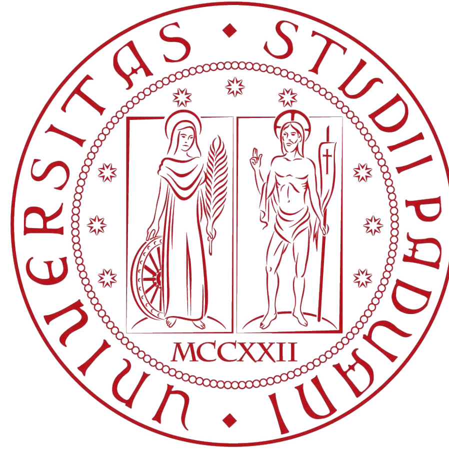


**Università degli Studi di Padova**

Department of Mathematics "Tullio Levi-Civita"

Master's Degree in Computer Science


# Hierarchical Line Graph Neural Network: A Study on Alternative Representations of Graph-Structured Data

*Master's Thesis*


*Supervisor*

Prof. Alessandro Sperduti

*Master Candidate*

Solmaz Mohammadi

*Matricola* 2041589



Academic Year 2023-2024

Solmaz Mohammadi: *Hierarchical Line Graph Neural Network: A Study on Alternative Representations of Graph-Structured Data,* Master's Thesis, © July 2024.

“Regardless of the staggering dimensions of the world about us, the density of our ignorance, the risks of catastrophes to come, and our individual weakness within the immense collectivity, the fact remains that we are absolutely free today if we choose to will our existence in its finiteness, a finiteness which is open on the infinite. And in fact, any man who has known real loves, real revolts, real desires, and real will knows quite well that he has no need of any outside guarantee to be sure of his goals; their certitude comes from his own drive.”

— Simone de Beauvoir

# Abstract

This thesis addresses the *over-smoothing* issue common in deep graph neural networks (GNNs), a topic of considerable interest over the past decade. Despite the expansion of GNN models aiming to capture richer representations of graph-structured data, the deep variants often encounter the issue of over-smoothing. This phenomenon inhibits their performance potential, limiting their ability to extract and preserve meaningful information from graph inputs. To tackle this challenge, we propose the Hierarchical Line Graph Neural Network (**HLGNN**), a novel framework that leverages the concept of iterated line graphs from graph theory. While line graph transformation—a method for converting a graph into its line graph representation to capture higher-order connectivity patterns—has gained attention in graph theory, its application within the realm of graph neural networks remains under-explored. Inspired by the principles of hierarchical and higher-order GNNs, we introduce a message-passing mechanism within HLGNN that facilitates the flow of information across both intra-graph and inter-graph levels. This hierarchical approach enables HLGNN to address the over-smoothing problem effectively while enhancing the model's capacity to capture complex graph structures. The versatility of HLGNN extends to various graph-related tasks, including node classification, graph classification, and community detection. Throughout this thesis, we conduct theoretical research along with practical experiments to evaluate the efficacy of the proposed framework, focusing particularly on node classification tasks.

*There's a crack in everything, that's how the light gets in.*

— Leonard Cohen

# Acknowledgments

*First and foremost, I would like to express my gratitude to Prof. Alessandro Sperduti, my thesis advisor, for the help and support provided during the writing of this work.*

*I am also deeply thankful to my parents, whose unwavering trust and love have been a source of comfort throughout this journey, despite the physical distance between us.*

*As I reflect on this transformative journey, I am overwhelmed with gratitude for the people who listened, understood, and cared for me. Alongside my academic pursuits for a Master's degree, I faced the challenge of adapting to a completely new life, and I feel incredibly fortunate to have had such an enriching experience.*

*I was lucky to embark on this journey with friends from home. I am eternally grateful for my study and travel companion, Sepide, who has stood by me through thick and thin. She always made me feel valued by capturing moments of my life, and I am thankful to have her in my life.*

*I have learned that meaningful connections can be formed in any context if one is open to them. I had the opportunity to strengthen my friendships with Shayan, Khashayar, and Amirhossein. Despite being miles apart, we shared common struggles, and they offered me their empathy, support, and a unique, inspiring outlook on life.*

*Throughout this journey, I have met many wonderful people. Among them, Leonardo has had a profound impact on my life. His energetic, inquisitive, and playful nature opened my mind to new perspectives and taught me the value of embracing silliness.*

*Lastly, my dear partner, Davide, has been a guiding light during my darkest days, filling my life with love and happiness. His dedication and ambition have inspired me to pursue my goals and strive to become a better person. I am forever grateful for his companionship and eagerly look forward to the new chapters of our life together.*

*Padova, July 2024* Solmaz Mohammadi

# Contents

Introduction 1

1 Theoretical Background 3
1.1 Introduction . . . 3
1.2 Table of Notations . . . 3
1.3 The Graph Neural Network Model . . . 4
1.3.1 Permutation Invariance . . . 5
1.3.2 Neural Message-passing . . . 5
1.3.3 The GNN Module . . . 6
1.3.4 Over-smoothing Problem . . . 6
1.3.5 GNN Task Types . . . 6
1.4 Line graphs . . . 7
1.4.1 Definition . . . 7
1.4.2 Basic Properties . . . 8
1.4.3 Transformation Complexity . . . 9
1.4.4 Spectral Properties . . . 9
1.4.5 Iterated Line Graphs . . . 10
1.4.6 Line Graphs in Graph Neural Networks . . . 11

2 Literature Review 12
2.1 Introduction . . . 12
2.2 Propagation and Sampling Modules . . . 13
2.2.1 GCN (Graph Convolutional Network) . . . 13
2.2.2 GraphSAGE . . . 14
2.2.3 GAT (Graph Attention Network) . . . 14
2.2.4 GIN (Graph Isomorphic Network) . . . 14
2.3 Hierarchical Modules . . . 15
2.3.1 Diff-Pool . . . 15
2.3.2 Att-Pool . . . 16
2.3.3 Graph Transformers . . . 16
2.3.4 HGNet . . . 16
2.3.5 HGNN . . . 17
2.3.6 HC-GNN . . . 17

3 Methodology 18
3.1 Introduction . . . 18
3.2 Hierarchy Construction . . . 19

3.3 Message-passing Mechanism . . . 20
3.3.1 Within-level Aggregation . . . 20
3.3.2 Bottom-up Fusion . . . 20
3.3.3 Top-down Fusion . . . 21
3.3.4 Update Function . . . 21
3.4 Theoretical Analysis . . . 21
3.4.1 Permutation Invariance . . . 22
3.4.2 Computational Complexity . . . 22

**4 Experiments** **24**
4.1 Introduction . . . 24
4.2 Task Setup . . . 24
4.2.1 Fully-supervised vs. Semi-supervised . . . 24
4.2.2 Loss Function . . . 25
4.3 Datasets . . . 25
4.4 Implementation . . . 26
4.4.1 GraphConv . . . 27
4.4.2 Fusions . . . 28
4.4.3 Update Function . . . 29
4.4.4 ReLU Activation . . . 29
4.4.5 Normalization . . . 29
4.4.6 Dropout . . . 30
4.4.7 Weight Decay . . . 30
4.5 Model Selection . . . 30
4.5.1 Hyper-parameters . . . 30
4.5.2 Training Settings . . . 31

**5 Results** **32**
5.1 Introduction . . . 32
5.2 Baseline Comparison . . . 32
5.3 Training Performance . . . 33
5.4 Model Evaluation . . . 36
5.4.1 Confusion Matrices . . . 36
5.4.2 ROC Curves . . . 37
5.4.3 Node Embedding Visualizations. . . . 39
5.5 Discussion . . . 41
5.5.1 Addressing Over-smoothing . . . 41
5.5.2 Advantages of Line Graph Transformation . . . 41
5.5.3 Computational Complexity . . . 41
5.5.4 Interpretability and Robustness . . . 41
5.5.5 Versatility and Future Work . . . 42

**Conclusion** **43**

**A Algorithms** **45**
A.1 HLGNN Forward pass . . . 45
A.2 HLGNN Fusion Operations . . . 45
A.3 HLGNN Training Algorithm . . . 46

# List of Figures

1.1 Example of a graph and its line graph. . . . . . . . . . . . . . . . . . . . 8
1.2 Example of a graph and its iterated line graphs. . . . . . . . . . . . . 11

3.1 The construction of the bipartite graphs between the graphs in the hierarchy using the incidence matrix. On the left, we have a graph $G$ and its line graph $L(G)$, and on the right, another iteration of line graphs, $L(G)$ and $L(L(G))$. A bipartite graph is then formed between the graphs throughout the hierarchy using the incidence matrices. The blue dashed lines represent the edges of the bipartite graph. . . 21
3.2 Illustration of the message-passing mechanism for node $u$ in Graph $G_t$ within the hierarchy. The orange dashed lines depict the bottom-up fusion operation, aggregating information from $G_{t-1}$, while the blue dashed lines represent the top-down fusion operation, aggregating information from corresponding nodes in $G_{t+1}$. These fusion operations, combined with within-level aggregations in $G_t$, are integrated using the UPDATE function. . . . . . . . . . . . . . . . . . . . . . . . 22

4.1 A scheme of the implementation of the HLGNN layer. The diagram highlights the important parts of the model construction, including inter-level fusions and convolution operators. . . . . . . . . . . . . . . 26
4.2 An example of the HLGNN architecture, with a hierarchical level of 3, and two hidden (HLGNN) layers. The generated line graphs' embeddings are initialized as zero, and the hidden layers are applied to the graphs independently. The output embedding is constructed by feeding the concatenation of the final embeddings into a fully connected layer. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.3 A scheme of the message-passing operations leveraging the incidence matrix $B$. On the left, the top-down fusion is shown, transmitting messages from the next graph. On the right, the bottom-up fusion is illustrated, aggregating messages from the previous graph. Both operations utilize the incidence matrix and subsequently apply a convolution layer, *GraphConv*, to generate the new embeddings. . . . 28

5.1 Accuracy and loss curves for the training, validation, and test sets on the three datasets. For the Cora dataset, (a) the accuracy plot shows rapid convergence and stable performance across all sets, while (b) the loss plot demonstrates a steady decrease in loss, indicating effective learning. For the Pubmed dataset, (c) displays consistent improvement in accuracy, and the loss plot (d) shows a continuous reduction in loss, reflecting the model's robust training process and generalization capability. For the Citeseer dataset, both (e) and (f) illustrate the model's convergence and stabilization, with slight fluctuations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.2 Confusion matrices for the three datasets, (a) Cora, (b) Pubmed, and (c) Citeseer. The diagonal elements represent the number of correct predictions for each class. The darker colors along the diagonal indicate high number of correct predictions. Off-diagonal elements represent misclassifications, the intensity of colors can show which classes are commonly confused with each other. . . . . . . . . . . . . 36
5.3 The ROC curves depicted for the three datasets. In (a), (c), and (e) individual barbarized ROC curves are presented for each class, while (b), (d), and (f) showcase the Micro-average ROC curve. The area under the ROC curve indicates the model's ability to discriminate between positive and negative classes, and a curve closer to the top-left corner indicates a better performing model. Both visualizations highlight a notable sensitivity rate, indicating that the model effectively distinguishes between positive and negative instances across various classes, contributing to its robust performance in classification tasks. 38
5.4 Visualization of Node Embeddings across three datasets: Cora, Pubmed, and Citeseer. In the left column, (a), (c), and (e) display the initial (pre-training) node features. Correspondingly, (b), (d), and (f) on the right column represent the output embeddings generated by the trained model. Class distinctions are illustrated through varied coloring in each plot. The closeness of the nodes suggest they have similar properties while the separation between the groups of clusters indicate dissimilarities. . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

# List of Tables

1.1 A table of notations containing the key symbols used in this thesis and their description. . . . 4

4.1 Hyper-parameters used for training models on the Cora, Pubmed, and Citeseer datasets. . . . 31

5.1 Accuracy comparison of the proposed model (HLGNN) against three baseline models (GCN, GraphConv, GAT) on a fully-supervised node classification task. ask. Model selection is based on results from the validation set. The highest accuracy is highlighted in bold for each dataset. . . . 32

5.2 Hyper-parameters used for training models on the Cora, Pubmed, and Citeseer datasets. . . . 33

# Introduction

Graph Neural Networks (GNNs) have become a powerful tool for modeling complex relational data structures. In the last decades, the demand for effective graph representation learning methods has grown significantly. They are widely used in various areas such as social networks [19], recommendation systems [26, 31], and biological [9] and chemical networks [29, 21]. Simple graph kernels and heuristics cannot capture the complex relations present in the graph and the traditional neural network architectures like Convolutional Neural Networks (CNNs) designed for visual data and Recurrent Neural Networks (RNNs) designed to learn from sequences, cannot be applied to graph-structured data due to its non-Euclidean nature. Therefore, GNN models that introduce a learned representation of the data while preserving the graph's structural information, are dominantly more effective that other approaches.

Despite their effectiveness, existing GNN models have some limitations. One of the challenges that GNN models, especially flat ones face is the over-smoothing problem. Over-smoothing often happens in deep GNN models, where after several layers of message-passing, the node representations become similar and lose their initial diversity, resulting in a low performance model. Another problem often present in GNN models is their inability to capture long-range interactions in the graph. Often-times there are dependencies between nodes that are not directly connected. Capturing long-range interactions are essential to understand the higher-order patterns in the graph. Traditional GNN architectures like Graph Convolutional Networks GCNs [16], operate on local information aggregation where each node aggregates information only from its immediate neighbors. Thus, this limits a model's ability to capture long-range dependencies effectively.

Hierarchical GNNs have shown to be more resilient toward over-smoothing problem since they tend to capture higher-order representations of the graph [23, 20, 32]. Moreover, line graphs have always been an interesting topic in graph theory. Informally, a line graph of a graph is a graph in which the edges of the original graph are nodes in the transformed graph [2]. Our approach, Hierarchical Line Graph Neural Network (HLGNN), integrates iterations of line graphs to construct a hierarchy of graphs to learn. These factors have motivated us to construct a hierarchical model that can widen the receptive field through alternative representations of the graph and also transmit messages between sparsely connected components in the graph.

In the following chapters, we provide a comprehensive theoretical background on GNN models and line graphs. We then review some existing GNN models,

highlighting their contribution and acknowledging their drawbacks. Following that, we introduce our novel GNN architecture along with detailed explanation of its components. The subsequent chapters explain our experimental design and provide analysis of the results to demonstrate the effectiveness of our approach compared to other baseline methods. Finally, we will discuss the implications of our findings and outline potential future works in the graph representation field.

# Chapter 1

# Theoretical Background

## 1.1 Introduction

It is beneficial to present a theoretical background of the fundamentals discussed in this thesis, to present a comprehensive overview in Chapter 2, and explain the proposed methodology in Chapter 3 thoroughly. Therefore, this chapter covers a brief review of graph concepts, followed by the basic definition of the graph neural network model and the message-passing scheme which can be considered the standard way of information aggregation. The second section of this chapter is dedicated to the definition of the line graph transformation, its important properties, and its relevance to graph neural network model. To demonstrate this relevance, we mention some earlier works done to integrate the line graphs into a GNN module.

