. ISSN 1533-5399. doi: 10.1145/1183463.1183466. URL https://doi.org/10.1145/1183463.1183466.

[146] K. Tsuda, T. Kin, and K. Asai. Marginalized kernels for biological sequences. *Bioinformatics*, 18(Suppl 1):S268–S275, July 2002. doi: 10.1093/bioinformatics/18.suppl_1.s268. URL https://doi.org/10.1093/bioinformatics/18.suppl_1.s268.

[147] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*,

abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

[148] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. In *The 6th ICLR*, 2018.

[149] N. Wale, I.A. Watson, and G. Karypis. Comparison of descriptor spaces for chemical compound retrieval and classification. *Knowledge and Information Systems*, 14(3): 347–375, 2008.

[150] Qi Wang, Yue Ma, Kun Zhao, and Yingjie Tian. A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, 9(2):187–212, Apr 2022. ISSN 2198-5812. doi: 10.1007/s40745-020-00253-5. URL https://doi.org/10.1007/s40745-020-00253-5.

[151] J.H. Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

[152] D.J. Watts and S.H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[153] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997. doi: 10.1109/4235.585893.

[154] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6861–6871. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/wu19e.html.

[155] 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(2):513–530, 2018.

[156] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S.Y. Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[157] Zhenqin Wu, Bharath Ramsundar, Evan N. Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S. Pappu, Karl Leswing, and Vijay Pande. Moleculenet: a benchmark for molecular machine learning. *Chem. Sci.*, 9:513–530, 2018. doi: 10.1039/C7SC02664A. URL http://dx.doi.org/10.1039/C7SC02664A.

[158] Y. Xie, Z. Xu, J. Zhang, Z. Wang, and S. Ji. Self-supervised learning of graph neural networks: A unified review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[159] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *Proc. of ICLR*, 2019.

[160] P. Yanardag and S.V.N. Vishwanathan. Deep graph kernels. In *Proc. of 21th SIGKDD*, pages 1365–1374, 2015.

[161] J. You, R. Ying, and J. Leskovec. Design space for graph neural networks. In *NeurIPS*, 2020.

[162] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J. Kim. Graph transformer networks. *CoRR*, abs/1911.06455, 2019. URL `http://arxiv.org/abs/1911.06455`.

[163] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna. GraphSAINT: Graph sampling based inductive learning method. In *Proc. of ICLR*, 2020.

[164] Jieyu Zhao, Tianlu Wang, Mark Yatskar, Vicente Ordonez, and Kai-Wei Chang. Men also like shopping: Reducing gender bias amplification using corpus-level constraints. In Martha Palmer, Rebecca Hwa, and Sebastian Riedel, editors, *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pages 2979–2989. Association for Computational Linguistics, 2017. doi: 10.18653/v1/d17-1323. URL `https://doi.org/10.18653/v1/d17-1323`.

[165] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[166] M. Zitnik and J. Leskovec. Predicting multicellular function through multi-layer tissue networks. *Bioinformatics*, 33(14):i190–i198, 2017.

Appendix A

Derivation of Support-Vector Machine (SVM) Algorithms

A.1. Hard-margin SVM

Let us start from the most basic and intuitive case, the linear SVM [11]. Assume that we have p data points from two classes in the form of

$$(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_p, y_p)$$

where $\mathbf{x}_i$ is some N-dimensional vector, and $y \in 1, -1$. SVM training outputs a decision function $D(\mathbf{x})$ such that $D(\mathbf{x}) > 0 \implies y = 1$ and $y = -1$. otherwise. For the linear SVM, the decision function defines a separating hyperplane in the form of $\langle \mathbf{w}, \mathbf{x} \rangle + b$, where sign of the output determines the class:

$$D(\mathbf{x}) = \text{sign}(\langle \mathbf{w}, \mathbf{x} \rangle + b) \tag{A.1.1}$$

