GRAPH NEURAL NETWORKS: A FEATURE AND STRUCTURE LEARNING APPROACH

A Dissertation

by

HONGYANG GAO

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2020

Major Subject: Computer Science

ABSTRACT

Deep neural networks (DNNs) have achieved great success on grid-like data such as images, but face tremendous challenges in learning from more generic data such as graphs. In convolutional neural networks (CNNs), for example, the trainable local filters enable the automatic extraction of high-level features. The computation with filters requires a fixed number of ordered units in the receptive fields. However, the number of neighboring units is neither fixed nor are they ordered in generic graphs, thereby hindering the applications of deep learning operations such as convolution, attention, pooling, and unpooling. To address these limitations, we propose several deep learning methods on graph data in this dissertation.

Graph deep learning methods can be categorized into graph feature learning and graph structure learning. In the category of graph feature learning, we propose to learn graph features via learnable graph convolution operations, graph attention operations, and line graph structures. In learnable graph convolution operations, we propose the learnable graph convolutional layer (LGCL). LGCL automatically selects a fixed number of neighboring nodes for each feature based on value ranking in order to transform graph data into grid-like structures in 1-D format, thereby enabling the use of regular convolutional operations on generic graphs. In graph attention operations, we propose novel hard graph attention operator (hGAO) and channel-wise graph attention operator (cGAO). hGAO uses the hard attention mechanism by attending to only important nodes. Compared to GAO, hGAO improves performance and saves computational cost by only attending to important nodes. To further reduce the requirements on computational resources, we propose the cGAO that performs attention operations along channels. cGAO avoids the dependency on the adjacency matrix, leading to dramatic reductions in computational resource requirements. Beside using original graph structures, we investigate feature learning on auxiliary graph structures such as line graph. By using line graph structures, we propose a weighted line graph that corrects biases in line graphs by assigning normalized weights to edges. Based on our weighted line graphs, we develop a weighted line graph convolution layer that takes advantage of line graph structures for better feature learning. In particular, it performs message passing operations on both the original graph and its corresponding weighted line graph. To address efficiency issues in line graph neural networks, we propose to use an incidence matrix to accurately compute the adjacency matrix of the weighted line graph, leading to dramatic reductions in computational resource usage.

In the category of graph structure learning, we propose several deep learning methods to learn new graph structures. Given images are special cases of graphs with nodes lie on 2D lattices, graph embedding tasks have a natural correspondence with image pixel-wise prediction tasks such as segmentation. While encoder-decoder architectures like U-Nets have been successfully applied on many image pixel-wise prediction tasks, similar methods are lacking for graph data. This is due to the fact that pooling and up-sampling operations are not natural on graph data. To address these challenges, we propose novel graph pooling (gPool) and unpooling (gUnpool) operations in this work. The gPool layer adaptively selects some nodes to form a smaller graph based on their scalar projection values on a trainable projection vector. However, gPool uses global ranking methods to sample some of the important nodes, which is not able to incorporate graph topology information in computing ranking scores. To address this issue, we propose the topology-aware pooling (TAP) layer that uses attention operators to generate ranking scores for each node by attending each node to its neighboring nodes. The ranking scores are generated locally while the selection is performed globally, which enables the pooling operation to consider topology information. We further propose the gUnpool layer as the inverse operation of the gPool layer. The gUnpool layer restores the graph into its original structure using the position information of nodes selected in the corresponding gPool layer. Based on our proposed gPool and gUnpool layers, we develop an encoder-decoder model on graph, known as the graph U-Nets.

Our experimental results on node classification graph classification tasks using both real and simulated data demonstrate the effectiveness and efficiency of our methods.

DEDICATION

To my family

CONTRIBUTORS AND FUNDING SOURCES

Contributors

I would like to thank my advisor, Dr. Shuiwang Ji, for his trust, encouragement, and support. He sets a high research standard for me from the very beginning of my research career, which has been one of my strongest motive powers in the past four years. His invaluable suggestions and expertise in my research field greatly helped me to finish my doctoral degree program. It's my pleasure to have Dr. James Caverlee, Dr. Xiaoning Qian, and Dr. Zhangyang Wang on my dissertation committee. They gave me a lot of valuable and helpful suggestions for my research projects and dissertation.

I would like to thank all members in the Data Integration, Visualization, and Exploration (DIVE) Laboratory at TAMU for their helpful suggestions and collaborations. Last but not least, I appreciate my beloved wife. Without her continuous support and encouragement, I would not have finished my Ph.D study.

Funding Sources

This work was supported in part by National Science Foundation grants IIS-1633359, IIS-1908166, IIS-1908198, IIS-1908220, DBI-1922969, DBI-1661289, DBI-1641223, and Defense Advanced Research Projects Agency grant N66001-17-2-4031.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Deep learning methods [1] are becoming increasingly powerful in solving various challenging artificial intelligence tasks. These deep learning methods have demonstrated promising performance in many image-related applications, such as image classification [2], semantic segmentation [3], and object detection [4, 5]. A variety of deep learning models have been proposed to continuously set the performance records [2, 6, 7, 8]. In addition to images, deep learning methods have also been successfully applied to natural language processing tasks such as neural machine translation [9, 10, 11]. One common characteristic behind these tasks is that the data can be represented by grid-like structures. This enables the use of kernel-based operations such as convolution and pooling in the form of the same local filters scanning every position on the input. Unlike traditional hand-crafted filters, the local filters used in convolutional layers are trainable. The networks can automatically decide what kind of features to extract by learning the weights in these trainable filters, thereby avoiding hand-crafted feature extraction [12].

In many real-world applications, the data can be naturally represented as graphs, such as social, citation, and biological networks. Figure 1.1 provides an illustration of graph data. Many interesting discoveries can be made by analyzing these graph data, such as social network analysis [13]. An important task on graph data is node classification [14, 15], in which models make predictions for every node in a graph based on node features and graph topology. As mentioned above, deep learning methods, with the power of automatic feature extraction, have achieved great success on tasks with grid-like data, which can be considered as special cases of graph data. Therefore, applying deep learning methods such as convolution, attention, pooling, and unpooling on graph tasks is appealing. However, using regular convolutional operations on generic graphs faces two main challenges. These challenges are resulted from the fact that regular convolutions require the number of neighboring nodes for each node remains the same, and these neighboring nodes are ordered. In generic graphs, the numbers of neighboring nodes usually differ for different nodes in a graph. In addition, among the neighboring nodes of a node, there is no ranking information

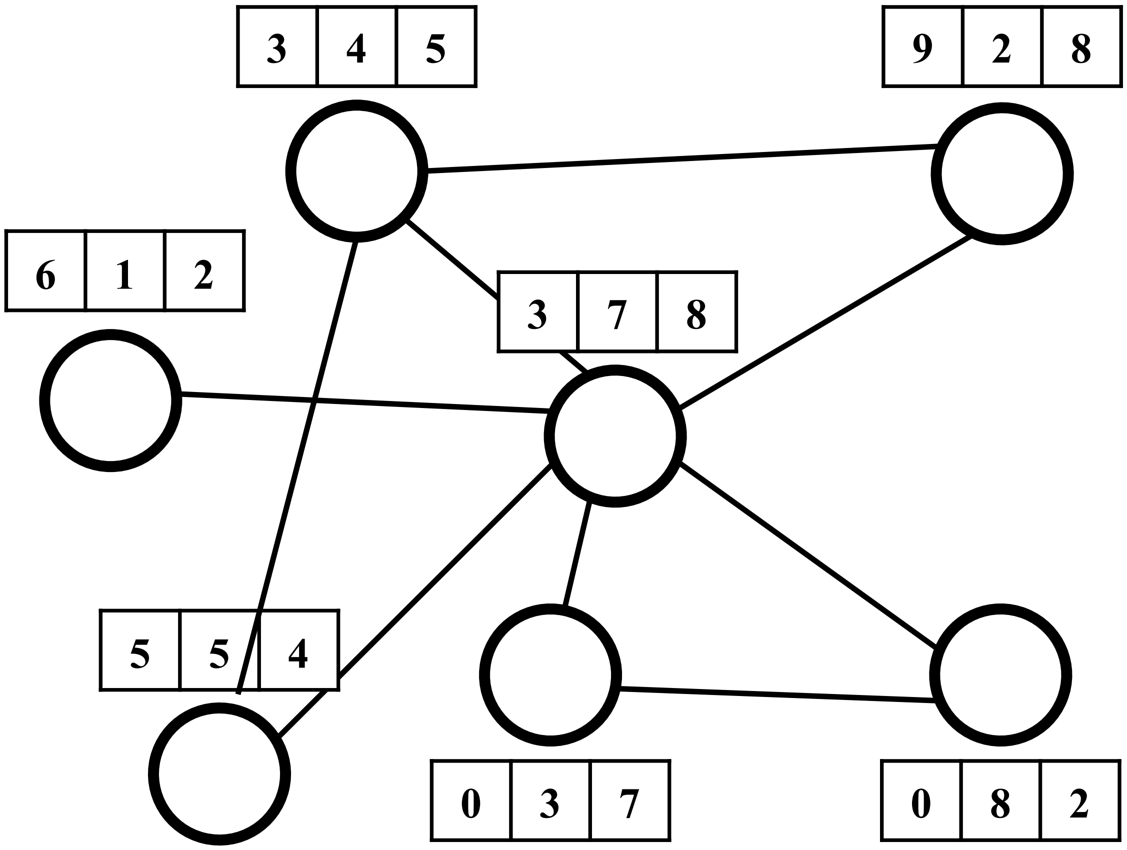

Figure 1.1: An illustration of graph data. There are 7 nodes in this graph and each node has 3 features. Each node in this graph may have a different number of neighboring nodes, and there is no relative order among them.

based on which we can order them to ensure the output is deterministic. Based on these limitations, regular deep learning methods can not be directly applied on graph data.

1.1 Dissertation Outline

To address the challenges we discussed in previous section, this dissertation proposes several graph deep learning methods that enable deep learning methods on graph data. In particular, we categorize graph deep learning methods into graph feature learning and graph structure learning. Graph feature learning methods learn node features by aggregating information from neighboring nodes. We introduce our graph feature learning methods in Section 2, 3 and 4. While graph structure learning methods learn new graph structures, which is covered in Section 5.