## 1.2 Table of Notations

In this section, we provide a table of notations that lists the key symbols and their descriptions used throughout this thesis. This table serves as a reference to help the reader understand the terminology and symbols consistently.

| **Symbol** | **Description** |
|---|---|
| $G$ | A graph |
| $L(G)$ | Line graph of $G$ |
| $\mathcal{V}$ | Set of vertices in graph |
| $\mathcal{E}$ | Set of edges in graph |
| $u$ | A vertex (node) in graph |
| $e$ | An edge in graph |
| $\deg(u)$ | Degree of vertex $v$ |
| $A$ | Adjacency matrix of the graph |
| $B$ | Incidence matrix of the graph |
| $D$ | Degree matrix of the graph |
| $\mathcal{N}(u)$ | The set of neighbors of vertex $u$ |
| $\mathcal{I}(u)$ | The set of incident edges to vertex $u$ |
| $K_n$ | Complete graph on $n$ vertices |
| $C_n$ | Cycle graph on $n$ vertices |
| $P_n$ | Path graph on $n$ vertices |
| $G_t$ | $t$-th graph in the hierarchy |
| $\mathcal{S}$ | Set of hierarchical line graphs |
| $x_u$ | Input feature vector of vertex $u$ in the graph |
| $X$ | Input feature matrix of the graph |
| $d$ | Feature dimension |
| $h_u^{(k)}$ | Feature vector of vertex $u$ at the $k$-th layer |
| $H^{(k)}$ | Matrix of feature vectors of all vertices at the $k$-th layer |
| $W^{(k)}$ | Weight matrix at the $k$-th layer |
| $b^{(k)}$ | Bias term at the $k$-th layer |
| $\sigma$ | Activation function |

**Table 1.1:** A table of notations containing the key symbols used in this thesis and their description.

## 1.3 The Graph Neural Network Model

To define a GNN model, we first need to understand graph data-structure and its limitations. Graph-structured data contains *nodes* that are connected to each other and their relations are represented as *edges*. These relations can be directed, undirected, heterogeneous, etc. and form various types of graphs [13]. The contextual information of the graph-structured dataset is dependent on both the node features, and structural properties present in the graph. Ignoring the relations between nodes and assuming the nodes independent, causes a loss of structural information, and results in a less expressive, interpretable model. Therefore, it is important to integrate the structure of graph into our model. We first review the permutation invariance concept and discuss how it is a feasibility factor in GNN models. We then define the message-passing scheme for GNNs and explain how it integrates both feature-based and structural information utilizing local feature extraction. Lastly, we discuss the different tasks that GNNs can be applied to.

### 1.3.1 Permutation Invariance

A GNN model should be *permutation invariant* or *permutation equavariant* in order to be feasible. That is, the function $f$ should not depend on the arbitrary ordering of the rows/columns in the adjacency matrix.

**Definition 1.3.1.** Any function $f$ that takes an adjacency matrix A as input should satisfy one of the two following properties:

1. $f(PAP^T) = f(A)$ (Permutation Invariance)
2. $f(PAP^T) = Pf(A)$ (Permutation Equavariance)

where $P$ is a permutation matrix.

In other words, we cannot feed the adjacency matrix of a graph into a neural network to generate an entire graph embedding because this approach depends on the arbitrary ordering of the node in the adjacency matrix.

### 1.3.2 Neural Message-passing

To understand how a GNN works, we first need to define its crucial property, *neural message-passing.* Message-passing is a key defining feature in all GNN models [10]. In general, all GNN models operate based on a message-passing scheme in which the nodes exchange vector messages in their neighborhood. The idea is to transmit information between nodes that are locally connected. This locality usually expands based on the number of the layers of the GNN model. Therefore, a GNN with $K$ layers, aggregates messages from each node's $K$-hop neighborhood. There are various versions of message-passing depending on the approach and the application, but almost all of them can fit into the following generalization.
The book *Graph Representation Learning*, a comprehensive introductory study on GNN models, defines the message-passing operation by two arbitrary different functions, UPDATE and AGGREGATE [10]. In each message-passing iteration, a hidden embedding for each node in the graph is generated by combining *(1) the node's previous embedding* and *(2) the message aggregated from the node's neighborhood.* In other words, first a *message* is aggregated from a node's neighborhood by AGGREGATE function, and then it is combined with node's previous embedding through UPDATE function. Formally, the aggregated message for node $u$ in the $k^{th}$ layer can be written as

$$\mu_{\mathcal{N}(u)}^{(k)} = \text{AGGREGATE}^{(k)}(\{h_v^{(k-1)}, \forall v \in \mathcal{N}(u)\}) \tag{1.1}$$

Finally, the $k^{th}$ embedding of node $u$ is obtained by combining $h_u^{(k-1)}$ and the aggregated message $\mu_{\mathcal{N}(u)}^{(k)}$:

$$h_u^{(k)} = \text{UPDATE}\left(h_u^{(k-1)}, \mu_{\mathcal{N}(u)}^{(k)}\right) \tag{1.2}$$

The choice of AGGREGATE and UPDATE functions can vary depending on the task. It is notable that regardless of the type of the operations, the message $\mu_{\mathcal{N}(u)}^{(k)}$ is permutation invariant by definition since it is computed from a *set* of $u$'s neighbors.

We can stack these layers of message-passing embedding to form the hidden layers of a neural network. The first hidden embedding is usually initialized to each node's own feature vector, $h_u^{(0)} = x_u$. After K iterations of message-passing, we can set the output embedding $z_u = h_u^{(K)}$.

### 1.3.3 The GNN Module

There are various methods and theoretical approaches for implementing a GNN model. We refer to [24] as the foundational GNN, representing the most general form of the GNN module. Formally, a graph-structured dataset can be represented as a graph $G(|\mathcal{V}|, |\mathcal{E}|)$ along with the node embedding feature matrix $X \in \mathbb{R}^{|\mathcal{V}| \times d}$ where $d$ is the feature dimension. The basic GNN model proposed in [24] is described as follows:

$$h_u^{(k)} = \sigma \left( W_{self}^{(k)} h_u^{(k-1)} + W_{neigh}^{(k)} \sum_{v \in \mathcal{N}(u)} h_v^{(k-1)} + b^{(k)} \right) \tag{1.3}$$

where $\sigma$ is a non-linear activation function. $W_{self}^{(k)}, W_{neigh}^{(k)} \in \mathbb{R}^{d^{(k)} \times d^{(k-1)}}$ are trainable weight matrices, and $b^{(k)} \in \mathbb{R}^{d^{(k)}}$ is the bias term.

### 1.3.4 Over-smoothing Problem

After every iteration $k$, each node in the graph has aggregated information from its $k$-hop neighborhood. The intuition behind message-passing is straightforward: since nodes are not entirely independent, it benefits the model when nodes in a graph update their embeddings based on their neighborhood information. In essence, nodes become *aware* of their context. By context, we mean the two types of information that nodes exchange: structural and feature-based.

Structural information refers to the structural properties of the nodes within the graph. For instance, each node can acquire information regarding the degrees of the nodes in its neighborhood. Feature-based information, on the other hand, involves aggregating feature vectors from the neighborhood. While exchanging these pieces of information can enhance the model's performance, over multiple iterations, nodes in the graph tend to lose their diverse properties. This decrease in diversity diminishes the model's expressiveness and results in *over-smoothing*.

The issue of over-smoothing often arises in GNNs due to the way message-passing is utilized. Over-smoothing is particularly problematic in deep GNNs, where after several layers of message-passing, node representations become similar and lose their initial diversity, leading to poor model performance. As stated in [10], "Over-smoothing is problematic because it makes it impossible to build deeper GNN models—which leverage longer-term dependencies in the graph—since these deep GNN models tend to just generate over-smoothed embeddings."

### 1.3.5 GNN Task Types

Generally, the majority of GNN tasks can fit into one of these categories:

1. **Node classification**. The nodes of a graph are classified based on their features and structural information. Predicting the category of a user in a social network is an example of node classification.

2. **Graph classification**. Sometimes, we need to classify an entire graph based on its overall properties. Graph classification GNNs are used in studies on molecular structures, biological graphs, etc.

3. **Link prediction**. GNNs can predict the likelihood of a possible edge between nodes in the graph. They are useful for constructing recommender systems for social media platforms or advertisement strategies.

4. **Community detection** or graph clustering refers to the process of identifying underlying structures in the graph. These "clusters" represent subsets of nodes that share similar characteristics in the graph. Hierarchical methods are often utilized for community detection tasks as they can represent high-level connectivity patterns.

As the field of GNNs expands, their applications also become broader. Graph generation, graph regression, and graph representation learning are examples of other tasks that GNNs are utilized for.

## 1.4 Line graphs

In this section, we provide the formal definition of line graphs and some of their useful properties, and we discuss how message-passing in the context of GNNs can be done in (a series of) line graphs and why it can be beneficial to do so.

### 1.4.1 Definition

The line graph transformation can be considered one of the most interesting transformations in graph theory. “The concept of the line graph of a given graph is so natural that it has been independently discovered by many authors" [13]. The line graph has appeared in the literature by various names like interchange graph, edge-to-vertex dual, covering graph, derivative, etc. Informally, the line graph $L(G)$ of a graph $G$ is the result of taking the edges of a graph as the vertices of the new graph and joining the new vertices if the corresponding edges are incident by a vertex in graph $G$ (Figure 1.1). The line graph represents an alternative structure of the original graph, and integrating it in GNNs can be beneficial and increase the model’s expressiveness. Formally, the line graph of a graph is defined by [2] below.

**Definition 1.4.1.** Given a graph $G(\mathcal{N}, \mathcal{E})$ with at least one edge, the *line graph* of $G$, denoted by $L(G)$, is a graph whose vertices are the edges of $G$, with two of these vertices being adjacent if the corresponding edges are incident in $G$.

If $G_2 = L(G_1)$, we denote $G_1$ as the *root graph* of $G_2$. Since the line graph of a graph is still a graph, the process can be iterated and result in a sequence of line graphs (see Figure 1.2). Moreover, if $G$ is a non-trivial graph that is connected, the line graph $L(G)$ is also connected.

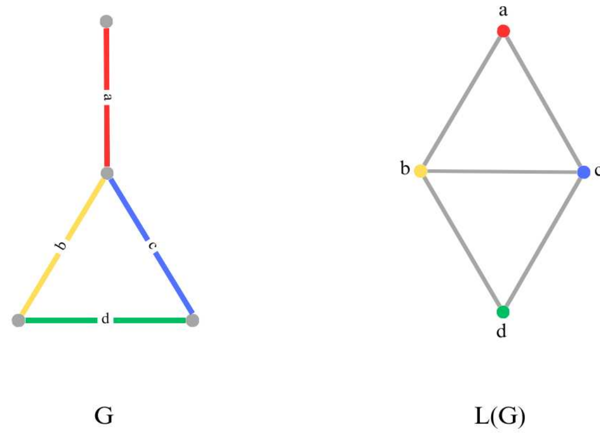


**Figure 1.1:** Example of a graph and its line graph.

### 1.4.2 Basic Properties

Although line graphs of special graphs like cycles and paths have interesting attributes, our aim is not to discuss any of the specified properties but rather to maintain the generality of this section. One of the useful properties of line graphs is the ability to determine the number of nodes and edges, which is desirable when defining a class of graphs [13].

**Theorem 1.4.1.** Let $G$ be a non-null graph with $n$ vertices and $m$ edges. Then

1. $L(G)$ has $m$ vertices and $\frac{1}{2}\sum(\deg v)^2 - m$ edges.
2. The degree of a vertex $e = uv$ in $L(G)$ is $\deg e = \deg u + \deg v - 2$.

The proof of the theorem can be found in [2]. We also mention briefly the properties of line graphs of some elementary families of graphs:

1. Paths: $L(P_n) \cong P_{n-1}$ for $n \geq 2$. (The line graph of a path $P_n$ with $n$ vertices is isomorphic to another path $P_{n-1}$.)
2. Cycles: $L(C_n) \cong C_n$. (The line graph of a cycle $C_n$ is isomorphic to the same cycle $C_n$.)
3. Stars: $L(K_{1,s}) \cong K_s$. (The line graph of a star graph $K_{1,s}$ is isomorphic to a complete graph $K_s$.)

The property concerning paths, which refers to the reduction of path length in the line graph, can be considered useful in capturing long-range interactions in GNN models, particularly in scenarios involving sparse graphs. This transformation essentially operates as an edge contraction, without loss of information, enabling the detection of distant interactions. However, it is important to limit the number of

transformations, as excessive iterations can over-simplify the graph structure, leading to information loss. Regarding cycle and star graphs, the ability to identify these patterns or sub-structures in the graph is valuable, especially in graph classification tasks. By employing a distinct transformation, the process of recognizing these specific patterns becomes much easier.

To understand the scope of the line graph, it's useful to know the smallest and largest degrees among its vertices. As we know, the degree of a vertex $e = uv$ in the line graph $L(G)$ is $\deg u + \deg v$. We denote $\delta(L(G))$ and $\Delta(L(G))$ as the smallest and largest degrees of the line graph $L(G)$, respectively. The observation below was made by [6], and it follows from Theorem 1.4.1.

**Theorem 1.4.2.** Let $G$ be a graph with at least one edge:

1. $\delta(L(G)) \geq 2\delta(G) - 2$ with equality if and only if $G$ has two adjacent vertices of degree $\delta(G)$.
2. $\Delta(L(G)) \leq 2\Delta(G) - 2$ with equality if and only if $G$ has two adjacent vertices of degree $\Delta(G)$.

Intuitively, the first bound suggests that the line graph $L(G)$ tends to have higher minimum degrees compared to the original graph $G$. This implies that, in general, the connectivity between edges in $L(G)$ is stronger than the connectivity between vertices in $G$. The second bound indicates that the maximum degree in the line graph $L(G)$ is typically lower in comparison to the maximum degree in the original graph $G$. This suggests that the line graph $L(G)$ tends to have a more balanced degree distribution compared to the original graph $G$. In summary, the transformation into a line graph tends to redistribute the degrees of nodes, potentially leading to a graph with stronger local connectivity and a more balanced degree distribution compared to the original graph.

### 1.4.3 Transformation Complexity

The line graph transformation can be divided into two transformations, (1) vertex transformation and (2) edge transformation. The line graph $L(G)$ has a vertex for each edge in $G$. Therefore, the complexity of vertex transformation is proportional to the number of edges in $G$, which is $O(m)$. Moreover, we know that for each vertex $v$ in $G$, its incident edges contribute to the edges in $L(G)$, and the number of edges in $L(G)$ is related to the sum of squares of the degrees of vertices in $G$ (see Theorem 1.4.1). The computation involves traversing all vertices in $G$, resulting in the edge transformation being $O(n)$. Finally, the overall complexity of transforming a non-null graph $G$ into its line graph $L(G)$ is $O(m) + O(n) = O(m + n)$.

### 1.4.4 Spectral Properties

In this section, we discuss some of the important spectral properties of graphs and line graphs, along with defining some necessary matrices. We begin by defining the adjacency matrix, an important matrix in graph theory.

**Definition 1.4.2.** Given a graph $G(\mathcal{V}, \mathcal{E})$ where $|\mathcal{V}| = n$ and $|\mathcal{E}| = m$, the adjacency matrix, denoted as $A = A(G)$, is an $n \times n$ matrix where $a_{ij}$ is 1 when vertices $v_i$ and $v_j$ are adjacent and 0 otherwise.