Here, $\mathbf{w}$ is an N-dimensional vector that determines the orientation of the hyperplane, and b is a bias that represents the offset from the origin. For linearly separable training data $(\mathbf{x}_i, y_i)$, the decision function satisfies

$$\frac{y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)}{\|\mathbf{w}\|} \geq M \tag{A.1.2}$$

and the training objective is to find some $\mathbf{w}$ (which we can normalize to unit length) that maximizes the margin M, resulting in the *maximum-margin hyperplane*:

$$\begin{aligned} M^* = \max_{\mathbf{w}, \|\mathbf{w}\|=1} M \\ \text{s.t.} \quad y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq M, \quad i = 1, 2, \ldots, p \end{aligned} \tag{A.1.3}$$

The maximum-margin hyperplane separates the data points of the two classes in a way such that the distance of the hyperplane to the nearest data point from either class is maximized.

Inevitably, some data points will satisfy

$$\min_i y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) = M^* \tag{A.1.4}$$

by defining the maximum margin; these points constitute the *support vectors* where the algorithm derives its name from. Now, let us denote $\mathbf{w}_M = \frac{\mathbf{w}}{M}$ and $b_M = \frac{b}{M}$. This allows us to reframe Equation A.1.3 as:

$$\begin{aligned} M^* = &\max_{\mathbf{w}, \|\mathbf{w}\|=1} M \\ \text{s.t.} \quad &y_i(\langle \mathbf{w}_M, \mathbf{x}_i \rangle + b_M) \geq 1, \quad i = 1, 2, \ldots, p \end{aligned} \tag{A.1.5}$$

Since $\|\mathbf{w}_M\| = \frac{\|\mathbf{w}\|}{M} = \frac{1}{M}$, the size of the full margin becomes $2M = \frac{2}{\|\mathbf{w}_M\|}$ (as M denotes the distance denotes the distance from the hyperplane to only one side of the full margin). Thus, maximizing M is equivalent to minimizing the norm $\|\mathbf{w}_M\|$. We can then once again reframe our objective as the following optimization problem:

$$\begin{aligned} &\arg\min_{\mathbf{w}_M, b_M} \frac{1}{2}\|\mathbf{w}_M\|^2 \\ \text{s.t.} \quad &y_i(\langle \mathbf{w}_M, \mathbf{x}_i \rangle + b_M) \geq 1, \quad i = 1, 2, \ldots, p \end{aligned} \tag{A.1.6}$$

The resulting problem is now a convex quadratic programming (QP) problem, and can be solved directly. However, for high-dimensional spaces the solution space grows extremely large. One can instead solve for the Lagrangian dual of this problem for a more efficient solution.

Introducing Lagrange multipliers $\lambda_i \geq 0, i = 1, \ldots, p$ for the inequality constraints above gives us the primal Lagrangian function below. Now that the reframing of the problem is complete, with slight abuse of notation, let's revert $\mathbf{w}_M$ to $\mathbf{w}$, and b_M to b for the sake of clarity.

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\lambda}) = \frac{1}{2}\|\mathbf{w}\|^2 + \sum_{i=1}^{p} \lambda_i (1 - y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)) \tag{A.1.7}$$

The Lagrangian dual is then given by

$$\phi(\boldsymbol{\lambda}) = \inf_{\mathbf{w} \in \mathbb{R}^N, b \in \mathbb{R}} \mathcal{L}(\mathbf{w}, b, \boldsymbol{\lambda}) \tag{A.1.8}$$

We can obtain the dual by setting the gradient of $\mathcal{L}$ w.r.t. $\mathbf{w}$ and b to 0, which gives us

$$\mathbf{w} = \sum_{i=1}^{p} \lambda_i y_i \mathbf{x}_i \tag{A.1.9}$$

$$\sum_{i=1}^{p} \lambda_i y_i = 0 \tag{A.1.10}$$

Substituting these into the primal Lagrangian in A.1.7, we get the simpler dual Lagrangian:

$$\phi(\boldsymbol{\lambda}) = \sum_{i=1}^{p} \lambda_i - \frac{1}{2} \sum_{i=1}^{p} \sum_{j=1}^{p} \lambda_i \lambda_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \tag{A.1.11}$$