In Section 2, we introduce our proposed graph feature learning methods. First, we propose the learnable graph convolutional layer (LGCL) to enable the use of regular convolutional operations on graphs. Note that prior studies modified the original convolutional operations to fit them for graph data. In contrast, our LGCL transforms the graphs to enable the use of regular convolutions. Our models based on LGCL achieve better performance on both transductive learning and inductive node classification tasks, as demonstrated by our experimental results. Second, we observe another limitation of prior methods; that is, their training process takes the adjacency matrix of the whole graph as an input. This requires excessive memory and computational resources when the graph has a large amount of nodes, which is usually the case in real-world tasks. In order to overcome this limitation, we develop a sub-graph training method, which is a simple yet effective approach to allow the training of deep learning methods on large-scale graph data. The sub-graph training method can significantly reduce the amount of required memory and computational resources, with negligible loss in terms of model performance.

In Section 3, we propose novel hard graph attention operator (hGAO). hGAO performs attention operation by requiring each query node to only attend to part of neighboring nodes in graphs. By employing a trainable project vector $\boldsymbol{p}$, we compute a scalar projection value of each node in graph on $\boldsymbol{p}$. Based on these projection values, hGAO selects several important neighboring nodes to which the query node attends. By attending to the most important nodes, the responses of the query node are more accurate, thereby leading to better performance than methods based on soft attention. Compared to GAO, hGAO also saves computational cost by reducing the number of nodes to attend. GAO also suffers from the limitations of excessive requirements on computational resources, including computational cost and memory usage. hGAO improves the performance of attention operator by using hard attention mechanism. It still consumes large amount of memory, which is critical when learning from large graphs. To overcome this limitation, we propose a novel channel-wise graph attention operator (cGAO). cGAO performs attention operation from the perspective of channels. The response of each channel is computed by attending to all channels. Given that the number of channels is far smaller than the number of nodes, cGAO can significantly save computational resources. Another advantage of cGAO over GAO and hGAO is that it does not rely on the adjacency matrix. In both GAO and hGAO, the adjacency matrix is used to identify neighboring nodes for attention operators. In cGAO, features within the same node communicate with each other, but features in different nodes do not. cHAO does not need the adjacency matrix to identify nodes connectivity. By avoiding dependency on the adjacency matrix, cGAO achieves better computational efficiency than GAO and hGAO. Based on our proposed hGAO and cGAO, we develop deep attention networks for graph embedding learning. Experimental results on graph classification and node classification tasks demonstrate that our proposed deep models with the new operators achieve consistently better performance. Comparison results also indicates that hGAO achieves significantly better performance than GAOs on both node and graph embedding tasks. Efficiency comparison shows that our cGAO leads to dramatic savings in computational resources, making them applicable to large graphs.

In Section 4, we investigate graph feature learning using unique graph structures such as line graph structures. In particular, we propose to construct a weighted line graph that can correct biases in encoded topology information of line graphs. To this end, we assign each edge in a line graph a normalized weight such that each node in the line graph has a weighted degree of 2. In this weighted line graph, the dynamics of node features are the same as those in its original graph. Based on our weighted line graph, we propose a weighted line graph convolution layer (WLGCL) that performs a message passing operation on both original graph structures and weighted line graph structures. To address inefficiency issues existing in graph neural networks that use line graph structures, we further propose to implement our WLGCL via an incidence matrix, which can dramatically reduce the usage of computational resources. Based on our WLGCL, we build a family of weighted line graph convolutional networks (WLGCNs). We evaluate our methods on graph classification tasks and show that WLGCNs consistently outperform previous state-of-the-art models. Experiments on simulated data demonstrate the efficiency advantage of our implementation.

In Section 5, we introduce our graph structure methods. In particular, we propose novel graph pooling (gPool) and unpooling (gUnpool) operations. Based on these two operations, we propose

U-Net-like architectures for graph data. The gPool operation samples some nodes to form a smaller graph based on their scalar projection values on a trainable projection vector. As an inverse operation of gPool, we propose a corresponding graph unpooling (gUnpool) operation, which restores the graph to its original structure with the help of locations of nodes selected in the corresponding gPool layer. Based on the gPool and gUnpool layers, we develop graph U-Nets, which allow high-level feature encoding and decoding for network embedding. We use gPool and propose hConv layers in FCN-like graph convolutional networks for text modeling. Since graphs are extracted from texts, we maintain the node orders as in the original texts. We propose the hConv layer that combines GCN and regular convolutional operations to enable automatic high-level feature extraction. Based on our gPool and hConv layers, we propose four networks for the task of text categorization. Our results show that the model based on gPool and hConv layers achieves new state-of-the-art performance compared to CNN-based models. gPool layers involve negligible number of parameters but bring significant performance boosts, demonstrating its contributions to model performance.

In Section 6, we introduce a topology-aware pooling method that can address the limitations of our gPool layer. Our gPool layer used global ranking methods to sample some of the important nodes, but most of them are not able to incorporate graph topology information in computing ranking scores. In this work, we propose the topology-aware pooling (TAP) layer that uses attention operators to generate ranking scores for each node by attending each node to its neighboring nodes. The ranking scores are generated locally while the selection is performed globally, which enables the pooling operation to consider topology information. To encourage better graph connectivity in the sampled graph, we propose to add a graph connectivity term to the computation of ranking scores in the TAP layer. Based on our TAP layer, we develop a network on graph data, known as the topology-aware pooling network. Experimental results on graph classification tasks demonstrate that our methods achieve consistently better performance than previous models.

1.2 Contributions

The main contributions of this dissertation are summarized as below:

- We propose the learnable graph convolutional layer (LGCL), which can address the challenges on graphs and enable effective convolutional operations. LGCL automatically selects a fixed number of neighboring nodes for each feature based on value ranking in order to transform graph data into grid-like structures in 1-D format, thereby enabling the use of regular convolutional operations on generic graphs.

- We propose novel hard graph attention operator (hGAO) and channel-wise graph attention operator (cGAO). hGAO uses the hard attention mechanism by attending to only important nodes. Compared to regular graph attention operator, hGAO improves performance and saves computational cost by only attending to important nodes. To further reduce the requirements on computational resources, we propose the cGAO that performs attention operations along channels. cGAO avoids the dependency on the adjacency matrix, leading to dramatic reductions in computational resource requirements.

- We utilize line graph structures to enhance feature learning in graphs. In particular, we construct a weighted line graph that can correct the bias in original line graph structures. Based our weighted line graph structures, we propose the weighted line graph layer that leverages the advantage of the weighted line graph structure. A practical challenge faced by graph neural networks on line graphs is that they consume excessive computational resources, especially on dense graphs. To address this limitation, we propose to use the incidence matrix to implement the WLGCL, which can dramatically save the computational resources.

- We propose novel graph pooling (gPool) and unpooling (gUnpool) operations. Based on these two operations, we propose U-Net-like architectures for graph data. The gPool operation samples some nodes to form a smaller graph based on their scalar projection values on a trainable projection vector. As an inverse operation of gPool, we propose a corresponding graph unpooling (gUnpool) operation, which restores the graph to its original structure with the help of locations of nodes selected in the corresponding gPool layer. Based on the gPool and gUnpool layers, we develop graph U-Nets, which allow high-level feature encoding and decoding for network embedding.

- We apply our proposed graph deep learning methods on text data to overcome long-range dependency problem. One limitation of graph deep learning methods when used on graph-based text representation tasks is that, graph deep learning methods do not consider the order information of nodes in graph. To address this limitation, we propose the hybrid convolutional (hConv) layer that combines GCN and regular convolutional operations. The hConv layer is capable of increasing receptive fields quickly and computing features automatically. Based on the proposed gPool and hConv layers, we develop new deep networks for text categorization tasks. Our experimental results show that the networks based on gPool and hConv layers achieves new state-of-the-art performance as compared to baseline methods.

2. GRAPH FEATURE LEARNING VIA LEARNABLE CONVOLUTION OPERATIONS

In this section, we focus on how to learn high-level features without changing graph structures. In particular, we propose the learnable graph convolutional layer to enable learnable kernels in spatial dimension[1].

2.1 Introduction

Deep learning methods are becoming increasingly powerful in solving various challenging artificial intelligence tasks. Among these deep learning methods, convolutional neural networks (CNNs) [1] have demonstrated promising performance in many image-related applications, such as image classification [2], semantic segmentation [3], and object detection [4, 5]. A variety of CNN models have been proposed to continuously set the performance records [2, 6, 7, 8]. In addition to images, CNNs have also been successfully applied to natural language processing tasks such as neural machine translation [9, 10, 11]. One common characteristic behind these tasks is that the data can be represented by grid-like structures. This enables the use of convolutional operations in the form of the same local filters scanning every position on the input. Unlike traditional hand-crafted filters, the local filters used in convolutional layers are trainable. The networks can automatically decide what kind of features to extract by learning the weights in these trainable filters, thereby avoiding hand-crafted feature extraction [12].

In many real-world applications, the data can be naturally represented as graphs, such as social, citation, and biological networks. Figure 1.1 provides an illustration of graph data. Many interesting discoveries can be made by analyzing these graph data, such as social network analysis [13]. An important task on graph data is node classification [14, 15], in which models make predictions for every node in a graph based on node features and graph topology. As mentioned above, CNNs, with the power of automatic feature extraction, have achieved great success on tasks with grid-like

[1]Reprinted with permission from "Large-scale learnable graph convolutional networks." by Hongyang Gao, Zhengyang Wang, and Shuiwang Ji, 2018, *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, vol. 1, pp. 1416-1424, Copyright 2018 by ACM.

data, which can be considered as special cases of graph data. Therefore, applying deep learning models, especially CNNs, on graph tasks is appealing. However, using regular convolutional operations on generic graphs faces two main challenges. These challenges are resulted from the fact that regular convolutions require the number of neighboring nodes for each node remains the same, and these neighboring nodes are ordered. In generic graphs, the numbers of neighboring nodes usually differ for different nodes in a graph. In addition, among the neighboring nodes of a node, there is no ranking information based on which we can order them to ensure the output is deterministic. In this work, we analyze the necessity of having a fixed number of ordered neighboring nodes in regular convolutional operations and propose elegant solutions to address these challenges.

Several recent studies tried to apply convolutional operations on generic graphs. Graph convolutional networks (GCNs) [14] proposed to use a convolution-like operation to aggregate features of all adjacent nodes for each node, followed by a linear transformation to generate new a feature representation for a given node. Specifically, all feature vectors in the neighborhood, including the feature vector of the central node itself, are summed up, weighted by non-trainable weights depending on the number of neighbors. This can be thought of as a convolution-like operation which, however, is intrinsically different from the regular convolutional operation in two aspects. First, it does not use the same local filter to scan every node; that is, nodes that have different numbers of adjacent nodes have filters of different sizes and weights. Second, the weights in the filters are the same for all neighboring nodes in the receptive field as they are determined by the number of neighbors. Consequently, the weights are not learned. Graph attention networks (GATs) [15] employed the attention mechanism [16] to obtain different and trainable weights for adjacent nodes by measuring the correlation between their feature vectors and that of the central node. Yet graph attention operation still differs from the regular convolution which learns weights in local filters directly. Moreover, the attention mechanism requires extra computation in terms of pairs of feature vectors, resulting in excessive memory and computational resource requirements in practice.