The adjacency matrix contains some of the properties of the graph itself. It can be shown by an induction proof that "the $(i, j)$ entry of the $k^{th}$ power of $A$ is the number of walks of length $k$ from vertex $v_i$ to vertex $v_j$" [2]. It is also widely used in GNN message-passing as it contains information about each node's neighbors. As the line graph is the main focus of this work, the *incidence matrix* is another graph matrix that has significant importance because it focuses on edges and their relations with vertices.

**Definition 1.4.3.** Given a graph $G(\mathcal{V}, \mathcal{E})$ where $|\mathcal{V}| = n$ and $|\mathcal{E}| = m$, the incidence matrix, denoted as $B = B(G)$, is an $n \times m$ matrix where $b_{ij}$ is 1 when vertex $v_i$ and edge $e_j$ are incident and 0 otherwise.

To define a line graph's adjacency matrix, we can make use of the incidence matrix in the following theorem proved by [2].

**Theorem 1.4.3.** For any graph $G$,

1. $A(G) = BB^T - D$
2. $A(L(G)) = B^T B - 2I$

where $D \in \mathbb{R}^{n \times n}$ is the diagonal degree matrix of $G$ and $I$ is the identity matrix.

The theorem provides insights into how the connectivity patterns and node-edge relationships in $G$ are transformed when constructing the line graph $L(G)$. Understanding how information flows and propagates in $G$ and $L(G)$ can aid in designing more effective graph embedding methods. Spectral embedding techniques, in particular, can benefit from insights provided by the theorem to capture structural similarities. Moreover, the theorem's description of the relationships between the adjacency matrices can be useful in identifying isomorphic graphs.

### 1.4.5 Iterated Line Graphs

Since the line graph is a graph itself, a sequence of iterated line graphs can be generated. Formally, we define the *iterated line graphs* of a graph $G$ as $L^1(G) = L(G)$, $L^2(G) = L(L(G))$, etc. In general,

$$L^n(G) = L(L^{n-1}(G)).$$

Figure 1.2 shows a graph and its sequence of iterated line graphs. These sequences can reveal underlying structural information in the graph. It is proved by [2] that if a graph is *prolific*, then the number of vertices in the iterated line graphs grows indefinitely by iteration. A prolific graph is defined by [2] as "a graph that is connected, has a vertex of degree at least 3, and is not $K_{1,3}$, in other words, if it is connected and not a path, cycle, or claw."

**Theorem 1.4.4.** Let $G$ be a prolific graph, and let $n_k$ be the number of vertices in $L^k(G)$. Then

$$\lim_{k \to \infty} n_k = \infty.$$

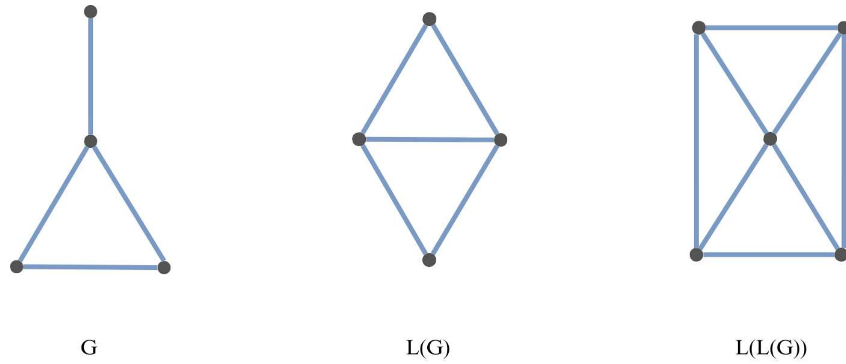


**Figure 1.2:** Example of a graph and its iterated line graphs.

The theorem implies that the line graphs of prolific graphs become increasingly expansive as the iteration proceeds. There's no upper bound to the growth of vertices in the iterated line graphs. This suggests that the iterations of line graphs continue to reveal new structural information about the original graph. In the next section, we discuss the implications of utilizing iterated line graphs to construct rich hierarchical graph representations. It is notable that the iterated line graphs tend to get larger at each iteration, causing considerable computational complexity.

### 1.4.6 Line Graphs in Graph Neural Networks

Despite the line graph's potential for building more representative learning schemes, the concept has not been thoroughly investigated. The work by [5] introduces LGNN (Line Graph Neural Network), a framework designed for the link prediction task. It converts the link prediction task to a binary classification task using line graph transformation and then leverages node classification approaches to solve the binary classification task on the transformed graph. Another work by [7], among other contributions, presents a GNN model based on a family of multi-scale graph operators that utilizes line graphs for the community detection problem.

The line graph, a straightforward graph transformation, has been particularly interesting in graph theory. However, its potential in the graph representation domain remains largely unexplored. With promising theoretical properties, line graphs suggest the possibility of creating more robust representations of their original graph, emphasizing edge connections and alternative representations. In this study, our objective is to develop a hierarchical graph neural network built upon iterated line graphs. We aim to establish an efficient flow of information by leveraging incidence matrices.

# Chapter 2

# Literature Review

## 2.1 Introduction

Graph representation learning is a field of research in the category of deep learning that has risen in popularity over the last two decades. The increasing amount of large graph-structured datasets, such as social networks and recommender systems, has driven the demand for effective graph representations. Moreover, the impressive performance of deep learning models in other areas, such as CNNs and RNNs, has motivated many to integrate the fundamentals of neural networks with graph data.

Initial works by [25] and [24] utilized recurrent networks to construct graph neural network models. However, it was the introduction of Graph Convolutional Networks (GCNs) by [16] that revolutionized graph data processing. Inspired by CNNs (Convolutional Neural Networks) [18], which prioritize local feature extraction and weight sharing to widen the receptive field, GCNs utilize *local message-passing* to build node representations that integrate local information.

Since then, many contributions have been made in this field, each derived from a different theoretical motivation. The comprehensive analysis of GNN models carried out by [36] classifies the existing computation modules for graph neural networks into three primary categories: (1) propagation modules, consisting of convolution and recurrent operators; (2) sampling modules, which prioritize inductive and more general solutions; and (3) pooling modules, or hierarchical modules, that construct hierarchical representations of the input graph to capture higher-level activities within the graph.

In this chapter, we review the aforementioned GNN modules, highlighting the most influential works in this domain. We recognize their contributions as well as their limitations. Our focus is particularly on the hierarchical modules and their diverse approaches and motivations. We then introduce our approach, HLGNN, which adopts a hierarchical methodology by constructing line graphs. We argue how this approach can address certain shortcomings of earlier models, notably the over-smoothing issue.

## 2.2 Propagation and Sampling Modules

In the context of Graph Neural Networks (GNNs), propagation modules play a critical role in adapting convolution operators from traditional domains to the graph domain, facilitating effective information aggregation across graph structures. These modules are inspired by both spectral and spatial approaches, aiming to generalize convolutional operations for graph-structured data. Notable examples include Graph Convolutional Networks (GCNs) [16] and Graph Attention Networks (GATs) [28], which enhance node representations by aggregating information from neighboring nodes with varying degrees of attention. Moreover, recurrent network architectures such as LSTM and GRU modules have been seamlessly integrated into graph models such as TreeLSTM [27] and GraphRNN [33], enabling sequential processing and capturing long-range dependencies within graph contexts.

Addressing the challenge of neighbor explosion in GNNs, particularly in deep architectures where the size of neighboring sets grows exponentially with network depth, sampling modules mitigate computational complexity by selectively sampling and aggregating information. Notably, GraphSAGE [4] exemplifies this approach, demonstrating its effectiveness in scaling GNNs for large-scale and inductive learning tasks while ensuring efficient transformation of graph data.

### 2.2.1 GCN (Graph Convolutional Network)

Inspired by CNNs in images, [16] proposes **GCN (Graph Convolutional Network)**. GCN operates by aggregating feature information from a node's local neighborhood, allowing each node to learn representations based on its neighbors' features. It simplifies spectral graph convolutions [12] by limiting dependence on the neighborhood of nodes. It also utilizes a single weight matrix per layer and keeps the learning stable by normalization techniques. The message-passing operation of GCN is defined as:

$$h_u^{(k)} = \sigma\left(W^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{h_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right) \tag{2.1}$$

where instead of taking the sum as aggregation, it takes the mean of neighbor nodes by applying symmetric normalization.

This approach has proved to be one of the popular and effective baseline GNN architectures, however, it has some limitations. The memory requirement of GCN grows linearly with the size of the dataset. For large and densely connected graph datasets, it might be beneficial to apply further approximations. It inherently does not support edge features and is only applicable to undirected graphs. The authors suggest handling this limitation by representing the original graph as an undirected bipartite graph with additional nodes that represent edges in the original graph, which interestingly, mirrors our approach.

Moreover, GCN's message-passing allows exposure to only the $K^th$ order neighborhood of a node for $K$ layers, and all of the contributing nodes have equal importance.

This property can limit the model from capturing long-range interactions and also cause over-smoothing in a deep GCN model.

### 2.2.2 GraphSAGE

Instead of training individual embeddings for each node, **GraphSAGE**, introduced by [11], learns to generate embeddings. The proposed framework, GraphSAGE (Sample and Aggregate), is designed to work with inductive node embedding problems. Inductive problems are especially difficult as the model needs to learn effective node representations that generalize to unseen data while preserving the graph's structural properties.

GraphSAGE trains "a set of aggregator functions" that extract information from a node's local neighborhood instead of learning a node embedding for each node. For instance, a mean aggregator in GraphSAGE can be expressed as :

$$h_u^k \leftarrow \alpha(W.MEAN(\{h_u^{k-1}\} \cup \{h_v^{k-1}, \forall v \in \mathcal{N}(u)\})). \tag{2.2}$$

The authors also experimented with two other aggregators, LSTM (inspired by RNNs) and MAX. Although GraphSAGE is inherently feature-based, it can learn the structural properties of the graph, even if the nodes are sampled randomly.

### 2.2.3 GAT (Graph Attention Network)

GATs utilize masked attentional layers. In each attentional layer, nodes perform self-attention using a shared attention mechanism. The importance of node $v$'s feature to node $u$ is determined by attention coefficients:

$$a_{uv} = a(Wh_uWh_v). \tag{2.3}$$

By stacking such layers, nodes attend over their neighborhood features, implicitly assigning weights to the neighboring nodes. GATs don't depend on graph structure and are suitable for both inductive and transductive problems. In general, the model allows every node to attend to every other node, discarding all structural information. The structural properties are included through performing masked attentions, computing $a_{uv}$ for nodes $v \in \mathcal{N}(u)$.

Overall, GATs are versatile, applicable to directed and undirected graphs, and handle inductive and transductive tasks. They offer computational efficiency through parallelization. Nonetheless, they overlook the graph's structural properties, which can be critical in graphs with complex structures or underlying patterns.

### 2.2.4 GIN (Graph Isomorphic Network)

In their study, [30] examine how **GIN (Graph Isomorphic Network)** challenges the expressive limitations of models such as GCN [16] and GraphSAGE [11]. Inspired by GNNs and the Weisfeiler-Lehman (WL) graph isomorphism test, GIN introduces a straightforward yet powerful framework for graph representation learning. The proposed approach integrates multi-layer perceptrons (MLPs) as an aggregation

function, enabling information aggregation not only from neighboring nodes but also from the node itself, thus preserving graph structure and local information:

$$h_u^{(k)} = MLP^{(k)}((1 + \epsilon^{(k)}) \cdot h_u^{(k-1)} + \sum_{v \in \mathcal{N}(u)} h_v^{(k-1)}). \tag{2.4}$$

The authors theoretically demonstrate that GIN's expressive power is equivalent to that of the Weisfeiler-Lehman (WL) test, achieving a high discriminative power among GNN models. However, challenges may arise in scaling GIN to large-scale graphs due to increased computational complexity and memory requirements. Additionally, GIN's reliance on initial node features could undermine its robustness when confronted with noisy data.

## 2.3 Hierarchical Modules

The inspiration behind the graph pooling operation derives from the pooling layers found in convolutional networks, aiming to capture more concise, higher-level representations. Flat GNN message-passing suffers from several limitations. It predominantly focuses on local feature extraction, leading to over-smoothing in deep networks and an inability to effectively capture complex structures within large-scale graphs. To address these challenges, hierarchical message-passing approaches have been introduced.

Hierarchical and pooling GNNs typically establish a sequence of alternative graph representations, known as the graph hierarchy, based on specific transformation schemes. This involves adapting message-passing to enable the transmission of messages both within each graph (intra-level propagation) and between graphs in the hierarchy (inter-level propagation). By incorporating this approach, GNNs gain access to various representation levels of the graph, ultimately resulting in a more expressive model. In this section, we review some of the important studies conducted on hierarchical GNNs.

### 2.3.1 Diff-Pool

Introduced by [32], Diff-Pool was the first graph pooling mechanism. Diff-Pool employs a differentiable soft cluster assignment technique within each layer of a GNN, assigning nodes to clusters based on their learned embeddings, analogous to the spatial operation in CNNs [17]. Through the learned cluster assignment $S^{(k)}$, it coarsens the graph as expressed by:

$$H^{(k+1)} = S^{(k)^T} P^{(k)} \in \mathbb{R}^{n_{k+1} \times d}, \tag{2.5}$$

$$A^{(k+1)} = S^{(k)^T} A^{(k)} S^{(k)} \in \mathbb{R}^{n_{k+1} \times n_{k+1}}. \tag{2.6}$$

Here, $H^{(k+1)}$ and $A^{(k+1)}$ are the new embedding and adjacency matrices, respectively. The assignment matrix $S^{(k)}$ and the node embedding matrix $P^{(k)}$ are learned independently using separate GNN modules:

$$P^{(k)} = GNN_{k,embed}(A^{(k)}, H^{(k)}), \tag{2.7}$$

$$S^{(k)} = GNN_{k,pool}(A^{(k)}, H^{(k)}). \tag{2.8}$$

Diff-Pool learns meaningful node clusters, enabling the development of deeper GNN models while preserving high-order patterns and capturing hierarchical community structures. It performs well in densely connected graphs, aggregating information with minimal loss. However, it may struggle to capture sparse graph structure, particularly interesting structures such as paths, cycles, and tree-like structures.

### 2.3.2 Att-Pool

Att-Pool, presented by [15], offers an attention-based pooling strategy aimed at generating hierarchical feature representations. By leveraging attention mechanisms, Att-Pool dynamically selects and aggregates node features, enabling the creation of more informative hierarchical representations. The authors propose two attention modules: global attention, which considers all nodes in the graph, and local attention, which focuses on extracting local features to keep the attention balanced across all regions of the graph.

The approach can effectively be integrated with various GCNs in a hierarchical manner. Additionally, Att-Pool enables the model to prioritize relevant nodes while discarding less informative ones, thereby enhancing representation learning. The heavy reliance on attention mechanisms in the approach can lead to scalability issues when applied to large-scale graphs. This is because attention mechanisms introduce computational overhead, which becomes more problematic as the size of the graph increases.