The dual Lagrangian does not depend on $\mathbf{w}$ nor b, but only on $\boldsymbol{\lambda}$. Also, the stationary point, which is a minimum for the primal Lagrangian, is a maximum for the dual. The resulting dual problem is then

$$\begin{aligned} &\arg\max_{\boldsymbol{\lambda}} \sum_{i=1}^{p} \lambda_i - \frac{1}{2} \sum_{i=1}^{p} \sum_{j=1}^{p} \lambda_i \lambda_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \\ \text{s.t.} \quad &\sum_{i=1}^{p} \lambda_i y_i = 0, && i = 1, 2, \ldots, p \\ &\lambda_i \geq 0, && i = 1, 2, \ldots, p \end{aligned} \tag{A.1.12}$$

which is also a QP problem that can be solved numerically. The solution $\hat{\boldsymbol{\lambda}}$ can then be used to obtain $\hat{w}$ corresponding to the maximum-margin hyperplane via A.1.9:

$$\hat{\mathbf{w}} = \sum_{i=1}^{p} \hat{\lambda}_i y_i \mathbf{x}_i \tag{A.1.13}$$

By the *dual feasability* and *complementary slackness* conditions in the Karush–Kuhn–Tucker (KKT) conditions [85], we have the following properties respectively:

$$\begin{aligned} \hat{\lambda}_i &\geq 0, && i = 1, 2, \ldots, p \\ \hat{\lambda}_i (1 - y_i (\langle \hat{\mathbf{w}}, \mathbf{x}_i \rangle + b)) &= 0, && i = 1, 2, \ldots, p \end{aligned} \tag{A.1.14}$$

This implies

$$\hat{\lambda}_i > 0 \implies 1 - y_i (\langle \hat{\mathbf{w}}, \mathbf{x}_i \rangle + \hat{b}) = 0 \tag{A.1.15}$$

meaning $\hat{\lambda}_i$ is positive only when $y_i (\langle \hat{\mathbf{w}}, \mathbf{x}_i \rangle + b) = 1$, i.e for the support vectors that lie on the margin by definition, as per Eqs. A.1.4 and A.1.5. This allows us to refine Eq. A.1.13 to account for only the support vectors:

$$\begin{aligned} \text{SV} &= \{ \hat{\lambda}_i > 0, \quad i \in [1, 2, \ldots, p] \} \\ \hat{\mathbf{w}} &= \sum_{i \in \text{SV}} \hat{\lambda}_i y_i \mathbf{x}_i \end{aligned} \tag{A.1.16}$$

$\hat{b}$ is also obtained from the support vectors via A.1.15; while one support vector may be sufficient, in practice the average for all support vectors are computed for stability:

$$\hat{b} = \frac{1}{|\text{SV}|} \sum_{i \in \text{SV}} (y_i - \langle \hat{\mathbf{w}}, \mathbf{x}_i \rangle) \tag{A.1.17}$$

One can classify some new data point $\mathbf{x} \in \mathbb{R}^N$ using the resulting decision function

$$\text{sign}(\langle \hat{\mathbf{w}}, \mathbf{x}_i \rangle + \hat{b}) = \text{sign}\left(\sum_{i \in \text{SV}} \hat{\lambda}_i y_i \langle \mathbf{x}_i, \mathbf{x} \rangle + \hat{b} \right) \tag{A.1.18}$$

A.2. Soft-margin SVM

The *hard-margin* SVM algorithm introduced above does have some shortcomings. To begin, it is unsuitable for linearly non-separable data as aforementioned. Furthermore, it is

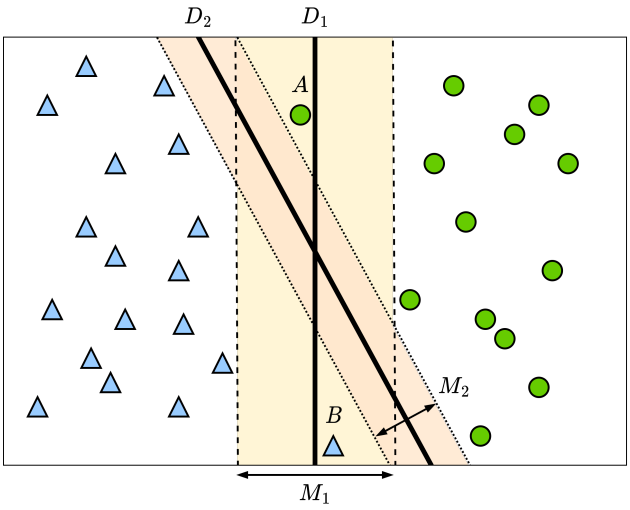


Figure A.1 – A linearly separable data with two outliers A and B that significantly reduce the separation margin from M_1 to M_2, corresponding to soft-margin classifier D_1 and hard-margin classifier D_2.