In this work, we make two major contributions to applying CNNs on generic graph data. First, we propose the learnable graph convolutional layer (LGCL) to enable the use of regular convolutional operations on graphs. Note that prior studies modified the original convolutional operations to fit them for graph data. In contrast, our LGCL transforms the graphs to enable the use of regular convolutions. Our models based on LGCL achieve better performance on both transductive learning and inductive node classification tasks, as demonstrated by our experimental results. Second, we observe another limitation of prior methods; that is, their training process takes the adjacency matrix of the whole graph as an input. This requires excessive memory and computational resources when the graph has a large amount of nodes, which is usually the case in real-world tasks. In order to overcome this limitation, we develop a sub-graph training method, which is a simple yet effective approach to allow the training of deep learning methods on large-scale graph data. The sub-graph training method can significantly reduce the amount of required memory and computational resources, with negligible loss in terms of model performance.

2.2 Related Work

A few recent studies have tried to apply convolutional operations on graph data. Graph convolutional networks (GCNs) were introduced in [14] and achieved the state-of-art performance on several node classification tasks. The authors defined and used a convolution-like operation termed the spectral graph convolution. This enables CNNs to directly operate on graphs. Basically, each layer in GCNs updates the feature vector representation of each node in the graph by considering the features of neighboring nodes. To be specific, the layer-wise forward-propagation operation of GCNs can be expressed as

$$X_{l+1} = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}X_l W_l), \tag{2.1}$$

where X_l and X_{l+1} are the input and output matrices of layer l, respectively. For both matrices, the numbers of rows are the same, corresponding to the number of nodes in the graph, while the numbers of columns can be different, depending on the dimensions of the input and output feature space. In Eq (2.1), $\hat{A} = A + I$ is used to aggregate feature vectors of adjacent nodes, where A is the adjacency matrix of the graph, and I is the identity matrix. Also, $\hat{A}$ is used, instead of A, because the layers need to add self-loop connections to make sure that the old feature vector of the node itself is taken into consideration when updating the representation of a node. $\hat{D}$ is the diagonal node degree matrix, which is used to normalize $\hat{A}$ so that the scale of feature vectors after aggregation remains the same. W_l is a trainable weight matrix and represents a linear transformation that changes the dimension of feature space. Therefore, the dimension of W^l depends on how many features that each node in the input and output have, *i.e.,* the number of columns in X_l and X_{l+1}, respectively. $\sigma(\cdot)$ denotes an activation function like ReLU.

We analyze the convolution-like operation, which is the feature aggregation step through pre-multiplying X_l by $\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}$. Consider a node with a feature vector corresponding to the i-th row in X_l. The aggregation output, controlled by the i-th row in $\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}$, is a weighted sum of the feature vectors of all of its adjacent nodes, including the node itself. We can see that the operation is equivalent to having a local filter for each node, whose receptive field consists of the node itself and all its neighboring nodes. As is common that nodes in a generic graph have different numbers of adjacent nodes, the receptive field size varies, resulting in different local filters. This is a key difference from the regular convolutional operation, where the same local filter is applied to scan each position in grid-like data. Moreover, while using local filters of different sizes for graph data seems reasonable, it is worth noting that there is no trainable parameter in $\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}$. In addition, each adjacent node receives the same weight in the weighted sum, which makes it a simple average. While CNNs achieve the power of automatic feature extraction by learning the weights in local filters, this non-trainable aggregation operation in GCNs limits the capability of CNNs on generic graph data.

From this perspective, graph attention networks (GATs) [15] tried to enable learnable weights when aggregating neighboring feature vectors by employing the attention mechanism [16, 17]. Like GCNs, each node still has a local filter with a receptive field covering the node itself and all of its adjacent nodes. When performing the weighted sum of feature vectors, each neighbor receives a different weight by measuring the correlation between its feature vector and that of the central node. Mathematically, for a node i and one of its adjacent nodes j, the correlation measurement process between layer l and $l+1$ is given by

$$\begin{aligned} e_l^{i,j} &= a_l(W_l x_l^i, W_l x_l^j) \\ \alpha_l^{i,j} &= \text{softmax}(e_l^{i,j}), \end{aligned} \tag{2.2}$$

where x_l^i and x_l^j represent the corresponding feature vectors, *i.e.,* the i-th and j-th row in X_l, respectively, W_l is a shared linear transformation and a_l represents a single-layer feed-forward neural network, $\alpha_l^{i,j}$ is the weight for node j in the feature aggregation operation of node i. Although in this way, GATs provide different and trainable weights to different adjacent nodes, the learning process differs from that of regular CNNs where weights in local filters are learned directly. Also, the attention mechanism requires extra computation between a node and all of its adjacent nodes, which will cause memory and computational resource problems in practice.

Unlike these prior models, which modified the regular convolutional operations to fit them for generic graph data, we instead propose to transform graphs into grid-like data to enable the use of CNNs directly. This idea was previously explored in [18]. However, the transformation in [18] is implemented in the preprocessing process while our method includes the transformation in the networks. Additionally, we introduce a sub-graph training method in this work, which is a simple yet effective approach to allow large-scale training.

2.3 Methods

In this section, we introduce the learnable graph convolutional layer (LGCL) and the sub-graph training strategy on generic graph data. Based on these developments, we propose the large-scale learnable graph convolutional networks (LGCNs).

2.3.1 Challenges of Applying Convolutional Operations on Graph Data

In order to apply regular convolutional operations on graphs, we need to overcome two main challenges that are caused by two major differences between generic graphs and grid-like data. First, the number of adjacent nodes usually varies for different nodes in a generic graph. Second, we cannot order the neighboring nodes in generic graphs, since there is no ranking information among them. For example, in a social network, each person in the network can be seen as a node and the edges represent friendships between people. Obviously, the number of adjacent nodes differs for each node since people can have different numbers of friends. Meanwhile, it is hard to order these friends without additional information for ranking.

Note that grid-like data can be viewed as a special type of graph data, where each node has a fixed number of ordered neighbors. As convolutional operations apply directly on grid-like data such as images, we analyze why the two characteristics mentioned above are necessary to performing regular convolutions. To see the need of having a fixed number of adjacent nodes with ranking information, consider a convolutional filter with a size of 3×3 scanning an image. We think of the image as a special graph by thinking of each pixel as a node. During the scan, the computation involves a central node with $3 \times 3 - 1 = 8$ adjacent nodes each time. These 8 nodes become neighbors of the central node by having edges connecting them in the special graph. Meanwhile, we can order these neighboring nodes by their relative positions with respect to the central node. This is crucial to convolutional operations since the correspondence between weights in the filter and nodes in the graph must be maintained during the scan. For instance, in the example above, the upper left weight in the 3×3 filter should always be multiplied with the neighboring node at the top left of the central node. Without such ranking information, the outputs of convolution operations are no longer deterministic. We can see from the above discussions that it is challenging to directly apply regular convolutional operations on generic graph data. To address these two challenges, we propose an approach to transform generic graphs into grid-like data.

2.3.2 Learnable Graph Convolutional Layers

To enable the use of regular convolutional operations on generic graphs, we propose the learnable graph convolutional layer (LGCL). Following the notations defined in Section 2.2, the layer-

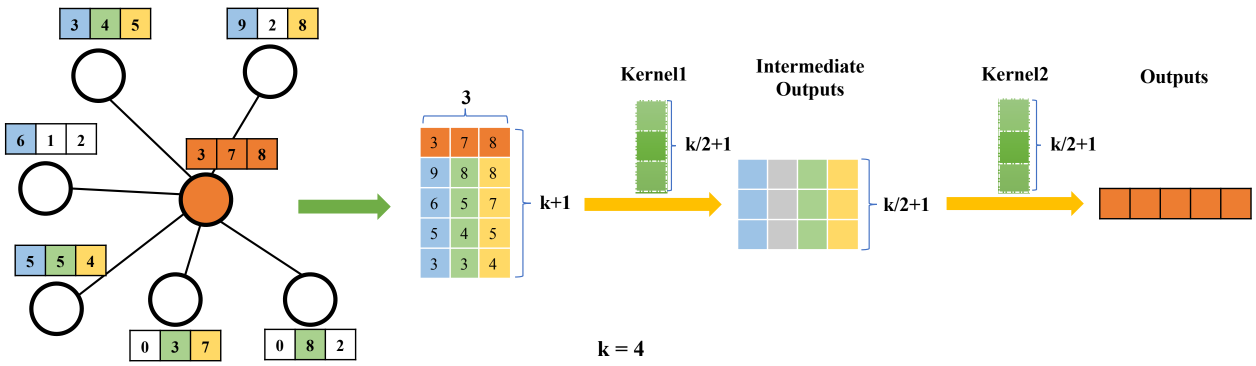

Figure 2.1: An illustration of a learnable graph convolutional layer (LGCL). We consider a node with 6 adjacent nodes. Each node has three features, represented by a 3-component feature vector. This layer selects $k = 4$ nodes in the neighborhood and employs a 1-D CNN to produce a new vector representation of five features for the central node, color-coded in orange. The left part describes the process of selecting the k-largest values for each feature from neighboring nodes. It can be seen from the graph that there are 6 neighbors. Since $k = 4$, for each feature, four largest values are selected from the neighborhood based on the ranking. For example, the results of this selection process for the first feature is {9, 6, 5, 3} out of {9, 6, 5, 3, 0, 0}. By repeating the same process for the other two features, we obtain $(k + 1)$ 3-component feature vectors, including that of the orange node itself. Concatenating them gives a 1-D data of grid-like structure, which has $(k + 1)$ positions and 3 channels. Afterwards, a 1-D CNN is applied to generate the final feature vector. Specifically, we use two convolutional layers with a kernel size of $(k/2 + 1)$ and without padding. The numbers of output channels are 4 and 5, respectively. In practice, the 1-D CNN can be any CNN model, as long as the final output is a vector, serving as the new feature representation of the central node.

wise propagation rule of LGCL is formulated as

$$\begin{aligned} \tilde{X}_l &= g(X_l, A, k), \\ X_{l+1} &= c(\tilde{X}_l), \end{aligned} \tag{2.3}$$

where A is the adjacency matrix, $g(\cdot)$ is an operation that performs the k-largest node selection to transform generic graphs to data of grid-like structures, and $c(\cdot)$ denotes a regular 1-D CNN that aggregates neighboring information and outputs a new feature vector for each node. We discuss $g(\cdot)$ and $c(\cdot)$ separately below.

k-largest Node Selection. We propose a novel method known as the k-largest node selection to achieve the transformation from graphs to grid-like data, where k is a hyper-parameter of LGCL.