### 2.3.3 Graph Transformers

In their 2019 paper, [34] introduced Graph Transformer Networks (GTNs), which extend graph neural networks (GNNs) to handle heterogeneous graphs by generating new graph structures. GTNs identify useful connections between unconnected nodes and create effective node representations through an end-to-end learning process. The core component, the Graph Transformer layer, selects edge types and composite relations to form meta-paths, enabling powerful node representations without domain-specific preprocessing.

### 2.3.4 HGNet

HGNet (Hierarchical Graph Net), introduced by [23], tackles the limitations of flat message-passing modules commonly used in most GNN architectures. The method utilizes edge contraction [8] or the Louvain method [3] to construct a hierarchical structure. Similar to other hierarchical GNNs, HGNet employs two-directional hierarchical message-passing. This involves an upward pass through the hierarchy and a subsequent downward pass, utilizing GCN layers and contraction methods.

HGNet demonstrates superior performance compared to stacking of GCN layers, especially in molecular property prediction benchmarks. It achieves this by propagating information within $O(\log |V(G)|)$ steps instead of $O(\text{diam}(G))$, thereby effectively capturing long-range interactions in sparse graphs with large diameters.

### 2.3.5 HGNN

The work done by [20] presents another hierarchical GNN architecture utilized specifically for particle tracking. To construct the hierarchical representation, the authors propose GMPool, a method that integrates the connected components algorithm and Gaussian Mixture Model (GMM). Initially, GMPool computes node similarities based on their embeddings, followed by GMM-based classification of edges into in-cluster or out-of-cluster categories. The connected components algorithm then identifies remaining node groups, forming *super-nodes* with embeddings defined as centroids. *Super-edges* are formed by connecting nodes to their respective super-nodes, maintaining sparsity with a bipartite graph approach.

Regarding the message-passing mechanism, each node aggregates information from (1) adjacent node features, (2) super-node features weighted by bipartite graph weights, and (3) its own features. HGNN demonstrates better tracking efficiency performance and enhanced robustness against inefficient input graphs. It not only reduces the distance between nodes through its hierarchical approach but also effectively expands the receptive field, enabling message-passing between weakly connected components.

### 2.3.6 HC-GNN

HC-GNN exploits the Louvain method [3] to build up the hierarchical structure, which is then used for the hierarchical message-passing mechanism. Like other approaches, [35] presents a global message-passing for a hierarchical GNN. The structure is constructed by grouping the high-density inter-connected nodes. The message-passing is done in three phases: (1) within-level propagation, (2) bottom-up propagation, and (3) top-down propagation.

# Chapter 3

# Methodology

## 3.1 Introduction

Flat message-passing GNNs struggle to capture long-term interactions between nodes, often overlooking multi-level information within the graph due to their reliance solely on aggregating messages across the observed topological structure [35]. Additionally, over-smoothing issue is evident in many popular GNN models, where node representations lose their distinctiveness after several layers of GNN message-passing. Thus, it is beneficial to construct a hierarchical message-passing mechanism, integrating multi-level information present in the graph into node representations. This approach not only preserves locality through intra-level communication but also is able to capture high-level neighborhood properties by facilitating inter-level interactions between hierarchical layers.

To address the mentioned issues, we present **HLGNN (Hierarchical Line Graph Neural Network)**, a hierarchical GNN framework inspired by line graphs and their inherent properties. The hierarchical structure is formed through iterative line graphs, wherein each graph $G_t$ at level $t$ represents the line graph of the graph from the previous layer $t-1$ ($G_t = L(G_{t-1})$). Subsequently, a hierarchical message-passing mechanism is developed by utilizing GCN layers [16] and incidence matrices. The key concept involves constructing a hierarchical framework that contains (1) iterations of line graphs, and (2) intra and inter-level propagation schemes [35].

In most hierarchical GNN studies, the graphs in the hierarchy typically serve as a coarsened version of the original graph, intended to create shortcuts for capturing interactions between distant nodes. However, the coarsening process often leads to information loss, particularly structural details [32]. Conversely, iterated line graphs tend to expand with each iteration. As discussed in Chapter 1, line graphs address the issue of node degree imbalance, as stated in Theorem 1.4.1, thereby avoiding bias towards high-degree nodes. Furthermore, they have the capability to enhance the representation of specific sub-graphs, making them more noticeable, and ultimately reducing path lengths, resulting in a more balanced representation. Employing iterated line graphs for a hierarchical GNN framework offers several additional advantages:

1. The line graph of a graph focuses on edge properties, which are often overlooked in traditional GNNs. However, these properties can provide crucial information if aggregated effectively.

2. GNNs incorporating line graphs can expand their receptive field twice as fast as flat GNNs by aggregating information from a $2k$-hop neighborhood rather than just a $k$-hop neighborhood. This is because each node in the line graphs represents the intersection of two nodes in the original graph.

3. Iterated line graphs tend to expand in size, which is beneficial when dealing with small graphs in graph classification tasks. They can unveil information that might be obscured to a flat GNN and diminished due to the over-smoothing issue in deep GNN models.

Leveraging simple matrix multiplication operations, we construct a hierarchical GNN based on iterated line graphs. Intra-level aggregations can be interpreted as flat message-passing operations, that aggregate information within the graph. On the other hand, inter-level aggregations employ incidence matrices, serving as a bridge between the levels of hierarchy.

## 3.2 Hierarchy Construction

Our novel framework, HLGNN, leverages the iterative nature of line graphs to construct a hierarchical framework utilizing GNN layers. In the context of a line graph, the edges of the original graph become the vertices of the line graph, with line vertices connected if the corresponding edges in the original graph are incident to a vertex. Recognizing that the line graph itself is also a graph, one can generate a series of iterative line graphs (see Chapter 1).

The inherent predictability of line graph theory enables us to pre-compose the iterated line graphs, thereby enhancing the overall efficiency of the model. To generate the series of iterated line graphs, given the input graph $G_0 = G(|\mathcal{V}| = n, |\mathcal{E}| = m)$ with a node feature matrix $X \in \mathbb{R}^{n \times d}$ and the desired hierarchy level $T$, we establish a set of hierarchical graphs $\mathcal{S} = \{G_0, G_1, G_2, ..., G_T\}$ where $G_1 = L(G_0)$, $G_2 = L(G_1)$, and so forth. According to the definition of the line graph, we observe that $\mathcal{V}_{L(G)} = \mathcal{E}_G$ implying $n_{L(G)} = m_G$. For any arbitrary graph $G_t$ within the hierarchy $0 < t < T$, the graph's adjacency matrix $A$ and incidence matrix $B$ can be denoted as

$$A_t \in \mathbb{R}^{n_t \times n_t}, \quad B_t \in \mathbb{R}^{n_t \times n_{t+1}}. \tag{3.1}$$

HLGNN is designed to create more expressive node representations applicable to diverse graph-structured tasks such as node classification, graph classification, and community detection. In subsequent chapters, we delve into the model's application to node classification tasks.

## 3.3 Message-passing Mechanism

Let's delve into the concept of simple message-passing as outlined in Chapter 1. In the context of a graph $G(|\mathcal{V}| = n, |\mathcal{E}| = m)$, the computation of $h_u^k$, representing the hidden embedding of node $u$ at layer $k$, involves two primary operations AGGREGATE and UPDATE. Initially, we perform AGGREGATE by gathering the message $\mu_{\mathcal{N}(u)}$ from the neighboring nodes of $u$, followed by combining it with the node's embedding using the UPDATE function.

$$\mu_{\mathcal{N}(u)}^{(k)} = \text{AGGREGATE}(\{h_v^{(k-1)}, \forall v \in \mathcal{N}(u)\}), \tag{3.2}$$

$$h_u^{(k)} = \text{UPDATE}(h_u^{(k-1)}, \mu_{\mathcal{N}(u)}^{(k)}). \tag{3.3}$$

When adapting GNN message-passing for line graphs, we introduce a modification by performing message passing on both the input graph and its line graph. This entails employing simple message-passing operations to facilitate the exchange of information between the two graphs. We achieve this by utilizing the incidence matrix B to establish a bipartite graph between each graph and its corresponding line graph.

Illustrated in Figure 3.1, this process forms a bipartite graph between each pair of graphs in the hierarchy. We can draw a straightforward analogy: given that each vertex in $L(G)$ corresponds to an edge in $G$, the incidence matrix constructs a bipartite graph where the neighbors of nodes in $G$ align with the nodes in $L(G)$, thus creating a meaningful connection between the two graphs.

Analogous to message aggregation from neighbor nodes, for any graph within the hierarchy $G_t$ $(1 < t < T)$, we aggregate messages from the corresponding nodes in graphs $G_{t-1}$ and $G_{t+1}$ through *bottom-up* and *top-down* message-passing mechanisms. It's worth noting that for $G_0$ and $G_T$, we exclude bottom-up and top-down message aggregation, respectively. We utilize GCN (Graph Convolutional Network) layers [16] to effectively propagate information throughout the network.

### 3.3.1 Within-level Aggregation

The process involves aggregation within each graph, where the message $\mu_{\mathcal{N}(u)}^{(k)}$ is computed based on the neighborhood of the nodes within the same graph, represented by:

$$\mu_{\mathcal{N}(u)}^{(k)} = \text{AGGREGATE}(\{h_v^{(k-1)}, \forall v \in \mathcal{N}(u)\}). \tag{3.4}$$

### 3.3.2 Bottom-up Fusion

The operation involves a *bottom-up* strategy, where the message is aggregated for a line graph $G_t = L(G_{t-1})$ from its root graph $G_{t-1}$ using the bipartite graph constructing according to the transpose of the incidence matrix $B_{t-1} \in \mathbb{R}^{n_t \times n_{t-1}}$ of graph $G_{t-1}$ (see Figure 3.2). For each node $u$ in $G_t$, the bottom-up message at layer $k$ is computed as:

$$\mu_{\mathcal{I}_{t-1}(u)}^{(k)} = \text{AGGREGATE}(\{h_v^{(k-1)}, \forall v \in \mathcal{I}_{t-1}(u)\}). \tag{3.5}$$

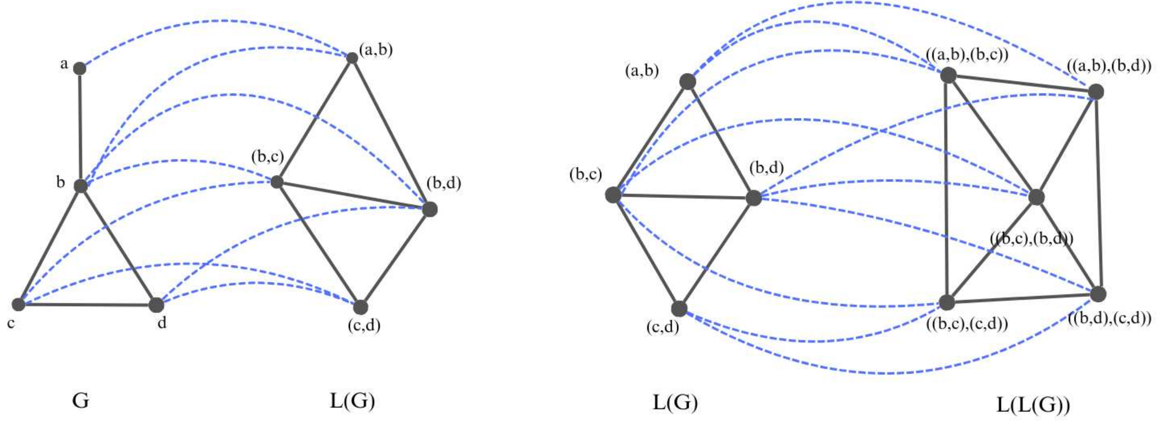


**Figure 3.1:** The construction of the bipartite graphs between the graphs in the hierarchy using the incidence matrix. On the left, we have a graph $G$ and its line graph $L(G)$, and on the right, another iteration of line graphs, $L(G)$ and $L(L(G))$. A bipartite graph is then formed between the graphs throughout the hierarchy using the incidence matrices. The blue dashed lines represent the edges of the bipartite graph.

where $\forall v \in \mathcal{I}_{t-1}(u)$ are all the nodes in $G_{t-1}$ that correspond (are neighbors) to node $u$ in the bipartite graph constructed by the incidence matrix $B_{t-1}^T$.

### 3.3.3 Top-down Fusion

The top-down fusion is employed when transmitting messages from the line graph $G_{t+1}$ to its root graph $G_t$. This process operates in reverse compared to bottom-up fusion, utilizing the incidence matrix $B_t \in \mathbb{R}^{n_t \times n_{t+1}}$ of graph $G_t$ (See Figure 3.2). For every node $u$ in graph $G_t$, the top-down message is aggregated from graph $G_{t+1}$ as follows:

$$\mu_{\mathcal{I}_t(u)}^{(k)} = \text{AGGREGATE}(\{h_v^{(k-1)}, \forall v \in \mathcal{I}_t(u)\}). \tag{3.6}$$

where $\forall v \in \mathcal{I}_t(u)$ are all the nodes in $G_{t+1}$ that correspond (are neighbors) to node $u$ in the bipartite graph constructed by the incidence matrix $B_t$.

### 3.3.4 Update Function

Finally, $h_{u_t}^{(k)}$, representing the $k^{th}$ embedding of node $u$ in graph $G_t$, is defined as:

$$h_{u_t}^{(k)} = \text{UPDATE}(h_u^{(k-1)}, \mu_{\mathcal{N}(u)}^{(k)}, \mu_{\mathcal{I}_{t-1}(u)}^{(k)}, \mu_{\mathcal{I}_t(u)}^{(k)}). \tag{3.7}$$

The update function in the proposed approach can vary depending on the task or the data. However, three widely used methods are primarily employed: sum, mean, and max.

## 3.4 Theoretical Analysis

In this section, we conduct a theoretical analysis of our model, focusing on key properties and computational complexities. We begin by examining the permutation

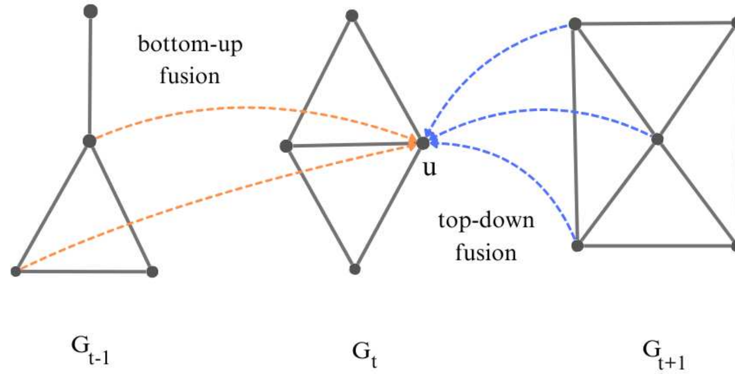


**Figure 3.2:** Illustration of the message-passing mechanism for node $u$ in Graph $G_t$ within the hierarchy. The orange dashed lines depict the bottom-up fusion operation, aggregating information from $G_{t-1}$, while the blue dashed lines represent the top-down fusion operation, aggregating information from corresponding nodes in $G_{t+1}$. These fusion operations, combined with within-level aggregations in $G_t$, are integrated using the UPDATE function.

invariance property of our model, which is crucial for maintaining consistency in feature aggregation and update mechanisms across different levels of hierarchical graphs. Furthermore, we analyze the computational complexities involved in constructing hierarchical graphs and implementing the message-passing mechanism.