sensitive to outliers. Even in cases where the data is linearly separable, outliers may alter the separating hyperplane in a way that reduces the separation margin, as illustrated by Fig A.1. In such cases, it may be beneficial to relax the linear separability constraints for outliers in order to obtain a larger margin for the majority of data points. This brings us to the *soft-margin* SVM algorithm.

The soft-margin SVM relaxes the separability constraints by introducing slack variables $\xi_1, \ldots, \xi_p \geq 0$ to the constraints in Eq. A.1.6:

$$y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1 - \xi_i, \quad i = 1, 2, \ldots, p \tag{A.2.1}$$

These slack constraints allow for data points within the margin, or even on the other side of the decision boundary. For each such data point $\mathbf{x}_i$, ξ_i denotes the distance of the data point from the margin. Now, we want our classifier to not just maximize the margin (and hence minimize $\|\mathbf{w}\|^2$ as shown above), but also to reduce the errors by minimizing ξ_i. We therefore rewrite our optimization problem in Eq. A.1.6 as

$$\begin{aligned} & \arg\min_{\mathbf{w}} \frac{1}{2}\|\mathbf{w}\|^2 + \beta \sum_{i=1}^{p} \xi_i \\ \text{s.t.} \quad & y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1 - \xi_i, \qquad i = 1, 2, \ldots, p \\ & \xi_i \geq 0, \qquad\qquad\qquad\qquad i = 1, 2, \ldots, p \end{aligned} \tag{A.2.2}$$

where β denotes a configurable constant that is used to manage the trade-off between the two terms to optimize. Whenever we have $y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) < 1$ for some x_i (meaning this data point is within the margins), we pay a cost of $\beta\xi_i$ in the objective function. A point is

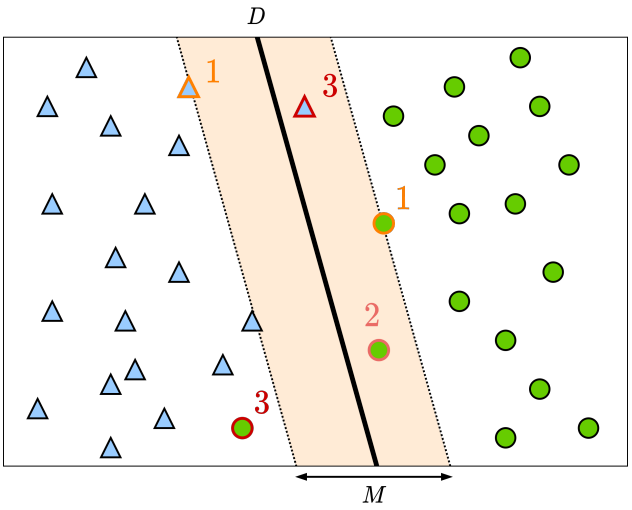


Figure A.2 – Three possible types of support vectors for non-linearly separable data: SV_1, SV_2, SV_3 as defined in Equation A.2.7.

misclassified if $\xi_i \geq 1$. A large β keeps classification errors to a minimum but may result in a reduced margin, while a small β permits more misclassified examples for a larger margin on the remaining data points.