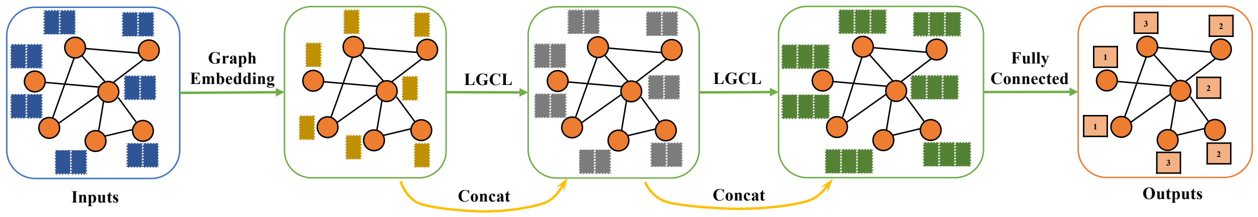

Figure 2.2: An illustration of the proposed learnable graph convolutional network (LGCN). In this example, the nodes in the input have two features. The input feature vectors are transformed into low-dimensional representations using a graph embedding layer. After that, we stack two LGCL layers with skip concatenation connections to refine the feature vectors of each node. Finally, a fully-connected layer is used for node classification. There are three different classes in this example.

After this operation, each node aggregates neighboring information and is represented in a 1-D grid-like format with $(k + 1)$ positions. The transformed data is then fed into a 1-D CNN to generate the updated feature vector.

Suppose $X_l \in \mathbb{R}^{N\times C}$ with row vectors $x_l^1, x_l^2, \cdots, x_l^N$, representing a graph of N nodes where each node has C features. We are given the adjacency matrix $A \in \mathbb{N}^{N\times N}$ and a fixed k. Now consider a specific node i whose feature vector is x_l^i and it has n neighboring nodes. Through a simple look-up operation in A, we can obtain the indices of these adjacent nodes, say $i_1, i_2, \cdots, i_n$. Concatenating the corresponding feature vectors $x_l^{i_1}, x_l^{i_2}, \cdots, x_l^{i_n}$ outputs a matrix $M_l^i \in \mathbb{R}^{n\times C}$. Without the loss of generalization, assume that $n \geq k$. If $n < k$ in practice, we can pad M_l^i using columns of zeros. The k-largest node selection is conducted on M_l^i; that is, for each column, we rank the n values and select k-largest values. This gives us a $k \times C$ output matrix. As the columns in M_l^i represent features, the operation is equivalent to selecting k-largest values for each feature. By inserting x_l^i in the first row, the output becomes $\tilde{M}_l^i \in \mathbb{R}^{(k+1)\times C}$. This is illustrated in the left part of Figure 2.1. By repeating this process for each node, $g(\cdot)$ transforms X_l to $\tilde{X}_l \in \mathbb{R}^{N\times(k+1)\times C}$.

Note that $\tilde{X}_l$ can be viewed as a 1-D grid-like structure by considering N, $(k+1)$, and C as the batch size, the spatial size, and the number of channels, respectively. Therefore, the k-largest node selection function $g(\cdot)$ successfully achieves the transformation from generic graphs to grid-like data. The operation makes use of the natural ranking information among real numbers and forces each node to have a fixed number of ordered neighbors.

1-D Convolutional Neural Networks. As discussed in Section 2.3.1, regular convolutional operations can be directly applied on grid-like data. As $\tilde{X}_l \in \mathbb{R}^{N\times(k+1)\times C}$ is 1-D, we employ a 1-D CNN model $c(\cdot)$. The basic functionality of LGCL is to aggregate adjacent information and update the feature vector for each node. Consequently, it requires $X_{l+1} \in \mathbb{R}^{N\times D}$, where D is the dimension of the updated feature space. The 1-D CNN $c(\cdot)$ should take $\tilde{X}_l \in \mathbb{R}^{N\times(k+1)\times C}$ as input and output a matrix of dimension $N \times D$, or equivalently, $N \times 1 \times D$. Basically, $c(\cdot)$ reduces the spatial size from $(k+1)$ to 1.

Note that N is considered as the batch size, which is not related to the design of $c(\cdot)$. As a result, we focus on only one data sample, *i.e.,* one node in the graph. Taking the example above, for node i, the transformed output is $\tilde{M}_l^i \in \mathbb{R}^{(k+1)\times C}$, which serves as the input to $c(\cdot)$. Due to the fact that any regular convolutional operation with a filter size larger than one and no padding reduces the spatial size, the simplest $c(\cdot)$ has only one convolutional layer with a filter size of $(k+1)$ and no padding. The numbers of input and output channels are C and D, respectively. Meanwhile, any multi-layer CNN can be employed, provided its final output has the dimension of $1 \times D$. The right part of Figure 2.1 illustrates an example of a two-layer CNN. Again, applying $c(\cdot)$ for all the N nodes outputs $X_{l+1} \in \mathbb{R}^{N\times D}$. In summary, our LGCL transforms generic graphs to grid-like data using the proposed k-largest node selection and applies a regular 1-D CNN to perform feature aggregation and refine the feature vector for each node.

2.3.3 Learnable Graph Convolutional Networks

It is known that deeper networks usually yield better performance. However, prior deep models on graphs like GCNs only have two layers. While they suffer from performance loss when going deeper [14], our LGCL enables a deeper design, resulting in the learnable graph convolutional networks (LGCNs) for graph node classification. We build LGCNs based on the architecture of densely connected convolutional networks (DCNNs) [19, 8], which achieved state-of-the-art performance in the ImageNet classification challenge [2].

In LGCNs, we first apply a graph embedding layer to produce low-dimensional representations

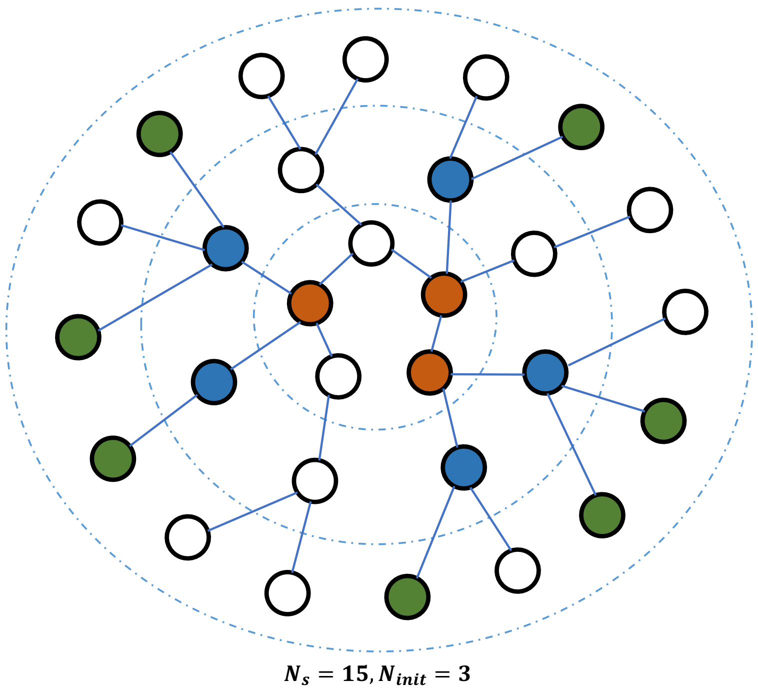

Figure 2.3: An example of the sub-graph selection process. We start with $N_{init} = 3$ randomly sampled nodes and obtain a sub-graph of $N_s = 15$ nodes. In the first iteration, we use BFS to find all the first-order neighboring nodes of the 3 initial nodes (orange), excluding themselves. Among these nodes, we randomly select $N_m = 5$ nodes (blue). In the next iteration, we select $N_m = 7$ nodes from neighbors of the blue nodes, excluding previously selected nodes. Note that N_m changes for the two iterations, which is a flexible choice in practice. After two iterations, we have selected $3 + 5 + 7 = 15$ nodes and obtained a required sub-graph. These nodes, along with the corresponding adjacency matrix, will form the input to the LGCN in a training iteration.

of nodes, since the original inputs are usually very high-dimensional feature vectors in some graph dataset, such as the Cora [20]. The graph embedding layer is essentially a linear transformation in the first layer expressed as

$$X_1 = X_0 W_0, \tag{2.4}$$

where $X_0 \in \mathbb{R}^{N \times C_0}$ represents the high-dimensional input and $W_0 \in \mathbb{R}^{C_0 \times C_1}$ changes the dimension of feature space from C_0 to C_1. As a result, $X_1 \in \mathbb{R}^{N \times C_1}$ and $C_1 < C_0$. Alternatively, a GCN layer can be used for graph embedding. As illustrated in Section 2.2, the number of training parameters in a GCN layer is equal to that of a regular graph embedding layer.

After the graph embedding layer, we stack multiple LGCLs, according to the complexity of the

Algorithm 1 Sub-Graph Selection Algorithm

Input: Adjacency matrix A, Number of nodes N, Sub-graph size N_s, Initial number of nodes N_{init}, Maximum number of nodes expanded per iteration N_m

Output: A set of nodes S as a sub-graph

```
1: S = $\phi$
2: initNodes = sample $N_{init}$ nodes from $N$ nodes.
3: S = S $\cup$ initNodes
4: newAddNodes = initNodes
5: while size(S) < $N_s$ and size(newAddNodes) $\neq$ 0 do
6:     candidateNodes = BFS(newAddNodes, A)
                          $\triangleright$ Obtain first-order neighboring nodes of newAddNodes
7:     newAddNodes = candidateNodes $\setminus$ S
8:     if size(newAddNodes) > $N_m$ then
9:         newAddNodes = sample $N_m$ nodes from newAddNodes
10:    end if
11:    if size(newAddNodes) + size(S) > $N_s$ then
12:        $N_r$ = $N_s$ - size(S)
13:        newAddNodes = sample $N_r$ nodes from newAddNodes
14:    end if
15:    S = S $\cup$ newAddNodes
16: end while
17: return S
```

graph data. As each LGCL only aggregates information from first-order neighboring nodes, *i.e.,* direct neighboring nodes, stacked LGCLs can collect information from a larger set of nodes, which is commonly done in regular CNNs. In order to promote the model performance and facilitate the training process, we apply skip connections to concatenate the inputs and outputs of LGCLs. Finally, a fully-connected layer is used before the softmax function for final predictions.