### 3.4.1 Permutation Invariance

Our model $f(G, X)$ satisfies the permutation invariance property. Within-level aggregation processes consider sets of neighbors for each node, and regardless of the order of the nodes, the resulting aggregation remains unchanged. Similarly, bottom-up and top-down aggregations aggregate features from sets of nodes in the bipartite graphs, independent of their order. The update rule, typically involving summation or averaging, is also permutation invariant as it operates on all messages without relying on any specific ordering. Given that both aggregation and update functions demonstrate permutation invariance, we can conclude that the overall model maintains this property. A proof on permutation invariance provided by [7] can be readily extended to our approach.

### 3.4.2 Computational Complexity

The complexity of constructing the hierarchical graphs primarily depends on the size of the input graph and the desired hierarchy level $T$. Generating each iterated line graph involves transforming the edges of the original graph into vertices and establishing connections based on incidences, which can be done in linear time (see Chapter 1). Therefore, the overall complexity for constructing the entire hierarchy $S = \{G_0, G_1, G_2, \ldots, G_T\}$ is $O(T \cdot (n + m))$, where $n$ is the number of nodes and $m$ is the number of edges in the original graph.

The computational complexity of message-passing depends on the number of nodes and edges in the graphs involved and the number of layers in the GNN. Within-level message-passing involves aggregating messages from neighboring nodes within the same graph, which typically has a complexity of $O(n \cdot d)$ where $d$ is the dimensionality of the node embeddings.

Bottom-up and top-down message-passing require aggregating messages across different graphs using the incidence matrix, which involves matrix multiplication and has a complexity of $O(n_{t-1} \cdot n_t \cdot d)$ where $n_t$ is the number of nodes on graph $G_t$ and $n_{t-1}$ is the number of nodes in graph $G_{t-1}$.

Therefore, the overall complexity for message-passing across all layers and graphs is $O(K \cdot (n \cdot d + \sum_{t=1}^{T-1} n_{t-1} \cdot n_t \cdot d))$, where $K$ is the number of layers in the GNN and $T$ is the number of hierarchical levels.

# Chapter 4

# Experiments

## 4.1 Introduction

In this chapter, we delve into the experiments conducted to evaluate the feasibility, applicability, and effectiveness of our proposed framework, HLGNN, in enhancing the expressiveness, robustness, and overall performance of graph neural networks. Our experimentation primarily focuses on fully-supervised node classification tasks, although our approach is applicable to a broader range of graph-structured tasks.

We begin by formulating the task and discussing the utilized loss function, followed by a brief introduction of the datasets employed in our experiments: Cora, Pubmed, and Citeseer, which are widely used to assess the performance of innovative approaches in the GNN domain. We provide a comprehensive overview of the experimental setup, including implementation details supported by flowcharts and diagrams, model selection procedures, hyper-parameter grid search, and training settings.

## 4.2 Task Setup

We utilize our model for the node classification task. A node classification task in graphs can be formally defined as follows: Given a graph $G(\mathcal{V}, \mathcal{E})$ and a feature matrix $X$, where each node in the graph belongs to one of $c$ classes and has a feature of dimension $d$, the objective is to predict a node's class based on its learned node embedding.

### 4.2.1 Fully-supervised vs. Semi-supervised

Node classification can be conducted in *fully-supervised* or *semi-supervised* settings. In fully-supervised settings, the model is trained on a dataset where the majority of nodes are labeled, enabling the model to learn from these labeled examples to make accurate predictions on unseen data. However, in semi-supervised settings, only a small subset of the data is labeled, and the model leverages both labeled and unlabeled data to enhance its performance. In our experiments, we set up our model to function under **fully-supervised conditions**. This results in an 80% training, 10% validation, and 10% test data partition.

### 4.2.2 Loss Function

We utilize the cross entropy loss as the loss function in our experiments. The cross entropy loss is a common choice for classification tasks, including node classification in graphs. It measures the difference between the predicted probability distribution and the target distribution. For a multi-class classification task, where $y_i$ is a binary indicator if class label $i$ is the true label for the observation, and $p_i$ is the predicted probability that observation is of class $i$, the cross-entropy $\mathcal{L}$ is given by:

$$\mathcal{L} = -\sum_i y_i \log(p_i). \tag{4.1}$$

## 4.3 Datasets

In the experiment phase, we utilize three widely recognized citation graphs that have been extensively employed in GNN frameworks. Below, we offer a brief description of each dataset.

**Cora** is a graph-based dataset that represents citations among scientific papers. Widely utilized for node classification tasks, it stands as a cornerstone dataset for evaluating GNN models. The version of Cora used in the experiments comprises 2708 scientific publications classified into seven classes, with a total of 5429 citation connections. Each node symbolizes a scientific paper, while edges represent citations, denoting one paper citing another. With a feature dimensionality of 1433, Cora encapsulates a dictionary of unique words, each assigned a binary value (0/1) to denote its presence or absence in a paper's content.

**Pubmed** database is a widely recognized repository of biomedical literature, containing vast amounts of information on research articles, abstracts, and citations from various biomedical fields. It serves as a vital resource for researchers and scholars in the medical domain. The Pubmed dataset consists of 19717 scientific publications derived from the Pubmed database, focusing on diabetes research and categorized into three classes. The citation network within this dataset is composed of 44,338 links. Each publication is represented by a TF/IDF weighted word vector derived from a dictionary containing 500 unique words.

**Citeseer** is a well-known digital library and search engine for academic and scientific literature in the computer and information science field. It contains a vast collection of research papers, articles, and citations from various conferences and journals. With 3312 publications classified into 6 distinct classes, Citeseer offers a comprehensive resource for researchers and scholars. The citation network consists of 4732 links. Each publication in the dataset is represented by a 0/1-valued word vector, where the presence or absence of words from the dictionary is indicated. This dictionary contains 3703 unique words, providing a detailed characterization of the content of each publication.

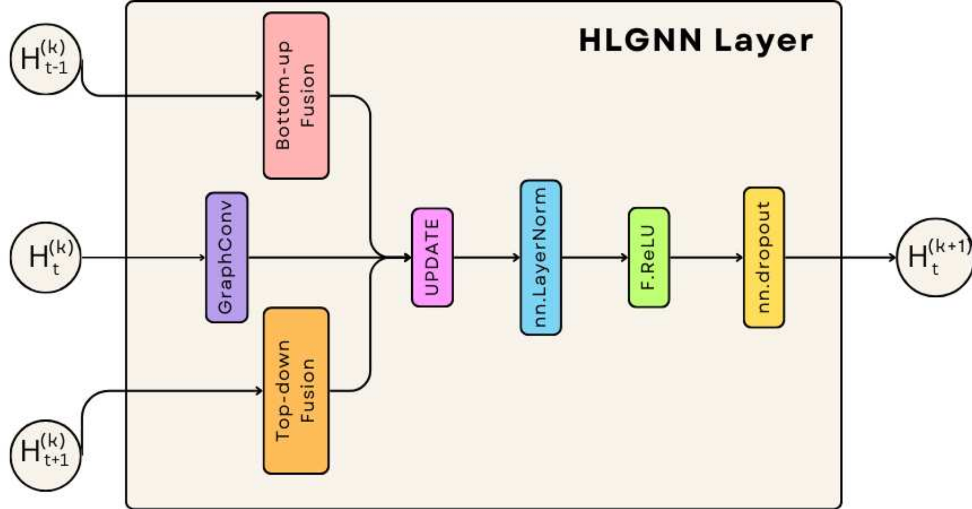


**Figure 4.1:** A scheme of the implementation of the HLGNN layer. The diagram highlights the important parts of the model construction, including inter-level fusions and convolution operators.

## 4.4 Implementation

We implemented our model using the Python language, leveraging well-known machine learning libraries like PyTorch and NumPy. Additionally, we utilize the Deep Graph Library (DGL), designed for graph-data processing, to access datasets and perform various graph processing tasks, such as line graph transformations and graph convolution operations. The central component of the implementation is what we refer to as the *HLGNN Layer*, which encompasses all message passing operations, convolutional layers, as well as the incorporation of non-linearities and normalization techniques.

Each layer is structured to accommodate a maximum of three hidden embedding matrices: (1) representing the current graph, and (2) and (3) representing the preceding and succeeding graphs in the hierarchy. Illustrated in Figure 4.1, within each layer, the aggregated messages from the graphs are merged using the UPDATE function. This function, which could be either a sum, max, or mean operation, is adaptable based on the specific task and dataset and is determined as a hyper-parameter during the model selection phase. Following the merging of embeddings, a layer-normalization is applied for stabilization, followed by the ReLU activation function, and finally, a layer-wise dropout operation is performed.

The hierarchical connection between graphs is facilitated through two message passing operators: (1) top-down fusion and (2) bottom-up fusion. As their names imply, top-down fusion transmits messages from the upper or subsequent graph in the hierarchy, or the line graph of the current graph. Similarly, bottom-up fusion maintains connections from the preceding graph in the hierarchy, or the root graph of the current graph. As detailed in Chapter 3, the inter-level message passing is executed leveraging the incidence matrix. The functionality of these operations is

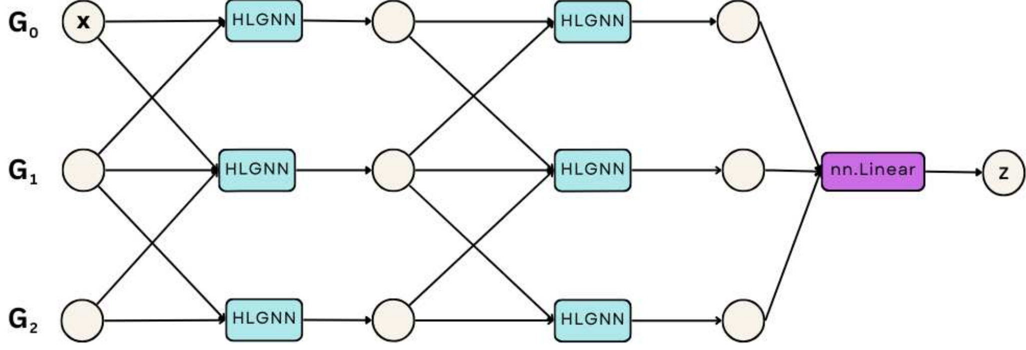


**Figure 4.2:** An example of the HLGNN architecture, with a hierarchical level of 3, and two hidden (HLGNN) layers. The generated line graphs' embeddings are initialized as zero, and the hidden layers are applied to the graphs independently. The output embedding is constructed by feeding the concatenation of the final embeddings into a fully connected layer.

depicted in Figure 4.3.

Finally, we can assemble the HLGNN layers to form the complete neural network architecture. Starting from the input graph, a series of iterated line graphs is generated, collectively constructing a hierarchy. This hierarchy, or the collection of graphs, is then fed into the network. Each graph undergoes the operations within HLGNN **independently**, with their embeddings influenced solely by the top-down and bottom-up fusion mechanisms (see Figure 4.2). The embedding matrices of the line graphs are initialized as zeros, gradually learning the embeddings from the original graph through information flow within the neural network. The hierarchy level, representing the number of line graphs, can be regarded as a hyper-parameter, varying across different tasks. To construct the final embedding, all embeddings are concatenated and subsequently fed into a fully connected layer.

### 4.4.1 GraphConv

The convolution layer is implemented using the DGL library's built-in GraphConv layer, which is based on the work of [16]. Mathematically, it is defined as follows:

$$\text{GraphConv}(h_u^{(k)}) = \alpha \left( \sum_{v \in \mathcal{N}(u)} \frac{1}{\sqrt{|\mathcal{N}(v)|}\sqrt{|\mathcal{N}(u)|}} h_v^{(k)} W^{(k)} + b^{(k)} \right) \tag{4.2}$$

where $b^{(k)}$ is the bias term and $\alpha$ is the activation function, which is set to none in our implementation, as we found performance improvement when applying non-linear activation after the update function. The update function combines the information aggregated from the convolution layer, and the top-down and bottom-up fusions. We initially selected GraphConv for its straightforward operation and effectiveness in preserving locality. However, the method can be adapted to work with any convolution method.

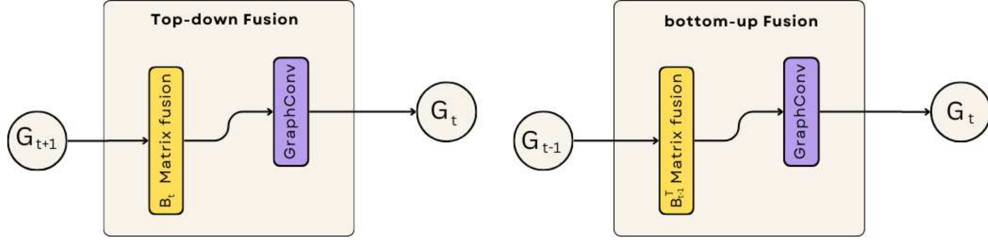


**Figure 4.3:** A scheme of the message-passing operations leveraging the incidence matrix $B$. On the left, the top-down fusion is shown, transmitting messages from the next graph. On the right, the bottom-up fusion is illustrated, aggregating messages from the previous graph. Both operations utilize the incidence matrix and subsequently apply a convolution layer, *GraphConv*, to generate the new embeddings.

### 4.4.2 Fusions

Fusions facilitate the integration of information flow within the hierarchical structure and enable message-passing between graphs in the hierarchy. In our method, we employ two types of fusions: bottom-up and top-down. Let $H_t^{(k)} \in \mathbb{R}^{n_t \times d}$ denote the hidden node embedding matrix of $G_t$. To integrate information from the graphs $G_{t-1}$ and $G_{t+1}$, the fusions are defined as follows:

#### Bottom-up Fusion

Bottom-up fusion integrates the embeddings of the previous graph $G_{t-1}$ within a hierarchical graph structure using the transpose of the incidence matrix of $G_{t-1}$. Let $H_{t-1}^{(k)} \in \mathbb{R}^{n_{t-1} \times d}$ represent the hidden embedding matrix of $G_{t-1}$, where $n_{t-1}$ is the number of nodes and $d$ is the feature dimension. The bottom-up fusion is defined as:

$$\text{BottomUpFusion}(H_{t-1}^{(k)}) = \text{GraphConv}(B_{t-1}^T H_{t-1}^{(k)}) \tag{4.3}$$

where $B_{t-1}^T \in \mathbb{R}^{n_t \times n_{t-1}}$ is transpose of the incidence matrix of $G_{t-1}$, and $m_{t-1} = n_t$ according to the definition of the line graph. This ensures that $B_{t-1}^T H_{t-1}^{(k)} \in \mathbb{R}^{n_t \times d}$ matches the dimensionality of the node embeddings of $G_t$, facilitating their integration in the update phase.