The dual Lagrangian problem formulation of the soft-margin linear SVM is identical to its hard-margin counterpart in A.1.12, except the final set of constraints on $\boldsymbol{\lambda}$ which are replaced by

$$\beta \geq \lambda_i \geq 0, \quad i = 1, 2, \ldots, p \tag{A.2.3}$$

One again solves the convex QP problem to obtain $\hat{\boldsymbol{\lambda}}$, and derive $\hat{\mathbf{w}}$ similarly:

$$\hat{\mathbf{w}} = \sum_{i=1}^{p} \hat{\lambda}_i y_i \mathbf{x}_i \tag{A.2.4}$$

By the KKT conditions we have:

$$\begin{aligned} \hat{\lambda}_i \left(1 - \hat{\xi}_i - y_i \left(\langle \hat{\mathbf{w}}, \mathbf{x}_i \rangle + \hat{b}\right)\right) &= 0 \\ \left(C - \hat{\lambda}_i\right) \hat{\xi}_i &= 0 \end{aligned} \tag{A.2.5}$$

Considering these equations along with A.2.3 gives:

$$\beta \geq \lambda_i \geq 0 \implies 1 - y_i \left(\langle \hat{\mathbf{w}}, \mathbf{x}_i \rangle + \hat{b}\right) = 0 \tag{A.2.6}$$

This results in three types of support vectors inferred from $\hat{\lambda}_i > 0$, as illustrated in Fig. A.2:

$$\begin{aligned} \mathrm{SV}_1 &= \{\beta > \hat{\lambda}_i > 0, \qquad i \in [1, 2, \ldots, p]\} \\ \mathrm{SV}_2 &= \{\hat{\lambda}_i = \beta, \hat{\xi} < 1, \qquad i \in [1, 2, \ldots, p]\} \\ \mathrm{SV}_3 &= \{\hat{\lambda}_i = \beta, \hat{\xi} \geq 1, \qquad i \in [1, 2, \ldots, p]\} \end{aligned} \tag{A.2.7}$$

SV_1: Margin support vectors, correctly classified

SV_2: Within margin, correctly classified

SV_2: Opposite side of decision boundary, misclassified

Then, $\hat{\mathbf{w}}$ can be rewritten using the union of the support vectors above, as they all have $\hat{\lambda}_i > 0$. $\hat{b}$, however, would be computed only using the margin support vectors SV_1:

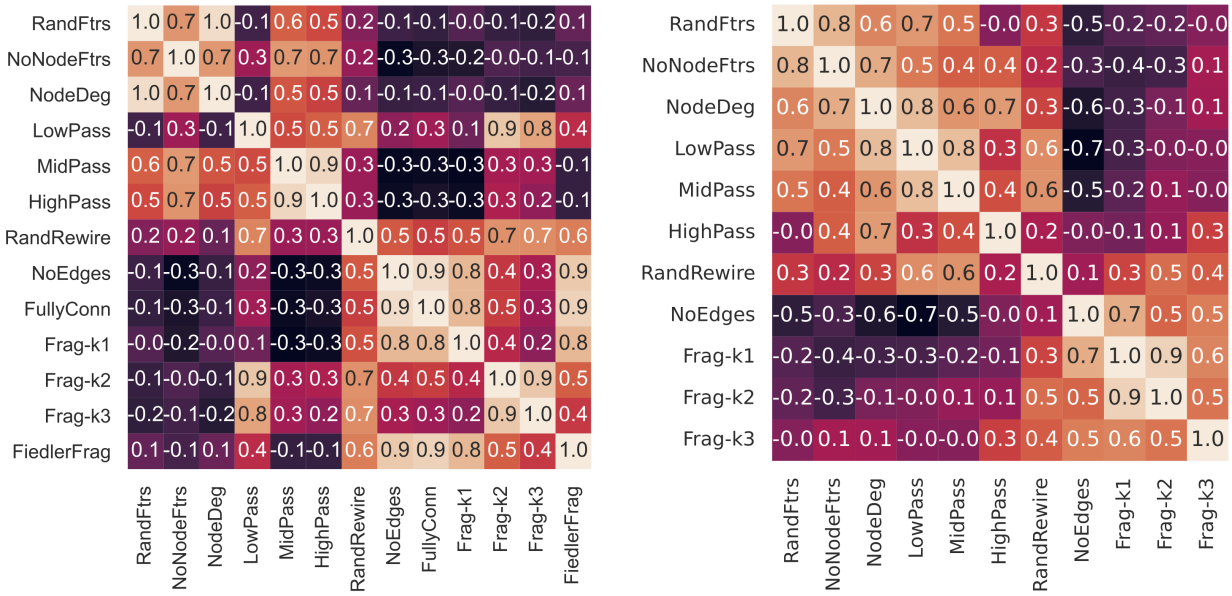
$$\hat{\mathbf{w}} = \sum_{i \epsilon SV_1 \cup SV_2 \cup SV_3} \hat{\lambda}_i y_i \mathbf{x}_i \tag{A.2.8}$$