Following the design principle of LGCNs, k and the number of stacked LGCLs are the most important hyper-parameters. The average degree of nodes in the graph can be a good reference for selecting k. Meanwhile, the number of LGCLs should depend on the complexity of tasks, such as the number of classes, the number of nodes in a graph, etc. More complicated tasks require deeper models.

Table 2.1: Summary of datasets used in our experiments [21, 22]. The Cora, Citeseer, and Pubmed datasets are used for transductive learning experiments, while the PPI dataset is for inductive learning experiments. The degree attribute listed is the average node degree of each dataset, which helps the selection of the hyper-parameter k in LGCLs.

Dataset	**Nodes**	**Features**	**Classes**	**Train**	**Valid**	**Test**	**Degree**
Cora	2708	1433	7	140	500	1000	4
Citeseer	3327	3703	6	120	500	1000	5
Pubmed	19717	500	3	60	500	1000	6
PPI	56944	50	121	44906	6514	5524	31

2.3.4 Sub-Graph Training on Large-Scale Data

Most prior deep models on graphs suffer from another limitation. In particular, during training the inputs are the feature vectors of all the nodes along with the adjacency matrix of the whole graph, whose sizes become large for large graph data. These prior models work properly on small-scale graphs. However, for large-scale graphs, those methods usually result in excessive memory and computational resource requirements, which limit the practical applications of these models.

Similar problems also happen for deep neural networks on other types of data, such as grid-like data. For example, deep models on image segmentation usually use randomly cropped patches when dealing with large images. Motivated by this strategy, we intend to randomly "crop" a graph to obtain smaller graphs for training. However, while a rectangular patch of an image naturally maintains neighboring information among pixels, how to handle irregular connections between nodes in a graph remains challenging.

In this work, we propose a sub-graph selection algorithm to address the memory and computational resource problems on large-scale graph data, as shown in Algorithm 1. Given a graph, we first sample some initial nodes. Staring from them, we use the Breadth-First-Search (BFS) algorithm to expand adjacent nodes into the sub-graph iteratively. With multiple iterations, high-order neighboring nodes of the initial nodes are included. Note that we use a single parameter N_m in Algorithm 1 for simplicity. In practice, we can set N_m to different values for each iteration. Figure 2.3 provides an example of the sub-graph selection process.

With such randomly "cropped" sub-graphs, we are able to train deep models on large-scale graphs. In addition, we can take advantage of the mini-batch training strategy to accelerate the learning process. In each training iteration, we can use the proposed sub-graph selection algorithm to sample several sub-graphs and put them in a mini-batch. The corresponding feature vectors and adjacency matrices form the inputs to the networks.

2.4 Experimental Studies

In this section, we evaluate our proposed large-scale learnable graph convolutional networks (LGCNs) on node classification tasks under both transductive and inductive learning settings. In addition to comparisons with prior state-of-the-art models, some performance studies are performed to investigate how to choose hyper-parameters. Experiments are also conducted to analyze the training strategy based on the proposed sub-graph selection algorithm. Experimental results show that LGCNs yield improved performance, and the sub-graph training is much more efficient than whole-graph training. Our code is publicly available[2].

2.4.1 Datasets

In our experiments, we focus on node classification tasks under both transductive and inductive learning settings.

Transduction Learning. Under the transductive setting, the unlabeled testing data are accessible and available during training. To be specific, for node classification, only a part of nodes in the graph are labeled. The testing nodes, which are also in the same graph, are accessible during training, including their features and connections, except for the labels. This means the training process knows about the graph structure that contains testing nodes. We use three standard benchmark datasets for transductive learning experiments; those are the Cora, Citeseer, and Pubmed [20], as summarized in Table 2.1. These three datasets are citation networks with nodes and edges representing documents and citations, respectively. The feature vector of each node corresponds to a bag-of-word representation for a document. For these three datasets, we employ the same experi-

[2]`https://github.com/divelab/lgcn/`

mental settings as those in GCN [14]. For each class, 20 nodes are used for training, 500 nodes are used for validation and 1,000 nodes are used for testing.

Inductive Learning. For inductive learning, the testing data are not available during training, which means the training process does not learn about the structure of test graphs. In inductive learning tasks, we usually have different training, validation, and testing graphs. During training, the model only use the training graphs without access to validation and testing graphs. We use the protein-protein interaction (PPI) dataset [22], which contains 20 graphs for training, 2 graphs for validation, and 2 graphs for testing. Since the graphs for validation and testing are separate, the training process does not use them. There are 2,372 nodes in each graph on average. Each node has 50 features including positional, motif genes and signatures. Each node has multiple labels from 121 classes.

2.4.2 Experimental Setup

We describe the experimental setup under both transductive and inductive learning settings.

Transduction Learning. In transductive learning tasks, we employ the proposed LGCN models as illustrated in Figure 2.2. Since transductive learning datasets employ high-dimensional bag-of-word representations as feature vectors of nodes, the inputs go through a graph embedding layer to reduce the dimension. Here, we use a GCN layer as the graph embedding layer. The dimension of the embedding output is 32. Then we apply LGCLs, each of which uses $k = 8$ and produces 8-component feature vectors. For the Cora, Citeseer, and Pubmed, we stack 2, 1, and 1 LGCLs, respectively. We use concatenation in skip connections. Finally, a fully-connected layer is used as a classifier to make predictions. Before the fully-connected layer, we perform a simple sum to aggregate feature vectors of adjacent nodes. Dropout [23] is applied on both input feature vectors and adjacency matrices in each layer with rates of 0.16 and 0.999, respectively. All LGCN models in transductive learning tasks use the sub-graph training strategy. The sub-graph size is set to $2,000$.

Inductive Learning. For inductive learning, the same LGCN model as above is used except for some hyper-parameters. For the graph embedding layer, the dimension of output feature vectors

Table 2.2: Results of transductive learning experiments in terms of node classification accuracies on the Cora, Citeseer, and Pubmed datasets. LGCN$_{sub}$ denotes the LGCN model using the sub-graph training strategy.

Models	**Cora**	**Citeseer**	**Pubmed**
DeepWalk [26]	67.2%	43.2%	65.3%
Planetoid [21]	75.7%	64.7%	77.2%
Chebyshev [27]	81.2%	69.8%	74.4%
GCN [14]	81.5%	70.3%	79.0%
$\mathbf{LGCN}_{sub}\mathbf{(Ours)}$	**83.3 ± 0.5%**	**73.0 ± 0.6%**	**79.5 ± 0.2%**

Table 2.3: Results of inductive learning experiments in terms of micro-averaged F1 scores on the PPI dataset.

Models	**PPI**
GraphSAGE-GCN [28]	0.500
GraphSAGE-mean [28]	0.598
GraphSAGE-pool [28]	0.600
GraphSAGE-LSTM [28]	0.612
$\mathbf{LGCN}_{sub}\mathbf{(Ours)}$	**0.772 ± 0.002**

is 128. We stack two LGCLs with $k = 64$. We also employ the sub-graph training strategy, with sub-graph initial node size equal to 500 and 200. Dropout with a rate of 0.9 is applied in each layer.

For both transductive and inductive learning LGCN models, the following configurations are shared. For all layers, only the identity activation function is used, which means no nonlinearity is involved in the networks. In order to avoid over-fitting, the L_2 regularization with $\lambda = 0.0005$ is applied. For training, the Adam optimizer [24] with a learning rate of 0.1 is used. Weights in LGCNs are initialized by the Glorot initialization [25]. We employ the early stopping strategy based on the validation accuracy and train 1,000 epochs at most.

2.4.3 Analysis of Results

The experimental results are summarized in Tables 2.2 and 2.3 for transductive and learning settings, respectively.

Transduction Learning. For transductive learning experiments, we report node classification accuracies as in [14]. Table 2.2 provides the comparisons with other graph models. According to the results, our LGCN models achieve better performance over the current state-of-the-art GCNs by a margin of 1.8%, 2.7%, and 0.6% on the Cora, Citeseer, and Pubmed datasets, respectively.

Inductive Learning. For inductive learning experiments, we report micro-averaged F1 scores like [28]. From table 2.3, we can observe that our LGCN model outperforms GraphSAGE-LSTM by a margin of 16%. Without observing the structure of test graphs in training, the LGCN model still achieves good generalization.

The results above show that the proposed LGCN models on generic graphs consistently yield new state-of-the-art performance in node classification tasks on different datasets. These results demonstrate the effectiveness of applying regular convolutional operations on transformed graph data. In addition, the proposed transformation approach through the k-largest node selection is shown to be effective.

2.4.4 LGCL versus GCN Layers

It may be argued that our LGCN models employ a deeper network architecture than GCNs, which could explain the improved performance. However, the performance of GCNs is reported to decrease when going deeper by stacking more layers. In addition, we conduct another experiment by replacing all LGCLs in LGCN models by GCN layers, denoted as LGCN$_{sub}$-GCN model. All the other settings remain the same in order to ensure the fairness of the comparisons. Table 2.4 provides the comparison results between LGCN$_{sub}$ and LGCN$_{sub}$-GCN. The results show that LGCN$_{sub}$ has better performance than LGCN$_{sub}$-GCN, which indicates that the LGCL is more effective than the GCN layer.

2.4.5 Sub-Graph versus Whole-Graph Training

For the experiments above, we use the sub-graph training strategy to learn the LGCN models, which aims at saving memory and training time. However, since the sub-graph selection algorithm samples some nodes as a sub-graph from the whole graph, it means that the models trained in this

Table 2.4: Results of transductive learning experiments for comparing the LGCN$_{sub}$ and GCN layers on the Cora, Citeseer, and Pubmed datasets. Using the network architecture of LGCN$_{sub}$, we replace LGCLs by GCN layers, resulting in the LGCN$_{sub}$-GCN model.

Models	**Cora**	**Citeseer**	**Pubmed**
LGCN$_{sub}$-GCN	$82.2 \pm 0.5\%$	$71.1 \pm 0.5\%$	$79.0 \pm 0.2\%$
LGCN$_{sub}$(Ours)	**$83.3 \pm 0.5\%$**	**$73.0 \pm 0.6\%$**	**$79.5 \pm 0.2\%$**

way do not learn about the structure of whole graph during training. Meanwhile, in transductive learning tasks, the information of testing nodes may be ignored, which raises the risk of performance loss. To address this concern, we perform experiments on transductive learning tasks to compare the sub-graph training strategy with the previous whole-graph training strategy. Through the experiments, we show the advantages of using the sub-graph training strategy, with negligible loss in terms of model performance.