#### Top-down Fusion

Top-down fusion integrates the embeddings of the next graph $G_{t+1}$ within a hierarchical graph structure using the incidence matrix of $G_t$. Let $H_{t+1}^{(k)} \in \mathbb{R}^{n_{t+1} \times d}$ represent the hidden embedding matrix of $G_{t+1}$, where $n_{t+1}$ is the number of nodes and $d$ is the feature dimension. The top-down fusion is defined as:

$$\text{TopDownFusion}(H_{t+1}^{(k)}) = \text{GraphConv}(B_t H_{t+1}^{(k)}) \tag{4.4}$$

where $B_t \in \mathbb{R}^{n_t \times n_{t+1}}$ is the incidence matrix of $G_t$, and $m_t = n_{t+1}$ according to the definition of the line graph. This ensures that $B_t H_{t+1}^{(k)} \in \mathbb{R}^{n_t \times d}$ matches the dimensionality of the node embeddings of $G_t$, facilitating their integration in the update phase.

### 4.4.3 Update Function

The goal of the update function is to integrate the information aggregated from the convolution layer, as well as the bottom-up and top-down fusions, to generate the node embeddings of the graph. Each component's contribution is weighted by learnable parameters, and three different operations can be applied: sum, mean, and max. The choice of the update function is considered a hyper-parameter in the model selection process. The overall structure of the update function is as follows:

$$\begin{aligned} H_t^{(k+1)} = \text{UPDATE}\Big(&W_{\text{conv}}\text{GraphConv}(H_t^{(k)}), \\ &W_{\text{bottom-up}}\text{BottomUpFusion}(H_{t-1}^{(k)}), \\ &W_{\text{top-down}}\text{TopDownFusion}(H_{t+1}^{(k)})\Big) \end{aligned} \tag{4.5}$$

where $W_{\text{conv}}$, $W_{\text{bottom-up}}$, and $W_{\text{top-down}}$ are learnable weights determining the contribution of each component. The UPDATE function can be implemented using sum, mean, or max operations.

### 4.4.4 ReLU Activation

For the activation function, we use ReLU (Rectified Linear Unit), is a popular activation function used in neural networks and deep learning models. It introduces non-linearity to the model by allowing the neuron to pass through values greater than zero unchanged while setting all negative values to zero. The function can be expressed as:

$$ReLU(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{4.6}$$

### 4.4.5 Normalization

Using a normalization layer in the neural network provides several key benefits, including stabilizing gradients, speeding up convergence time, and improving overall performance. Normalization prevents issues with excessively large or small updates during back-propagation, such as gradient vanishing or exploding. Additionally, normalization enhances the model's performance by ensuring balanced learning across features, reducing the risk of poor local minima, and making the model more robust to variations in input data. We utilize PyTorch's built-in Layer Normalization, which is implemented as described in [1]:

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} \cdot \gamma + \beta. \tag{4.7}$$

Here, $\mathbb{E}[x]$ represents the expectation (mean) of $x$, $\text{Var}[x]$ represents the variance of $x$, $\epsilon$ is a small constant added for numerical stability, and $\gamma$ and $\beta$ are learnable parameters.

### 4.4.6 Dropout

Dropout is a regularization technique used in neural networks to prevent over-fitting. It works by randomly "dropping out" a fraction of the neurons during each training iteration, which helps the model learn more robust features and prevents it from becoming overly reliant on specific neurons. This technique also enhances generalization, leading to better performance on unseen data. Additionally, dropout breaks co-adaptations, which occur when neurons adjust to compensate for each other's presence, resulting in a more independent and distributed learning process.

We utilize PyTorch's built-in function for Dropout, in which the zeroed elements are chosen independently for each forward call and are sampled from a Bernoulli distribution. The approach is proven to be effective for regularization and preventing co-adaptation of neurons as described in [14]. We apply the dropout layer as shown in Figure 4.1, after the non-linear activation. The dropout rate is determined as a hyper-parameter during the model selection procedure.

### 4.4.7 Weight Decay

Weight decay, also known as L2 regularization, is a technique used to prevent over-fitting by penalizing large weights in the model. It achieves this by adding a term to the loss function that is proportional to the sum of the squared weights. This encourages the model to find simpler and more generalized patterns in the data, thereby reducing the likelihood of fitting noise and irrelevant patterns present in the training data.

In PyTorch, weight decay can be implemented using the built-in method in the Adam optimizer. The modified loss function with weight decay is expressed as:

$$L' = L + \frac{\lambda}{2} \sum_i w_i^2 \tag{4.8}$$

where $L$ is the original loss function, $\lambda$ is the weight decay coefficient, and $w_i$ represents the weights in the model.

## 4.5 Model Selection

We carry out the model selection process by engaging in hyper-parameter tuning and adjusting training settings. With a focus on the node classification task, our experiments are conducted across three datasets: Cora, Pubmed and Citeseer. The optimal model is chosen independently for each dataset and may vary in settings. The selection is based on the highest validation accuracy achieved.

### 4.5.1 Hyper-parameters

Hyper-parameters are essential in the experimentation phase as they uncover a model's genuine abilities. Without fine-tuning these parameters, it's challenging to understand a model's full potential. Hence, during model selection, we employ

techniques like grid search to identify the optimal combination of hyper-parameters, refining the model for specific tasks.

Many of these hyper-parameters are commonly employed across various machine learning models, with some tailored to specific aspects like line graphs. This enables us to experiment with different configurations and determine the most effective ones for each task.

| Hyper-parameter | Cora | Pubmed | Citeseer |
|---|---|---|---|
| Hierarchy Levels | 1, 2 | 1, 2 | 1, 2 |
| Number of Layers | 4, 6, 8 | 4, 6, 8 | 4, 6, 8 |
| Hidden Units | 16, 32 | 16, 32 | 16, 32 |
| Dropout Rate | 0.0, 0.2, 0.5 | 0.0, 0.2, 0.5 | 0.0, 0.2, 0.5 |
| Weight Decay | 0.0, 5e-3, 5e-6 | 0.0, 5e-3, 5e-6 | 0.0, 5e-3, 5e-6 |
| Activation Function | ReLU | ReLU | ReLU |
| Back-Tracking | True, False | True, False | True, False |
| Update Function | Sum, Mean, Max | Sum, Mean, Max | Sum, Mean, Max |
| Learning Rate | 0.01 | 0.01 | 0.01 |
| Epochs | 250 | 250 | 250 |

**Table 4.1:** Hyper-parameters used for training models on the Cora, Pubmed, and Citeseer datasets.

### 4.5.2 Training Settings

For each test configuration, we conduct 5 runs and aggregate the results by computing the mean. During each run, the model undergoes training for a maximum of 250 epochs, employing an early stopping policy set with a tolerance of up to 150 epochs. We utilize the Adam optimizer, a default optimizer function in PyTorch, with a learning rate of 0.01.

# Chapter 5

# Results

## 5.1 Introduction

In this chapter, we present and analyze the results of our experiments designed to assess the effectiveness of our proposed approach in enhancing the performance of GNN models. Our objective is to determine whether HLGNN effectively mitigates the over-smoothing issue. We evaluate various aspects of the model, including comparisons with baseline models, training performance, and additional assessments to demonstrate the efficacy of our approach.

## 5.2 Baseline Comparison

To ensure a precise comparison with baseline models, we followed the experimental settings outlined in [22]. For effective hyper-parameter tuning, we employed grid search on the set of hyper-parameters discussed in previous sections. Table 5.1 presents the accuracy comparison of the proposed HLGNN model against three baseline models (GCN, GraphConv, GAT) on the fully-supervised node classification task across three datasets: Cora, Pubmed, and Citeseer.

| Model | Cora | Pubmed | Citeseer |
|---|---|---|---|
| GCN | 88.5±0.70 | 88.5±0.50 | **77.7±0.80** |
| GraphConv | 87.6±0.20 | 89.0±0.40 | 76.2±1.20 |
| GAT | 87.3±1.10 | 87.2±0.40 | 76.0±1.44 |
| HLGNN | **88.8±0.04** | **89.2±0.00** | 77.3±0.01 |

**Table 5.1:** Accuracy comparison of the proposed model (HLGNN) against three baseline models (GCN, GraphConv, GAT) on a fully-supervised node classification task. ask. Model selection is based on results from the validation set. The highest accuracy is highlighted in bold for each dataset.

For the Cora dataset, HLGNN achieved an accuracy of 88.8±0.04, which is the highest among all models tested. The GCN model follows closely with an accuracy of 88.5±0.70, while GraphConv and GAT models have slightly lower accuracies of

87.6±0.20 and 87.3±1.10, respectively. This demonstrates that HLGNN outperforms the baseline models, providing a slight edge in accuracy over GCN and a more noticeable improvement over GraphConv and GAT.

On the Pubmed dataset, HLGNN also shows superior performance with an accuracy of 89.2±0.00. This is slightly higher than the GraphConv model, which has an accuracy of 89.0±0.40. The GCN model achieved an accuracy of 88.5±0.50, and the GAT model had the lowest accuracy at 87.2±0.40. Therefore, HLGNN again surpasses all baseline models, with a particularly significant improvement over GAT.

For the Citeseer dataset, GCN holds the highest accuracy at 77.7±0.80, followed closely by HLGNN with an accuracy of 77.3±0.01. GraphConv and GAT models have accuracies of 76.2±1.20 and 76.0±1.44, respectively. While HLGNN does not outperform GCN in this case, it still demonstrates competitive performance, significantly exceeding the accuracies of GraphConv and GAT.

In summary, the best baseline model changes with each dataset: GCN for Cora and Citeseer, and GraphConv for Pubmed. However, HLGNN consistently shows high performance across all datasets, surpassing or closely matching the best baseline model in each case. This indicates the effectiveness and robustness of HLGNN across different datasets.

## 5.3 Training Performance

Our approach, HLGNN, demonstrated a significant convergence rate across all datasets. Its hierarchical structure allows HLGNN to extract complex data representations without requiring multiple layers, as explained in Chapter 1.

| Hyper-parameter | Cora | Pubmed | Citeseer |
|---|---|---|---|
| Number of Layers | 6 | 8 | 6 |
| Hidden Units | 32 | 32 | 32 |
| Dropout Rate | 0.2 | 0.2 | 0.5 |
| Weight Decay | 0.0 | 0.0 | 0.0 |
| Learning Rate | 0.01 | 0.01 | 0.01 |
| Activation Function | ReLU | ReLU | ReLU |
| Back-Tracking | True | False | True |
| Update Function | SUM | SUM | MAX |
| Epochs | 250 | 250 | 250 |

**Table 5.2:** Hyper-parameters used for training models on the Cora, Pubmed, and Citeseer datasets.

Figure 5.1 displays the learning curves for the three datasets. For the Cora dataset, there is a rapid increase in accuracy during the initial epochs. Training accuracy stabilizes around 0.95 after approximately 100 epochs, while testing and validation accuracies peak around 0.9, with slight fluctuations. The model for the Cora dataset utilizes the hyper-parameters listed in Table 5.2, with the SUM update

function and backtracking operation enabled. This configuration focuses on robust training and leverages locally connected patterns, which accounts for the model's stable and strong performance on this dataset.

The accuracy plot for the Pubmed dataset (Figure 5.1) shows a continuous upward trend in training accuracy. The smaller gap between training and testing/validation accuracies compared to other datasets indicates the model's effectiveness in generalizing to unseen data. The selected model for the Pubmed dataset, as outlined in Table 5.2, incorporates a non-backtracking operator, using sparse connections to capture long-range interactions between nodes.

For the Citeseer dataset, the accuracy and loss plots indicate that the model converges within the first 50 epochs, with performance stabilizing afterward. There is a notable gap between training and validation/test accuracy after convergence, suggesting a slight over-fitting issue. The selected model uses the MAX update function, which explains the fluctuations in the validation/test trends.

Overall, the model demonstrates robust performance, achieving high accuracy across training, testing, and validation sets, with some variations reflecting dataset-specific challenges.

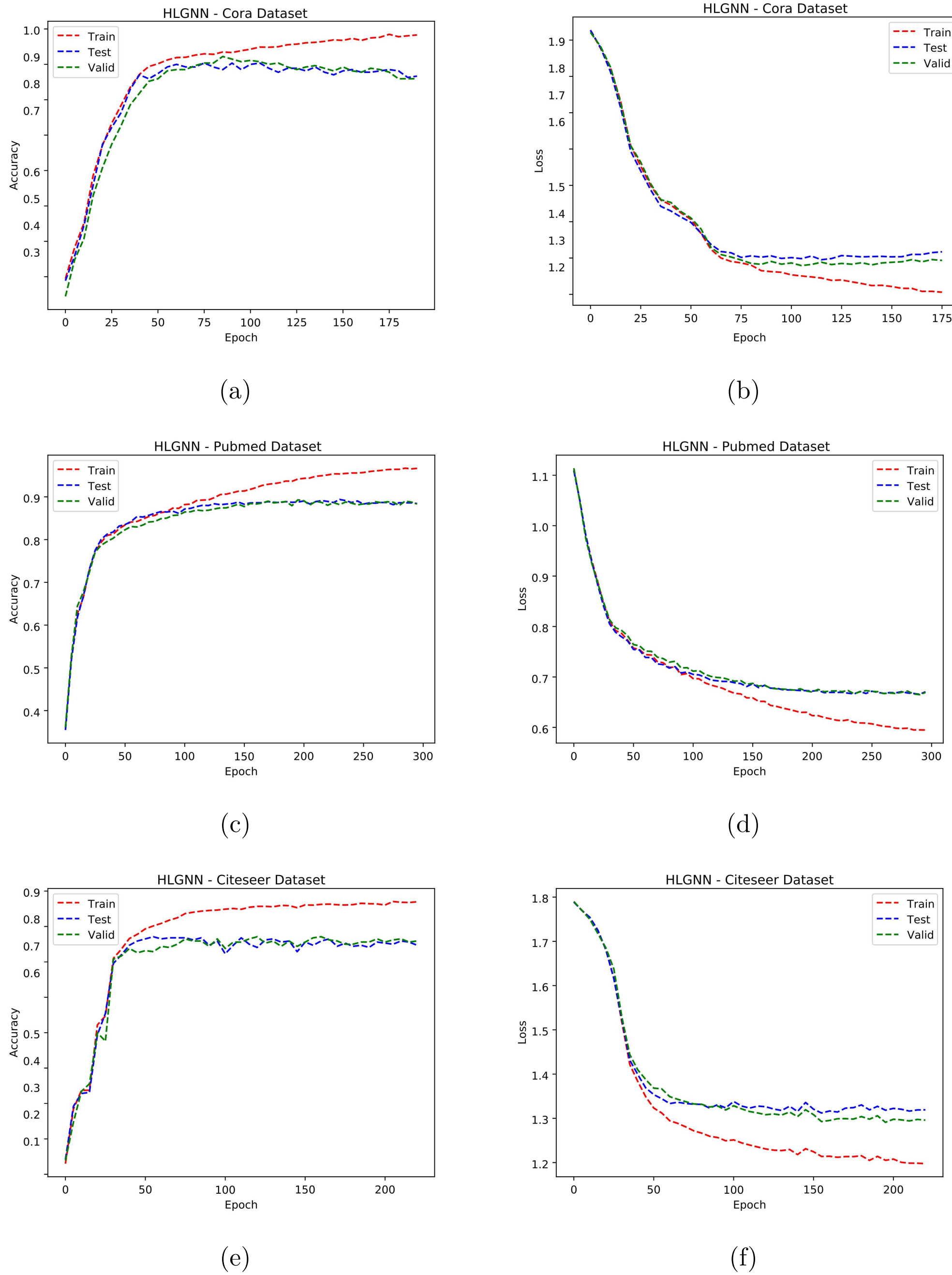


**Figure 5.1:** Accuracy and loss curves for the training, validation, and test sets on the three datasets. For the Cora dataset, (a) the accuracy plot shows rapid convergence and stable performance across all sets, while (b) the loss plot demonstrates a steady decrease in loss, indicating effective learning. For the Pubmed dataset, (c) displays consistent improvement in accuracy, and the loss plot (d) shows a continuous reduction in loss, reflecting the model's robust training process and generalization capability. For the Citeseer dataset, both (e) and (f) illustrate the model's convergence and stabilization, with slight fluctuations.