$$\hat{b} = \frac{1}{|SV_1|} \sum_{i \epsilon SV_1} \left(y_i - \langle \hat{\mathbf{w}}, \mathbf{x}_i \rangle \right) \tag{A.2.9}$$

Appendix B

Additional Taxonomy Visualizations

B.1. Correlations of Perturbations



(a) Inductive benchmarks **(b)** Transductive benchmarks

Figure B.1 – Pearson correlation coefficients of the log2 performance fold change between different perturbations (w.r.t. a GCN model).

We compute the Pearson correlation between all pairs of perturbations based on the log2 performance fold change. The results in Figure B.1 indicate that many perturbations correlate with each other to some extend. For both transductive and inductive benchmarks, the perturbations roughly cluster into two groups, separating node feature perturbations (see Section 4.2.1) and graph structure perturbations (see Section 4.2.2). In particular, perturbations that replace the original node features with other less informative features, including *RandFtrs*, *NoNodeFtrs*, and *NodeDeg*, highly correlate with one another (Pearson

$r \geq 0.6$). Similarly, perturbations that severely break the graphs apart, including *NoEdges*, *Frag-k1*, and *FiedlerFrag*, are highly correlated (Pearson $r \geq 0.8$).

Appendix C

Supplementary Studies

C.1. Distribution of Classical Graph Properties in Benchmarking Datasets

In this work we use *perturbation sensitivity profiles* derived from a GNN's prediction performance in order to gauge *how task-related information* is encoded in the graph datasets. In this section we explore an alternative approach. We analyze classical graph properties in multiple datasets and their classes to investigate whether we can establish a meaningful taxonomy without a dependence on a particular GNN method, while using well-established graph properties.

A static analysis of the graph properties alone is insufficient without taking into account the prediction task as well. The graph domain that a dataset X is sampled from (e.g., drug-like molecules, proteins, ego networks, citation networks) may exhibit varying range of properties (e.g., density, node degree distribution, local/global clustering coefficients, number of triangles, graph diameter, girth, maximum clique, etc.), however these do not take into account node features in attributed graphs, and could be irrelevant to the prediction task Y. Therefore, we look at the difference in graph properties compared among the individual classes of Y.

Particularly, we look at all 9 inductive binary-classification datasets from our dataset selection (Table 4.1). Within each class (the negative and positive label) of these 9 datasets we computed the average value of 9 graph properties computed by the NetworkX package [56]. The results are presented in Table C.1 and Figure C.1. Primarily, the computed graph properties vary more between datasets than between classes. The marginal graph properties of the positive and negative class are very similar to each other, especially for the SYNTHETICnew dataset. The largest difference between the classes appears to be the average size of the graphs, which is captured by the average number of nodes and edges. Therefore we

Table C.1 – Classical graph properties among positive and negative classes of 9 graph-classification datasets. The difference between datasets dominates within-dataset differences between classes.