For the sub-graph selection process described in Algorithm 1, the algorithm starts with some initial nodes that are randomly selected. In transductive learning tasks, we sample initial nodes only from the nodes with training labels to make sure that training can be conducted. To be specific, we sample 140, 120, and 60 initial nodes when selecting the sub-graph for the Cora, Citeseer, and Pubmed datasets, respectively. For each iteration in the sub-graph selection algorithm, we do not set N_m to limit the number of nodes expanded into the sub-graph. The maximum number of nodes in the sub-graph is set to 2,000 for all the three datasets, which is an feasible size for our GPUs in hand.

For comparison, we perform experiments using the same LGCN models, but train them using the same whole-graph training strategy as GCNs, which means the inputs are representations of the entire graph. We denote such models as LGCN$_{whole}$, compared to LGCN$_{sub}$ with the sub-graph training strategy. The comparing results of these two models with GCNs are provided in Table 2.5. The number of nodes reported represents how many nodes are used for one iteration of training. The time reported here is the training time for running 100 epochs using a single TITAN Xp GPU.

It can be seen that the actual numbers of nodes in the training sub-graph for the Cora, Citeseer,

Table 2.5: Results of transductive learning experiments for comparing the sub-graph training and whole-graph training strategies on the Cora, Citeseer, and Pubmed datasets. For comparison, we conduct experiments on LGCNs that employ the same whole-graph training strategy as GCNs, denoted as LGCN$_{whole}$.

		Cora	**Citeseer**	**Pubmed**
GCN	# Nodes	2708	3327	19717
	Accuracy	81.5%	70.3%	79.0%
	Time	**7s**	4s	38s
LGCN$_{whole}$	# Nodes	2708	3327	19717
	Accuracy	$83.8 \pm 0.5\%$	$73.0 \pm 0.6\%$	$79.5 \pm 0.2\%$
	Time	58s	30s	1080s
LGCN$_{sub}$	# Nodes	644	442	354
	Accuracy	$83.3 \pm 0.5\%$	$73.0 \pm 0.6\%$	$79.5 \pm 0.2\%$
	Time	14s	**3.6s**	**2.6s**

and Pubmed datasets are 644, 442, and 354, respectively, which are far smaller than the maximum sub-graph size of 2,000. This indicates that the nodes in the Cora, Citeseer, and Pubmed datasets are sparsely connected. Specifically, starting from several initial nodes with training labels, only a small set of nodes will be selected by expanding neighboring nodes to form connected sub-graphs. While these datasets are usually considered as a single large graph, the whole graph is actually composed of several separate sub-graphs that have no connection to each other. The sub-graph training strategy takes advantage of this fact and makes efficient use of the nodes with training labels. Since only the initial nodes have training labels and all their connectivity information is included in the selected sub-graphs, the amount of information loss in the sub-graph training is minimized, resulting in negligible performance loss. This is demonstrated by comparing the node classification accuracies of LGCN$_{sub}$ and LGCN$_{whole}$. According to the results, LGCN$_{sub}$ models only have a subtle performance loss of 0.5% on the Cora dataset, while yielding the same performance on the Citeseer and Pubmed datasets, as compared to the LGCN$_{whole}$ models.

After investigating the risk of performance loss, we point out the great advantages of the sub-graph training strategy in terms of training speed. By using the sub-graph training, LGCN$_{sub}$ models take a sub-graph of fewer nodes as inputs in contrast to the whole graph, which is expected

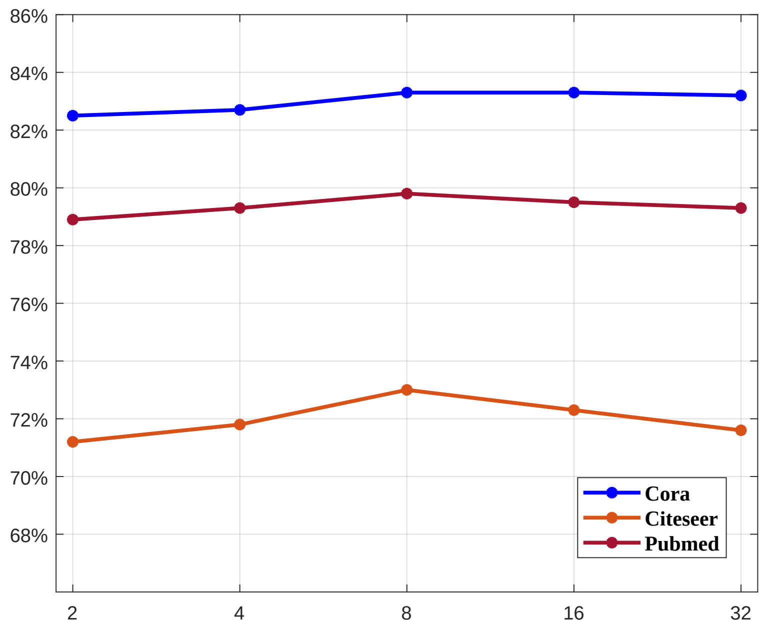

Figure 2.4: Results of using different values of hyper-parameter k in LGCN models. On the Cora, Citeseer, and Pubmed datasets, we employ the same experimental setups described in Section 2.4.2. We adjust the value of k in LGCN$_{sub}$ and report node classification accuracies in this figure. It can be seen that $k = 8$ achieves the best performance for these datasets.

to greatly promote the training efficiency. It can be seen from the results in Table 2.5 that the improvement is outstanding. Although GCNs require simpler computation, its running time is much longer than that of LGCN models on large-scale graph datasets like the Pubmed. Powerful deep models are usually used on large-scale data, which makes the sub-graph training strategy useful in practice. The sub-graph training strategy enables using more complex layers such as the proposed LGCLs without the concern of long training time. As a result, our large-scale LGCNs with the sub-graph training strategy are not only effective but also very efficient.

2.4.6 Performance Study of k in LGCL

As described in Section 2.3.3, the average degree of nodes in graph can be helpful when choosing the hyper-parameter k in LGCNs. In this part, we conduct experiments to show how different values of k affect the performance of LGCN models. We vary the value of k in LGCLs and observe the node classification accuracies on the Cora, Citeseer, and Pubmed datasets. The values of k are selected from 2, 4, 8, 16, and 32, which cover a reasonable range of integer values.

Figure 2.4 plots the performance change of LGCN models under different values of k. As demonstrated in the figure, the LGCN models achieve the best performance on all the three datasets when choosing $k = 8$. In the Cora, Citeseer, and Pubmed datasets, the average node degrees are 4, 5, and 6, respectively. This indicates that the best k is usually a bit larger than the average node degree in the dataset. When k is too large, the performance of LGCN models decreases. A possible explanation is that if k is much larger than the average node degree in the graph, too many zero padding is used in the k-largest node selection process, which compromises the performance of the following 1-D CNN models. For the inductive learning task on the PPI dataset, we also explore different values of k. The best performance is given by $k = 64$ while the average node degree is 31. This is consistent with our results above.

3. GRAPH FEATURE LEARNING VIA GRAPH ATTENTION OPERATORS

In the previous section, we discussed how to learn high-level features without changing graph structures. In this section, we move forward to discuss graph deep learning methods that learn new graph structures[1].

3.1 Introduction

Deep attention networks are becoming increasingly powerful in solving challenging tasks in various fields, including natural language processing [17, 10, 29], and computer vision [30, 31, 32]. Compared to convolution layers and recurrent neural layers like LSTM [33, 34], attention operators are able to capture long-range dependencies and relationships among input elements, thereby boosting performance [29, 35]. In addition to images and texts, attention operators are also applied on graphs [15]. In graph attention operators (GAOs), each node in a graph attend to all neighboring nodes, including itself. By employing attention mechanism, GAOs enable learnable weights for neighboring feature vectors when aggregating information from neighbors. However, a practical challenge of using GAOs on graph data is that they consume excessive computational resources, including computational cost and memory usage. The time and space complexities of GAOs are both quadratic to the number of nodes in graphs. At the same time, GAOs belong to the family of soft attention [36], instead of hard attention [31]. It has been shown that hard attention usually achieves better performance than soft attention, since hard attention only attends to important features [37, 31, 38].

In this work, we propose novel hard graph attention operator (hGAO). hGAO performs attention operation by requiring each query node to only attend to part of neighboring nodes in graphs. By employing a trainable project vector $\boldsymbol{p}$, we compute a scalar projection value of each node in graph on $\boldsymbol{p}$. Based on these projection values, hGAO selects several important neighboring nodes

[1]Reprinted with permission from "Graph representation learning via hard and channel-wise attention networks." by Hongyang Gao and Shuiwang Ji, 2019, *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, vol. 1, pp. 741-749, Copyright 2019 by ACM.

to which the query node attends. By attending to the most important nodes, the responses of the query node are more accurate, thereby leading to better performance than methods based on soft attention. Compared to GAO, hGAO also saves computational cost by reducing the number of nodes to attend.

GAO also suffers from the limitations of excessive requirements on computational resources, including computational cost and memory usage. hGAO improves the performance of attention operator by using hard attention mechanism. It still consumes large amount of memory, which is critical when learning from large graphs. To overcome this limitation, we propose a novel channel-wise graph attention operator (cGAO). cGAO performs attention operation from the perspective of channels. The response of each channel is computed by attending to all channels. Given that the number of channels is far smaller than the number of nodes, cGAO can significantly save computational resources. Another advantage of cGAO over GAO and hGAO is that it does not rely on the adjacency matrix. In both GAO and hGAO, the adjacency matrix is used to identify neighboring nodes for attention operators. In cGAO, features within the same node communicate with each other, but features in different nodes do not. cHAO does not need the adjacency matrix to identify nodes connectivity. By avoiding dependency on the adjacency matrix, cGAO achieves better computational efficiency than GAO and hGAO.

Based on our proposed hGAO and cGAO, we develop deep attention networks for graph embedding learning. Experimental results on graph classification and node classification tasks demonstrate that our proposed deep models with the new operators achieve consistently better performance. Comparison results also indicates that hGAO achieves significantly better performance than GAOs on both node and graph embedding tasks. Efficiency comparison shows that our cGAO leads to dramatic savings in computational resources, making them applicable to large graphs.

3.2 Background and Related Work

In this section, we describe the attention operator and related hard attention and graph attention operators.