## 5.4 Model Evaluation

Assessing a model's performance requires looking at various factors. While accuracy and loss are essential for understanding learning capability, they alone don't offer a comprehensive view. Therefore, we incorporate additional metrics such as ROC curves and confusion matrices to assess the model's robustness, and utilize t-SNE for node embedding visualizations to examine the model's interpretability.

### 5.4.1 Confusion Matrices

For a multi-class classification task, the confusion matrix will be a $\mathcal{C} \times \mathcal{C}$, where $\mathcal{C}$ is the number of classes in the dataset. Each row of the confusion matrix represents the instances of the true label and each column represents the instances of the predicted label.

The confusion matrix heat-maps for the three datasets are displayed in Figure 5.2. For the Citeseer dataset, the classifier exhibited strong performance with high precision and recall for most classes, particularly for Class 2 and Class 5, with F1-scores of 0.833 and 0.881 respectively. However, notable misclassifications occurred between Classes 2 and 3, as well as Classes 1 and 0/4, indicating potential feature overlap or similarity between these classes. In the Cora dataset, the classifier demonstrated robust performance with F1-scores exceeding 0.94 for most classes, though there were some misclassifications between Classes 5 and 6, suggesting areas for further feature refinement.

For the Pubmed dataset, the classifier showed excellent performance, particularly for Class 1, with an F1-score of 0.963, highlighting its ability to accurately distinguish this class. Nevertheless, minor misclassifications were observed between Classes 0 and 2, which could be attributed to shared feature characteristics. Overall, the analysis indicates that while the classifiers are generally effective, there are specific class pairs with higher confusion rates that warrant further investigation and potential model adjustments to enhance classification accuracy across all classes.

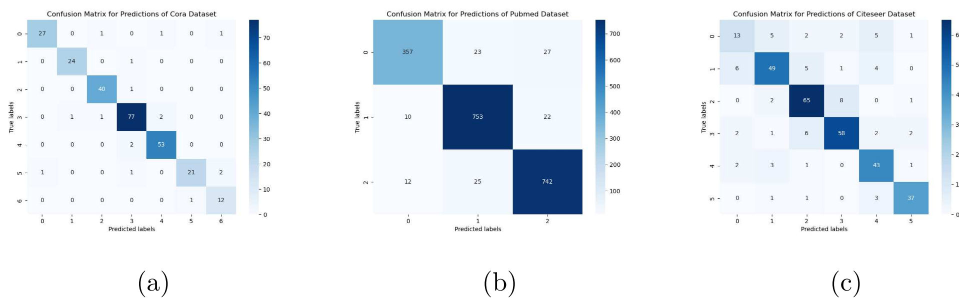


**Figure 5.2:** Confusion matrices for the three datasets, (a) Cora, (b) Pubmed, and (c) Citeseer. The diagonal elements represent the number of correct predictions for each class. The darker colors along the diagonal indicate high number of correct predictions. Off-diagonal elements represent misclassifications, the intensity of colors can show which classes are commonly confused with each other.

### 5.4.2 ROC Curves

A Receiver Operating Characteristic (ROC) curve is a graphical representation used to evaluate the performance of a binary classification. It plots the True Positive Rate (TPR) against the False Positive Rate (FPR). The ROC curve helps to understand the trade-off between sensitivity (recall) and specificity (precision). To apply to our multi-class classification task, we extend the binary ROC computation through One-vs-Rest (OvR) approach. In the OvR approach, the multi-class problem is broken down into multiple binary classification problems. For each class, we consider that class as the positive class and all other classes as the negative class.

Mathematically, for a given class $k$, the True Positive Rate $\mathrm{TPR}_k(t)$ and and False Positive Rate $\mathrm{FPR}_k(t)$ for a given threshold $t$ can be expressed as follows:

$$\mathrm{TPR}_k(t) = \frac{\mathrm{TP}_k(t)}{\mathrm{TP}_k(t) + \mathrm{FN}_k(t)} \tag{5.1}$$

$$\mathrm{FPR}_k(t) = \frac{\mathrm{FP}_k(t)}{\mathrm{FP}_k(t) + \mathrm{TN}_k(t)} \tag{5.2}$$

The ROC curves for the the three datasets are depicted in Figure 5.3. The ROC curve for the Cora dataset indicates exceptional performance by the classification model, particularly with six out of seven classes achieving a perfect AUC of 1.00. This near-perfect performance suggests that the model has an excellent capability to distinguish between the different classes in the Cora dataset. The high AUC values reflect the model's high accuracy and reliability in classifying these data points correctly.

The ROC curve for the Pubmed dataset also shows strong model performance across three classes. The high AUC scores indicate that the model is very effective at distinguishing between the classes in the Pubmed dataset. The slightly lower AUC for Class 0 and Class 2 compared to Class 1 suggests that the model is almost equally robust across all classes, with minimal variance in performance.

For the Citeseer dataset, the AUC value for the ROC curves indicate that the model performs reasonably well across all classes, with Class 0 having the lowest performance. Compared to the near-perfect scores of the Cora and Pubmed datasets, the Citeseer dataset shows slightly lower AUC values. Despite this, the high AUC values for most classes still indicate that the model performs reasonably well, suggesting a solid capability to distinguish between different classes.

Overall, the model demonstrates outstanding performance in the Cora and Pubmed datasets, with near-perfect AUC values indicating high accuracy and reliability in classification tasks. While the Citeseer dataset shows a slightly more varied performance, the model still maintains a strong capability to distinguish between classes. These ROC curves collectively underscore the model's robustness and effectiveness across different datasets, highlighting areas of excellence as well as opportunities for further enhancement.

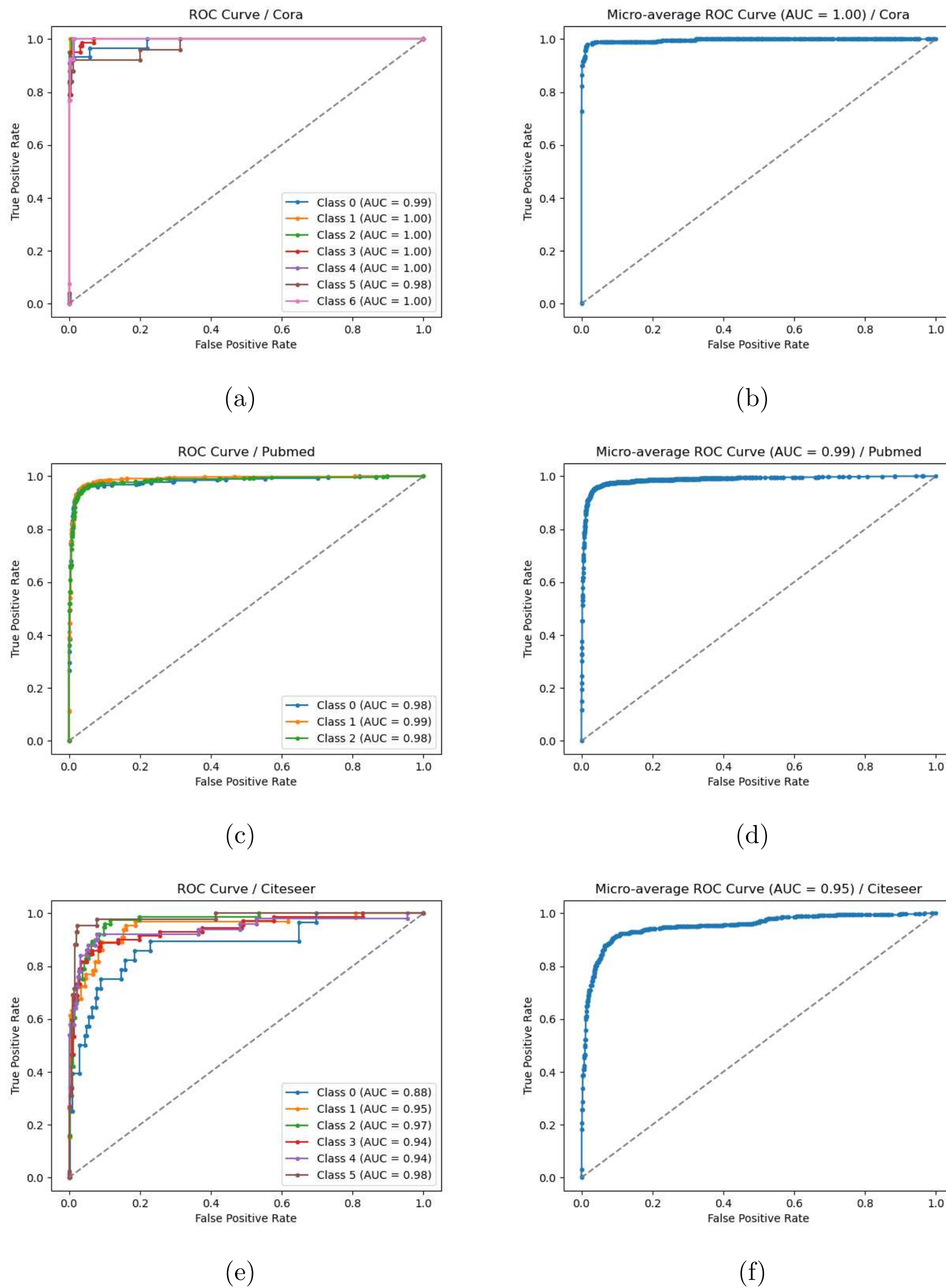


**Figure 5.3:** The ROC curves depicted for the three datasets. In (a), (c), and (e) individual barbarized ROC curves are presented for each class, while (b), (d), and (f) showcase the Micro-average ROC curve. The area under the ROC curve indicates the model's ability to discriminate between positive and negative classes, and a curve closer to the top-left corner indicates a better performing model. Both visualizations highlight a notable sensitivity rate, indicating that the model effectively distinguishes between positive and negative instances across various classes, contributing to its robust performance in classification tasks.

### 5.4.3 Node Embedding Visualizations.

t-SNE (t-Distributed Stochastic Neighbor Embedding) is a dimensionality reduction technique designed to maintain the local structure of data points. It works by calculating a similarity probability for each pair of high-dimensional points using a Gaussian distribution centered on each point. In the lower-dimensional space (typically 2D or 3D), t-SNE defines a similar probability distribution using a Student's t-distribution, which has heavier tails than the Gaussian. The algorithm then minimizes the difference between these high-dimensional and low-dimensional distributions using the Kullback-Leibler divergence as the cost function. This process ensures that points close to each other in the high-dimensional space remain close in the lower-dimensional space. Gradient descent is employed to iteratively adjust the positions of the points in the low-dimensional space to minimize this cost function.

The t-SNE visualizations of node embeddings for three datasets, both before and after training, are shown in Figure 5.4. Overall, these visualizations demonstrate the model's ability to distinguish different classes by forming clusters based on similarities. In the Citeseer embedding visualization, the well-separated clusters for classes 2 and 5 corroborate the high precision observed in the confusion matrix. In the Cora embedding visualization, some overlap between classes 5 and 6 suggests similarities between these classes. In the Pubmed embedding visualization, the classes are generally well-distinguished, with minor overlaps in the central part of the plot.

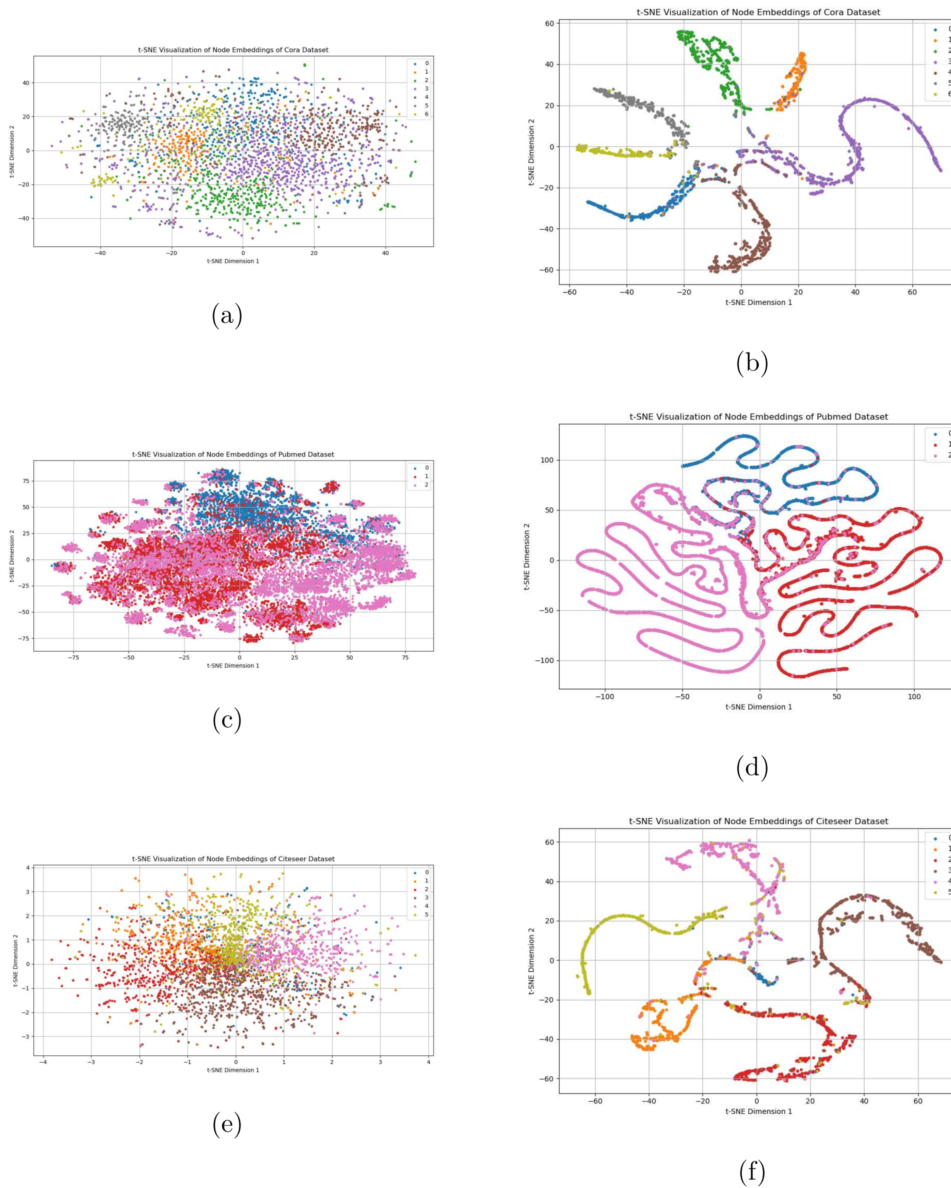


**Figure 5.4:** Visualization of Node Embeddings across three datasets: Cora, Pubmed, and Citeseer. In the left column, (a), (c), and (e) display the initial (pre-training) node features. Correspondingly, (b), (d), and (f) on the right column represent the output embeddings generated by the trained model. Class distinctions are illustrated through varied coloring in each plot. The closeness of the nodes suggest they have similar properties while the separation between the groups of clusters indicate dissimilarities.