	Num. nodes	Num. edges	Density	Connectivity	Diameter	Approx. max clique	Centrality	Cluster. coeff.	Num. triangles
IMDB-BINARY (class=0)	20.11	96.78	0.559	3.828	1.838	10.30	0.559	0.943	307.73
IMDB-BINARY (class=1)	19.43	96.29	0.482	3.388	1.884	10.01	0.482	0.951	476.25
REDDIT-BINARY (class=0)	641.25	735.95	0.012	0.556	5.646	3.22	0.012	0.054	35.96
REDDIT-BINARY (class=1)	218.00	259.56	0.032	0.423	3.778	2.95	0.032	0.041	13.71
D&D (class=0)	341.88	870.23	0.019	1.110	20.843	4.95	0.019	0.479	617.07
D&D (class=1)	183.72	449.43	0.040	1.140	17.460	4.79	0.040	0.480	302.55
PROTEINS (class=0)	50.00	94.06	0.142	1.196	13.837	3.85	0.142	0.473	34.30
PROTEINS (class=1)	22.94	41.52	0.315	1.420	7.278	3.80	0.315	0.575	17.24
NCI1 (class=0)	25.65	27.65	0.100	0.924	11.265	2.02	0.100	0.002	0.03
NCI1 (class=1)	34.07	36.94	0.078	0.796	11.917	2.05	0.078	0.004	0.07
NCI109 (class=0)	25.61	27.61	0.100	0.913	11.061	2.02	0.100	0.002	0.02
NCI109 (class=1)	33.69	36.59	0.079	0.794	11.644	2.05	0.079	0.004	0.07
MUTAG (class=0)	13.94	14.62	0.169	1.000	7.016	2.00	0.169	0.000	0.00
MUTAG (class=1)	19.94	22.40	0.123	1.000	8.824	2.00	0.123	0.000	0.00
SYNTHETICnew (class=0)	100.00	196.42	0.040	0.993	7.333	3.00	0.040	0.024	5.39
SYNTHETICnew (class=1)	100.00	196.08	0.040	0.993	7.213	3.00	0.040	0.022	4.54
ogbg-molhiv (class=0)	25.20	27.13	0.104	0.931	11.016	2.02	0.104	0.002	0.03
ogbg-molhiv (class=1)	34.18	36.69	0.084	0.824	12.183	2.01	0.084	0.001	0.01

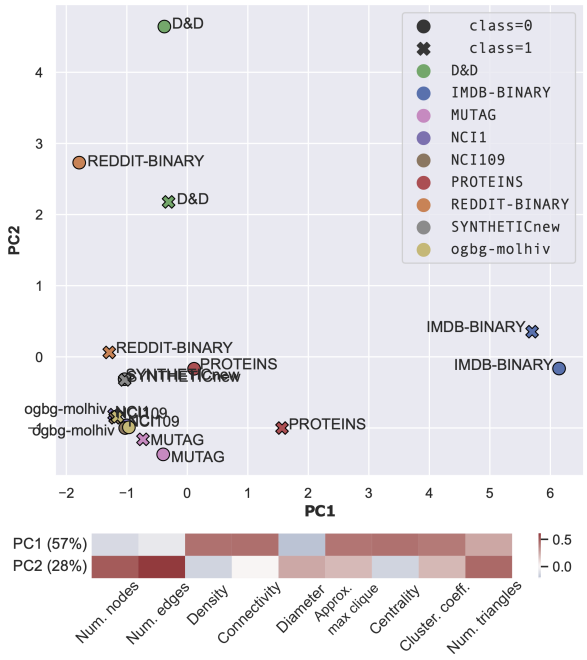


Figure C.1 – PCA plot of 9 binary graph-level classification datasets represented by their per-class graph properties. In the bottom, the loadings of the first two principal components are shown.

argue that basing a taxonomy on dataset or class-level marginal graph properties is grossly insufficient as it completely fails to capture the nature of the prediction task.

Alternatively, one could conduct a correlation analysis between classical graph properties (averaged per class) and the outcome Y. However, that would again only take into account

the marginal properties, assume linear relationship (as correlation captures only a linear relationship), and would rely on a fixed set of computable graph properties. These appear to be fundamental limitations compared to the perturbation analysis presented in the main text, that would result in a grossly skewed taxonomy.

C.2. Impact of random initialization on *Frag-k* perturbations

Our *Frag-k* perturbation is potentially sensitive to the random initializations of the initial seed nodes used in the fragmentation procedure. To measure this sensitivity of *Frag-k* perturbations to node initializations, we computed the variance of AUROC results across ten experiments with different random seeds for both GCN and GIN models. Here, we analysed five datasets, the performance on which was significantly altered by *Frag-k* in the original analysis, namely, CLUSTER, PATTERN, PPI, Synthie, and SYNTHETICnew. The variances are within 5%, with the only exception being `SYNTHETICnew`. We hypothesize that this is due to the randomness of the constructions of the `SYNTHETICnew` dataset. Thus, overall, the *Frag-k* approach is sufficiently stable for datasets whose constructions involve little randomness.