3.2.1 Attention Operator

In an attention operator, it takes three matrices as input. These matrices are a query matrix $\boldsymbol{Q} = [\mathbf{q}_1, \mathbf{q}_2, \cdots, \mathbf{q}_m] \in \mathbb{R}^{d \times m}$ with each $\mathbf{q}_i \in \mathbb{R}^d$, a key matrix $\boldsymbol{K} = [\mathbf{k}_1, \mathbf{k}_2, \cdots, \mathbf{k}_n] \in \mathbb{R}^{d \times n}$ with each $\mathbf{k}_i \in \mathbb{R}^d$, and a value matrix $\boldsymbol{V} = [\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_n] \in \mathbb{R}^{p \times n}$ with each $\mathbf{v}_i \in \mathbb{R}^p$. For each query vector $\boldsymbol{q}_i$, the attention operator produces its response by attending it to every key vector in $\boldsymbol{K}$. The results are used to compute a weighted sum of all value vectors in $\boldsymbol{V}$, leading to the output of the attention operator. The layer-wise forward-propagation operation of attn($\boldsymbol{Q}$, $\boldsymbol{K}$, $\boldsymbol{V}$) is defined as

$$\begin{aligned} \boldsymbol{E} &= \boldsymbol{K}^T\boldsymbol{Q} && \in\mathbb{R}^{n\times m}, \\ \boldsymbol{O} &= \boldsymbol{V}\text{softmax}(\boldsymbol{E}) && \in\mathbb{R}^{p\times m}, \end{aligned} \tag{3.1}$$

where softmax$(\cdot)$ is a column-wise softmax operator.

The coefficient matrix $\boldsymbol{E}$ is calculated by the matrix multiplication between $\boldsymbol{K}^T$ and $\boldsymbol{Q}$. Each element e_{ij} in $\boldsymbol{E}$ represents the inner product result between the key vector $\boldsymbol{k}_i^T$ and the query vector $\boldsymbol{q}_j$. The matrix multiplication $\boldsymbol{K}^T\boldsymbol{Q}$ computes all similarity scores between all query vectors and all key vectors. The column-wise softmax operator is used to normalize the coefficient matrix and make the sum of each column to 1. The matrix multiplication between $\boldsymbol{V}$ and softmax$(\boldsymbol{E})$ produces the output $\boldsymbol{O}$. Self-attention [17] is a special attention operator with $\boldsymbol{Q} = \boldsymbol{K} = \boldsymbol{V}$.

In Eq. 3.1, we employ dot product to calculate responses between query vectors in $\boldsymbol{Q}$ and key vectors in $\boldsymbol{K}$. There are several other ways to perform this computation, such as Gaussian function and concatenation. Dot product is shown to be the simplest but most effective one [30]. In this work, we use dot product as the similarity function. In general, we can apply linear transformations on input matrices, leading to following attention operator:

$$\begin{aligned} \boldsymbol{E} &= (\boldsymbol{W}^K\boldsymbol{K})^T\boldsymbol{W}^Q\boldsymbol{Q} && \in\mathbb{R}^{n\times m}, \\ \boldsymbol{O} &= \boldsymbol{W}^V\boldsymbol{V}\text{softmax}(\boldsymbol{E}) && \in\mathbb{R}^{p'\times m}, \end{aligned} \tag{3.2}$$

where $\boldsymbol{W}^V \in \mathbb{R}^{p'\times p}$ $\boldsymbol{W}^K \in \mathbb{R}^{d'\times d}$ and $\boldsymbol{W}^Q \in \mathbb{R}^{d'\times d}$. In the following discussions, we will skip linear transformations for the sake of notational simplicity.

The computational cost of the attention operator as described in Eq. 3.1 is $O(n \times d \times m) + O(p \times n \times m) = O(n \times m \times (d + p))$. The space complexity for storing intermediate coefficient matrix $\boldsymbol{E}$ is $O(n \times m)$. If $d = p$ and $m = n$, the time and space complexities are $O(m^2 \times d)$ and $O(m^2)$, respectively.

3.2.2 Hard Attention Operator

The attention operator described above uses soft attention, since responses to each query vector $\boldsymbol{q}_i$ are calculated by taking weighted sum over all value vectors. In contrast, hard attention operator [31] only selects a subset of key and value vectors for computation. Suppose k key vectors ($k < n$) are selected from the input matrix $\boldsymbol{K}$ and the indices are $i_1, i_2, \cdots, i_k$ with $i_m < i_n$ and $1 \leq m < n \leq k$. With selected indices, new key and value matrices are constructed as $\tilde{\boldsymbol{K}} = [\mathbf{k}_{i_1}, \mathbf{k}_{i_2}, \ldots, \mathbf{k}_{i_k}] \in \mathbb{R}^{d \times k}$ and $\tilde{\boldsymbol{V}} = [\mathbf{v}_{i_1}, \mathbf{v}_{i_2}, \ldots, \mathbf{v}_{i_k}] \in \mathbb{R}^{p \times k}$. The output of the hard attention operator is obtained by $\boldsymbol{O} = \text{attn}(\boldsymbol{Q}, \tilde{\boldsymbol{K}}, \tilde{\boldsymbol{V}})$. The hard attention operator is converted into a stochastic process in [31] by setting k to 1 and use probabilistic sampling. For each query vector, it only selects one value vector by probabilistic sampling based on normalized similarity scores given by $\text{softmax}(\boldsymbol{K}\boldsymbol{q}_i)$. The hard attention operators using probabilistic sampling are not differentiable, and requires reinforcement learning techniques for training. This makes soft attention more popular for easier back-propagation training [39].

By attending to less key vectors, the hard attention operator is computationally more efficient than the soft attention operator. The time and space complexities of the hard attention operator are $O(m \times k \times d)$ and $O(m \times k)$, respectively. When $k \ll m$, the hard attention operator reduces time and space complexities by a factor of m compared to the soft attention operator. Besides computational efficiency, the hard attention operator is shown to have better performance than the soft attention operator [31, 10], because it only selects important feature vectors to attend [40, 41].

3.2.3 Graph Attention Operator

The graph attention operator (GAO) was proposed in [15], and it applies the soft attention operator on graph data. For each node in a graph, it attends to its neighboring nodes. Given a graph with N nodes, each with d features, the layer-wise forward propagation operation of GAO in [15] is defined as

$$\begin{aligned} \tilde{\boldsymbol{E}} &= (\boldsymbol{X}^T\boldsymbol{X}) \circ \boldsymbol{A} \quad &\in \mathbb{R}^{N\times N}, \\ \boldsymbol{O} &= \boldsymbol{X}\text{softmax}(\tilde{\boldsymbol{E}}) \quad &\in \mathbb{R}^{d\times N}, \end{aligned} \tag{3.3}$$

where $\circ$ denotes element-wise matrix multiplication, $\boldsymbol{A} \in \{0,1\}^{N\times N}$ and $\boldsymbol{X} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N] \in \mathbb{R}^{d\times N}$ are the adjacency and feature matrices of a graph. Each $\boldsymbol{x}_i \in \mathbb{R}^d$ is node i's feature vector. In some situations, $\boldsymbol{A}$ can be normalized as needed [14]. Note that the softmax function only applies to nonzero elements of $\tilde{\boldsymbol{E}}$.

The time complexity of GAO is $O(Cd)$, where C is the number of edges. On a dense graph with $C \approx N^2$, this reduces to $O(N^2d)$. On a sparse graph, sparse matrix operations are required to compute GAO with this efficiency. However, current tensor manipulation frameworks such as TensorFlow do not support efficient batch training with sparse matrix operations [15], making it hard to achieve this efficiency. In general, GAO consumes excessive computational resources, preventing its applications on large graphs.

3.3 Hard and Channel-Wise Attention Operators and Networks

In this section, we describe our proposed hard graph attention operator (hGAO) and channel-wise graph attention operator (cGAO). hGAO applies the hard attention operation on graph data, thereby saving computational cost and improving performance. cGAO performs attention operation on channels, which avoids the dependency on the adjacency matrix and significantly improves efficiency in terms of computational resources. Based on these operators, we propose the deep graph attention networks for network embedding learning.

3.3.1 Hard Graph Attention Operator

Graph attention operator (GAO) consumes excessive computation resources, including computational cost and memory usage, when graphs have a large number of nodes, which is very common in real-world applications. Given a graph with N nodes, each with d features, GAO requires $O(N^2d)$ and $O(N^2)$ time and space complexities to compute its outputs. This means the computation cost and memory required grow quadratically in terms of graph size. This prohibits the application of GAO on graphs with a large number of nodes. In addition, GAO uses the soft attention mechanism, which computes responses of each node from all neighboring nodes in the graph. Using hard attention operator to replace the soft attention operator can reduce computational cost and improve learning performance. However, there is still no hard attention operator on graph data to the best of our knowledge. Direct use of the hard attention operator as in [31] on graph data still incurs excessive computational resources. It requires the computation of the normalized similarity scores for probabilistic sampling, which is the key factor of high requirements on computational resources.

To address the above limitations of GAO, we propose the hard graph attention operator (hGAO) that applies hard attention on graph data to save computational resources. For all nodes in a graph, we use a projection vector $\boldsymbol{p} \in \mathbb{R}^d$ to select the k-most important nodes to attend. Following the notations defined in Section 3.2, the layer-wise forward propagation function of hGAO is defined as

$$\boldsymbol{y} = \frac{|\boldsymbol{X}^T\boldsymbol{p}|}{\|\boldsymbol{p}\|} \qquad \in\mathbb{R}^N \tag{3.4}$$

for $i = 1, 2, \cdots, N$ do

$$\boldsymbol{idx}_i = \text{Ranking}_k(\boldsymbol{A}_{:\boldsymbol{i}} \circ \boldsymbol{y}) \qquad \in\mathbb{R}^k \tag{3.5}$$

$$\hat{\boldsymbol{X}}_i = \boldsymbol{X}(:, \boldsymbol{idx}_i) \qquad \in\mathbb{R}^{d\times k} \tag{3.6}$$

$$\tilde{\boldsymbol{y}}_i = \text{sigmoid}(\boldsymbol{y}(\boldsymbol{idx}_i)) \qquad \in\mathbb{R}^k \tag{3.7}$$

$$\tilde{\boldsymbol{X}}_i = \hat{\boldsymbol{X}}_i \text{diag}(\tilde{\boldsymbol{y}}_i) \qquad \in\mathbb{R}^{d\times k} \tag{3.8}$$

$$\boldsymbol{z}_i = \text{attn}(\boldsymbol{x}_i, \tilde{\boldsymbol{X}}_i, \tilde{\boldsymbol{X}}_i) \qquad \in\mathbb{R}^d \tag{3.9}$$

$$\boldsymbol{Z} = [\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots, \boldsymbol{z}_N] \qquad \in\mathbb{R}^{d\times N} \tag{3.10}$$

where $\boldsymbol{A}_{:\boldsymbol{i}}$ denotes the ith column of matrix $\boldsymbol{A}$, $\boldsymbol{X}(:, \boldsymbol{idx}_i)$ contains a subset of columns of $\boldsymbol{X}$ indexed by $\boldsymbol{idx}_i$, $|\cdot|$ computes element-wise absolute values, $\circ$ denotes element-wise matrix/vector multiplication, diag$(\cdot)$ constructs a diagonal matrix with the input vector as diagonal elements, Ranking$_k(\cdot)$ is an operator that performs the k-most important nodes selection for the query node i to attend and is described in detail below.