## 5.5 Discussion

Based on the results of our experiments, the proposed approach, HLGNN, appears to show promising effectiveness as a neural network model for processing graph-structured data in node classification tasks. It shows competitive performance compared to baseline models such as GCN, GAT, and GraphConv across the tested datasets, outperforming the baseline models in two out of the three datasets.

### 5.5.1 Addressing Over-smoothing

Our model aims to address the issue of over-smoothing in a structural manner. Instead of relying on deep convolutional layers, which are often associated with over-smoothing, our model employs a hierarchical architecture with multiple layers, each designed to learn independently. We believe this design allows the model to extract more complex representations, potentially mitigating over-smoothing issues. In this architecture, levels of graph hierarchies are intended to be learned independently, which we hypothesize could result in more robust representations. The bottom-up and top-down fusions for inter-level message passing are intended to enable controlled, meaningful information flow between graph levels. This approach aims to allow HLGNN to increase the number of parameters while maintaining a shallow network structure, potentially leveraging the third dimension—levels of line graphs.

### 5.5.2 Advantages of Line Graph Transformation

The line graph transformation balances the original graph and facilitates long-range interactions between distant nodes. It integrates edge-based representations and highlights specific patterns that might be useful. As discussed in Chapter 1, line graphs can shorten paths in graphs and emphasize patterns like stars and circles during iterations. Generally, line graph transformation reinforces connectivity and creates a more nuanced degree distribution within graphs (refer to Theorem 1.4.2). These attributes are highly desirable in the context of graph neural networks, where the goal is to avoid bias towards high-degree nodes and maintain long-range interactions.

### 5.5.3 Computational Complexity

One disadvantage of the model is its potential high computational complexity, which can limit its application to larger, well-connected graphs. Although the transformation itself is linear in complexity and can be computed pre-training, learning representations of increasingly large and connected graph levels can become unsustainable for very large graphs.

### 5.5.4 Interpretability and Robustness

The training plots demonstrate the model's fast convergence and stabilized, consistent improvement over epochs across all datasets, confirming the model's ability to learn the datasets effectively while the ROC curves and confusion matrices display the

model's classification performance, highlighting its robustness across all classes in the datasets. Moreover, the t-SNE representations showcase the model's learned representations compared to initial representations, implying the model's ability to interpret the dataset in a meaningful way.

### 5.5.5 Versatility and Future Work

The architecture of the model is versatile and can be integrated with various GNN layers, not limited to GCN. Due to time and resource constraints, we focused on GCN, a proven effective GNN mechanism. The model can also be applied to other graph-related tasks such as graph classification, community detection, and link prediction. It is particularly promising for graph classification tasks, as line graph transformation can highlight special patterns. Additionally, edge attributes, if available, can be incorporated into the neural network architecture instead of learning edge embeddings for line graphs from scratch. Further exploration of the non-backtracking operator and other desirable operators for the transformation could reveal additional structural properties or reduce computational complexity.

# Conclusion

In this thesis, we proposed a novel approach to addressing the over-smoothing problem in Graph Neural Networks (GNNs). Over-smoothing is one of the major issues in GNNs as it limits the development of deeper, more expressive models. Over-smoothing happens often because of how the neural message-passing is operated in GNN model, where after several iterations, the nodes of the graph lose their diverse representations. This is particularly an issue when implementing a deep GNN model, as it limit a model's capacity to learn.

Among the solutions that address this problem, the hierarchical GNN models have shown to be effective in addressing this issue. Hierarchical models generate alternative representations of the original graph, aiming to capture higher-order representations, establish long-range interactions between distance nodes, and address over-smoothing issue, by creating other ways of information flow through hierarchical message-passing. Our main objective was to address this issue by proposing a novel hierarchical framework, leveraging hierarchical GNNs and line graphs. Line graph is a graph transformation technique, that highlights edge properties, and provides an alternative representation.

Inspired by hierarchical GNNs and line graph theory, we designed and implemented HLGNN, a hierarchical GNN model that constructs a hierarchy by iteratively creating a series of line graphs from the given graph. We also introduced message-passing mechanisms to maintain the flow of information within the hierarchical graphs. To enable low-cost inter-level message transmission, we utilized incidence matrices. Our implementation incorporates GraphConv, a widely used GNN convolution model, along with various regularization and optimization techniques.

By employing a hierarchical structure, HLGNN aims to capture complex relationships within the graph, potentially leading to faster convergence. The line graph transformation is intended to enhance connectivity within the graph and facilitate long-range interactions. Additionally, the design attempts to address the over-smoothing issue by regulating the flow of information through bottom-up and top-down fusions.

Our results suggest that HLGNN exhibits promising performance compared to baseline models across most tested datasets, indicating its potential as a GNN model. Training trends indicate the model's tendency for fast convergence and consistent improvement throughout the training phase. Analysis using ROC curves

and confusion matrices further highlights the model's robustness and its ability to generalize effectively. Moreover, the model processes datasets in an interpretable manner, as evidenced by visualization of embeddings.

Our approach can be further investigated for other tasks such as graph classification, community detection, and link prediction. It can be particularly useful in graph classification tasks, as it enriches the graph's structure while proposing balanced alternatives. Moreover, the proposed framework's versatility allows for the integration of any convolutional layer, enhancing its applicability across different scenarios. The transformation itself can be integrated with various operators (both backtracking or non-backtracking) to generate graphs tailored to the specific needs of a problem.

Overall, our proposed model opens a new pathway towards improving graph data processing. The line graph, a fascinating yet simple transformation, has remained under-explored in the realm of machine learning and graph neural networks. Our work showcases a glimpse of the potential these transformations hold, and it remains a promising area for future exploration.

# Appendix A

# Algorithms

## A.1 HLGNN Forward pass

**Algorithm 1** HLGNN Forward Pass

```
1: Input: features, num_hidden_layers
2: lg_hidden ← Make a list of hidden layer tensors, each for one graph in the hierarchy
3: for layer in num_hidden_layers do
4:     lg_hidden ← HLGNN(lg_hidden)
5: end for
6: z ← self.nn.Linear(concat(lg_hidden))
7: z ← F.relu(z)
8: return z
```

## A.2 HLGNN Fusion Operations

**Algorithm 2** HLGNN Fusion Operations

```
1: function FORWARD(graph, curr_h, prev_h, next_h, curr_inc, prev_inc_T,
   update)
2:     graph ← dgl.add_self_loop(graph)
3:     conv_layer ← GraphConv(g, curr_h)
4:     bottom_up_layer ← GraphConv(g, prev_inc_T * prev_h)
5:     top_down_layer ← GraphConv(g, curr_inc * next_h)
6:     result ← update(conv_layer, bottom_up_layer, top_down_layer)
7:     result ← self.layer_norm(result)
8:     result ← F.relu(result)
9:     result ← self.dropout(result)
10:    return result
11: end function
```

## A.3 HLGNN Training Algorithm

**Algorithm 3** HLGNN Training Algorithm

```
1:  Input: model, features, labels, train_mask, epochs, patience, lr, weight_decay
2:  Initialize optimizer with Adam algorithm using model parameters
3:  learning rate (lr), and weight decay (weight_decay)
4:  no_improv ← 0
5:  for epoch ← 1 to epochs do
6:      if no_improv > patience then
7:          break
8:      end if
9:      Set model to training mode
10:     Zero the gradients of the optimizer
11:     model_out, logits ← model(features)
12:     loss ← CrossEntropyLoss(logits[train_mask], labels[train_mask])
13:     Perform back-propagation on the loss
14:     Update the model parameters using the optimizer
15: end for
```

# Bibliography


[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. arXiv:1607.06450 [cs, stat]. July 2016. DOI: 10.48550/arXiv.1607.06450.

[2] Lowell W. Beineke. *Line graphs and line digraphs*. eng. Developments in Mathematics ; Volume 68. Cham, Switzerland: Springer, 2021. ISBN: 978-3-030-81386-4.

[3] Vincent D. Blondel et al. "Fast unfolding of communities in large networks". In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (Oct. 2008). arXiv:0803.0476 [cond-mat, physics:physics], P10008. ISSN: 1742-5468. DOI: 10.1088/1742-5468/2008/10/P10008.

[4] Xavier Bresson and Thomas Laurent. *Residual Gated Graph ConvNets*. arXiv:1711.07553 [cs, stat]. Apr. 2018. DOI: 10.48550/arXiv.1711.07553.

[5] Lei Cai et al. *Line Graph Neural Networks for Link Prediction*. arXiv:2010.10046 [cs]. Oct. 2020. DOI: 10.48550/arXiv.2010.10046.

[6] Gary Chartrand and M. James Stewart. "The connectivity of line-graphs". In: *Mathematische Annalen* 182.3 (Sept. 1969), pp. 170–174. ISSN: 1432-1807. DOI: 10.1007/BF01350320.

[7] Zhengdao Chen, Xiang Li, and Joan Bruna. *Supervised Community Detection with Line Graph Neural Networks*. arXiv:1705.08415 [stat]. Aug. 2020. DOI: 10.48550/arXiv.1705.08415.

[8] Frederik Diehl. *Edge Contraction Pooling for Graph Neural Networks*. arXiv:1905.10990 [cs, stat]. May 2019. DOI: 10.48550/arXiv.1905.10990.

[9] Alex Fout et al. "Protein Interface Prediction using Graph Convolutional Networks". In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017.

[10] William L. Hamilton. *Graph Representation Learning*. en. Synthesis Lectures on Artificial Intelligence and Machine Learning. Cham: Springer International Publishing, 2020. DOI: 10.1007/978-3-031-01588-5.

[11] William L. Hamilton, Rex Ying, and Jure Leskovec. *Inductive Representation Learning on Large Graphs*. arXiv:1706.02216 [cs, stat]. Sept. 2018. DOI: 10.48550/arXiv.1706.02216.

[12] David K. Hammond, Pierre Vandergheynst, and Rémi Gribonval. *Wavelets on Graphs via Spectral Graph Theory*. arXiv:0912.3848 [cs, math]. Dec. 2009. DOI: 10.48550/arXiv.0912.3848.

[13] F. Harary. *Graph Theory*. Reading, MA: Addison-Wesley, 1969.

[14] Geoffrey E. Hinton et al. *Improving neural networks by preventing co-adaptation of feature detectors*. arXiv:1207.0580 [cs]. July 2012. DOI: 10.48550/arXiv.1207.0580.

[15] Jingjia Huang et al. *AttPool: Towards Hierarchical Feature Representation in Graph Convolutional Networks via Attention Mechanism*. Pages: 6488. Oct. 2019. DOI: 10.1109/ICCV.2019.00658.

[16] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. arXiv:1609.02907 [cs, stat]. Feb. 2017. DOI: 10.48550/arXiv.1609.02907.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012.

[18] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[19] Yaguang Li et al. *Diffusion Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting*. arXiv:1707.01926 [cs, stat]. Feb. 2018. DOI: 10.48550/arXiv.1707.01926.

[20] Ryan Liu et al. *Hierarchical Graph Neural Networks for Particle Track Reconstruction*. arXiv:2303.01640 [hep-ex]. Mar. 2023. DOI: 10.48550/arXiv.2303.01640.

[21] Chengqiang Lu et al. *Molecular Property Prediction: A Multilevel Quantum Interactions Modeling Perspective*. arXiv:1906.11081 [physics]. June 2019. DOI: 10.48550/arXiv.1906.11081.

[22] Luca Pasa, Nicolò Navarin, and Alessandro Sperduti. "Compact graph neural network models for node classification". In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. SAC '22. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 592–599. ISBN: 978-1-4503-8713-2. DOI: 10.1145/3477314.3507100.

[23] Ladislav Rampášek and Guy Wolf. *Hierarchical graph neural nets can capture long-range interactions*. arXiv:2107.07432 [cs, stat]. Aug. 2021. DOI: 10.48550/arXiv.2107.07432.

[24] Franco Scarselli et al. "The Graph Neural Network Model". In: *IEEE Transactions on Neural Networks* 20.1 (Jan. 2009). Conference Name: IEEE Transactions on Neural Networks, pp. 61–80. ISSN: 1941-0093. DOI: 10.1109/TNN.2008.2005605.

[25] A. Sperduti and A. Starita. “Supervised neural networks for the classification of structures”. In: *IEEE Transactions on Neural Networks* 8.3 (May 1997). Conference Name: IEEE Transactions on Neural Networks, pp. 714–735. ISSN: 1941-0093. DOI: 10.1109/72.572108.

[26] Chang Su, Min Chen, and Xianzhong Xie. “Graph Convolutional Matrix Completion via Relation Reconstruction”. In: *Proceedings of the 2021 10th International Conference on Software and Computer Applications*. ICSCA ’21. New York, NY, USA: Association for Computing Machinery, July 2021, pp. 51–56. ISBN: 978-1-4503-8882-5. DOI: 10.1145/3457784.3457792.

[27] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. *Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks*. 2015. arXiv: 1503.00075 [cs.CL].

[28] Petar Veličković et al. *Graph Attention Networks*. arXiv:1710.10903 [cs, stat]. Feb. 2018. DOI: 10.48550/arXiv.1710.10903.

[29] Zhenqin Wu et al. *MoleculeNet: A Benchmark for Molecular Machine Learning*. arXiv:1703.00564 [physics, stat]. Oct. 2018. DOI: 10.48550/arXiv.1703.00564.

[30] Keyulu Xu et al. *How Powerful are Graph Neural Networks?* arXiv:1810.00826 [cs, stat]. Feb. 2019. DOI: 10.48550/arXiv.1810.00826.

[31] Rex Ying et al. “Graph Convolutional Neural Networks for Web-Scale Recommender Systems”. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. arXiv:1806.01973 [cs, stat]. July 2018, pp. 974–983. DOI: 10.1145/3219819.3219890.

[32] Rex Ying et al. *Hierarchical Graph Representation Learning with Differentiable Pooling*. arXiv:1806.08804 [cs, stat]. Feb. 2019. DOI: 10.48550/arXiv.1806.08804.

[33] Jiaxuan You et al. *GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models*. 2018. arXiv: 1802.08773 [cs.LG].

[34] Seongjun Yun et al. “Graph Transformer Networks”. In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019.

[35] Zhiqiang Zhong, Cheng-Te Li, and Jun Pang. “Hierarchical message-passing graph neural networks”. en. In: *Data Mining and Knowledge Discovery* 37.1 (Jan. 2023), pp. 381–408. ISSN: 1573-756X. DOI: 10.1007/s10618-022-00890-9.

[36] Jie Zhou et al. *Graph Neural Networks: A Review of Methods and Applications*. arXiv:1812.08434 [cs, stat]. Oct. 2021. DOI: 10.48550/arXiv.1812.08434.