Table C.1 – Variances of AUROC across ten different random seeds for *Frag-k* for GCN.

Dataset	Perturbation	AUROC Avg.	AUROC Std.	AUC Std./Avg. (%)
CLUSTER	*Frag-k1*	0.637	0.001	0.165
CLUSTER	*Frag-k2*	0.913	0.000	0.039
CLUSTER	*Frag-k3*	0.913	0.000	0.037
PATTERN	*Frag-k1*	0.769	0.001	0.095
PATTERN	*Frag-k2*	0.933	0.000	0.016
PATTERN	*Frag-k3*	0.933	0.000	0.021
PPI	*Frag-k1*	0.620	0.003	0.529
PPI	*Frag-k2*	0.647	0.012	1.807
PPI	*Frag-k3*	0.720	0.011	1.519
SYNTHETICnew	*Frag-k1*	0.704	0.126	17.908
SYNTHETICnew	*Frag-k2*	0.533	0.078	14.701
SYNTHETICnew	*Frag-k3*	0.715	0.089	12.492
Synthie	*Frag-k1*	0.962	0.015	1.581
Synthie	*Frag-k2*	0.870	0.029	3.334
Synthie	*Frag-k3*	0.876	0.036	4.164

Table C.2 – Variances of AUROC across ten different random seeds for *Frag-k* for GIN.

Dataset	Perturbation	AUROC Avg.	AUROC Std.	AUC Std./Avg. (%)
CLUSTER	*Frag-k1*	0.643	0.001	0.162
CLUSTER	*Frag-k2*	0.910	0.001	0.101
CLUSTER	*Frag-k3*	0.910	0.001	0.130
PATTERN	*Frag-k1*	0.780	0.001	0.091
PATTERN	*Frag-k2*	0.934	0.000	0.013
PATTERN	*Frag-k3*	0.934	0.000	0.019
PPI	*Frag-k1*	0.617	0.002	0.376
PPI	*Frag-k2*	0.644	0.009	1.476
PPI	*Frag-k3*	0.704	0.013	1.843
SYNTHETICnew	*Frag-k1*	0.708	0.081	11.407
SYNTHETICnew	*Frag-k2*	0.532	0.071	13.276
SYNTHETICnew	*Frag-k3*	0.757	0.064	8.411
Synthie	*Frag-k1*	0.985	0.008	0.810
Synthie	*Frag-k2*	0.945	0.011	1.213
Synthie	*Frag-k3*	0.920	0.025	2.677

Appendix D

Les différentes parties et leur ordre d'apparition

J'ajoute ici les différentes parties d'un mémoire ou d'une thèse ainsi que leur ordre d'apparition tel que décrit dans le guide de présentation des mémoires et des thèses de la Faculté des études supérieures. Pour plus d'information, consultez le guide sur le site web de la facutlé (www.fes.umontreal.ca).

Ordre des éléments constitutifs du mémoire ou de la thèse		
1.	La page de titre	obligatoire
2.	La page d'identification des membres du jury	obligatoire
3.	Le résumé en français et les mots clés français	obligatoires
4.	Le résumé en anglais et les mots clés anglais	obligatoires
5.	Le résumé dans une autre langue que l'anglais ou le français (si le document est écrit dans une autre langue que l'anglais ou le français)	obligatoire
6.	Le résumé de vulgarisation	facultatif
7.	La table des matières, la liste des tableaux, la liste des figures ou autre	obligatoires
8.	La liste des sigles et des abréviations	obligatoire
9.	La dédicace	facultative
10.	Les remerciements	facultatifs
11.	L'avant-propos	facultatif
12.	Le corps de l'ouvrage	obligatoire
13.	Les index	facultatif
14.	Les références bibliographiques	obligatoires
15.	Les annexes	facultatifs
16.	Les documents spéciaux	facultatifs