We propose a novel node selection method in hard attention. For each node in the graph, we adaptively select the k most important adjacent nodes. By using a trainable projection vector $\boldsymbol{p}$, we compute the absolute scalar projection of $\boldsymbol{X}$ on $\boldsymbol{p}$ in Eq. (3.4), resulting in $\boldsymbol{y} = [y_1, y_2, \cdots, y_N]^T$. Here, each y_i measures the importance of node i. For each node i, the Ranking$_k(\cdot)$ operation in Eq. (3.5) ranks node i's adjacent nodes by their projection values in $\boldsymbol{y}$, and selects nodes with the k largest projection values. Suppose the indices of selected nodes for node i are $\boldsymbol{idx}_i = [i_1, i_2, \cdots, i_k]$, node i attends to these k nodes, instead of all adjacent nodes. In Eq. (3.6), we extract new feature vectors $\hat{\boldsymbol{X}}_i = [\boldsymbol{x}_{i_1}, \boldsymbol{x}_{i_2}, \ldots, \boldsymbol{x}_{i_k}] \in \mathbb{R}^{d\times k}$ using the selected indices $\boldsymbol{idx}_i$. Here, we propose to use a gate operation to control information flow. In Eq. (3.7), we obtain the gate vector $\tilde{\boldsymbol{y}}$ by applying the sigmoid function to the selected scalar projection values $\boldsymbol{y}(\boldsymbol{idx}_i)$. By

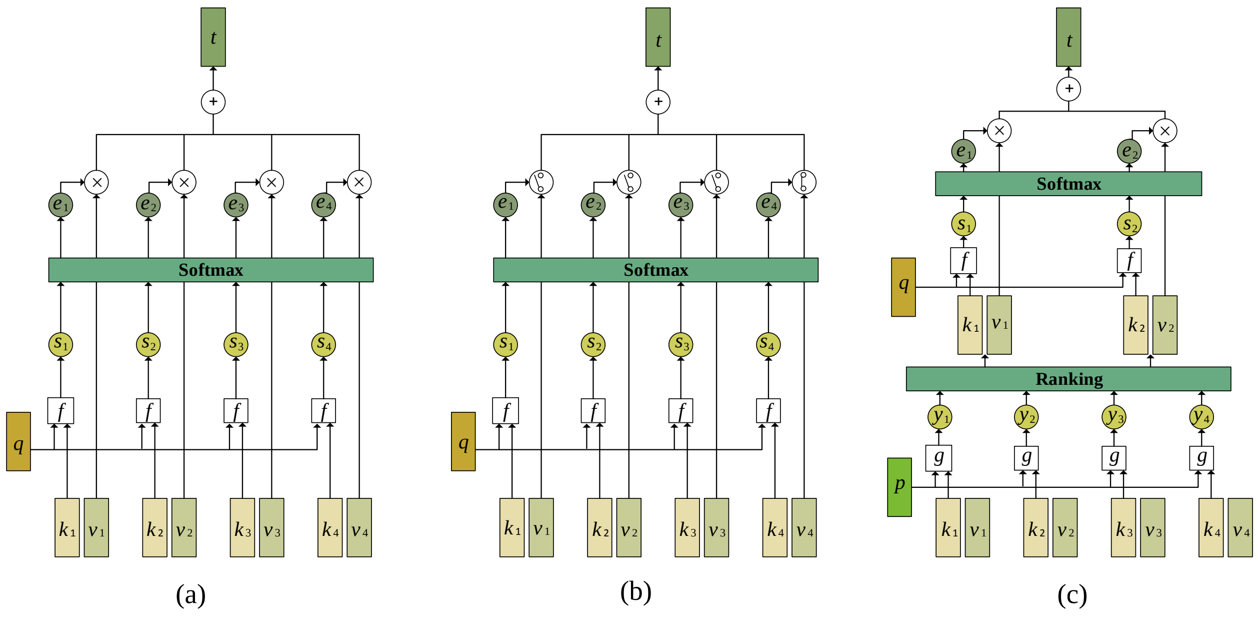

Figure 3.1: Illustration of GAO (a), hard attention operator in [31] (b), and our proposed hGAO (c). $\boldsymbol{q}$ is the feature vector of a node with four neighboring nodes in a graph. $\boldsymbol{k}_i$ and $\boldsymbol{v}_i$ are key and value vectors of the neighboring node i. In GAO (a), similarity scores are computed between query vector and key vectors, leading to scalar values s_i. The softmax normalizes these values and converts them into weights. The output is computed by taking a weighted sum of value vectors. In hard attention operator (b), the output is generated by probabilistic sampling, which samples a vector from value vectors using computed weights e_i. In hGAO (c), a projection vector $\boldsymbol{p}$ is used to compute the importance scores y_i. Based on these importance scores, two out of four nodes are selected by ranking. The output is computed by applying soft attention on selected nodes.

matrix multiplication $\hat{\boldsymbol{X}}_i\text{diag}(\tilde{\boldsymbol{y}}_i)$ in Eq. (3.8), we control the information of selected nodes and make the projection vector $\boldsymbol{p}$ trainable with gradient back-propagation. We use attention operator to compute the response of node i in Eq. (3.9). Finally, we construct the output feature matrix $\boldsymbol{Z}$ in Eq. (3.10). Note that the projection vector $\boldsymbol{p}$ is shared across all nodes in the graph. This means hGAO only involves d additional parameters, which may not increase the risk of over-fitting.

By attending to less nodes in graphs, hGAO is computationally more efficient than GAO. The time complexity of hGAO is $O(N \times \log N \times k + N \times k \times d^2)$ if using max heap for k-largest selection. When $k \ll N$ and $d \ll N$, hGAO consumes less time compared to the GAO. The space complexity of hGAO is $O(N^2)$ since we need to store the intermediate score matrix during k-most important nodes selection. Besides computational efficiency, hGAO is expected to have better performance than GAO, because it selects important neighboring nodes to attend [40]. We show in our experiments that hGAO outperforms GAO, which is consistent with the performance of hard attention operators in NLP and computation vision fields [31, 10].

This method can be considered as a trade-off between soft attention and the hard attention in [31]. The query node attends all neighboring nodes in soft attention operators. In the hard attention operator in [31], the query node attends to only one node that is probabilistically sampled from neighboring nodes based on the coefficient scores. In our hGAO, we employ an efficient ranking method to select k-most important neighboring nodes for the query node to attend. This avoids computing the coefficient matrix and reduces computational cost. The proposed gate operation enables training of the projection vector $\boldsymbol{p}$ using back-propagation [42], thereby avoiding the need of using reinforcement learning methods [43] for training as in [31]. Figure 3.1 provides illustrations and comparisons among soft attention operator, hard attention in [31], and our proposed hGAO.

Another possible way to compute the hard attention operator as hGAO is to implement the k-most important node selection based on the coefficient matrix. For each query node, we can select k neighboring nodes with k-largest similarity scores. The responses of the query node is calculated by attending to these k nodes. This method is different from our hGAO in that it needs to compute the coefficient matrix, which takes $O(N^2 \times d)$ time complexity. The hard attention operator using this implementation consumes much more computational resources than hGAO. In addition, the selection process in hGAO employs a trainable projection vector $\boldsymbol{p}$ to achieve important node selection. Making the projection vector $\boldsymbol{p}$ trainable allows for the learning of importance scores from data.

3.3.2 Channel-Wise Graph Attention Operator

The proposed hGAO computes the hard attention operator on graphs with reduced time complexity, but it still incurs the same space complexity as GAO. At the same time, both GAO and hGAO need to use the adjacency matrix to identify the neighboring nodes for the query node in the graph. Unlike grid like data such as images and texts, the number and ordering of neighboring nodes in a graph are not fixed. When performing attention operations on graphs, we need to rely on

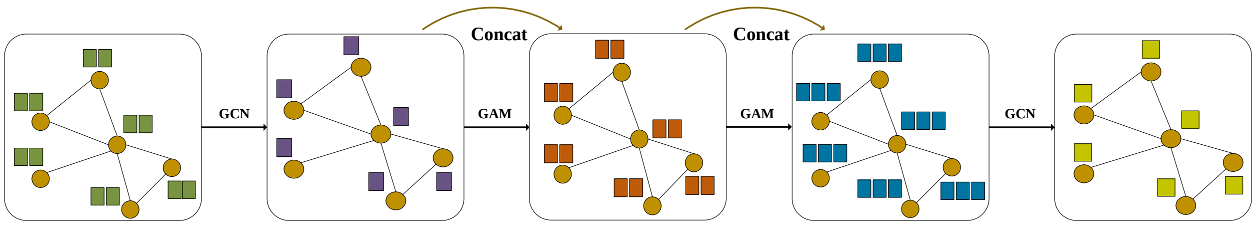

Figure 3.2: An illustration of our proposed GANet. In this example, the input graph contains 6 nodes, each of which has two features. A GCN layer is used to transform input feature vectors into low-dimensional representations. After that, we stack two GAMs for feature extraction. To facilitate feature reuse and gradients back-propagation, we add skip concatenation connections for GAMs. Finally, a GCN layer is used to output designated number of feature maps, which can be directly used for node classification predictions or used as inputs of following operations.

the adjacency matrix, which causes additional usage of computational resources. To further reduce the computational resources required by attention operators on graphs, we propose the channel-wise graph attention operator, which gains significant advantages over GAO and hGAO in terms of computational resource requirements.

Both GAO and our hGAO use the node-wise attention mechanism in which the output feature vector of node i is obtained by attending the input feature vector to all or selected neighboring nodes. Here, we propose to perform attention operation from the perspective of channels, resulting in our channel-wise graph attention operator (cGAO). For each channel $\boldsymbol{X}_{i:}$, we compute its responses by attending it to all channels. The layer-wise forward propagation function of cGAO can be expressed as

$$\begin{aligned} \boldsymbol{E} &= \boldsymbol{X}\boldsymbol{X}^T && \in \mathbb{R}^{d\times d}, \\ \boldsymbol{O} &= \text{softmax}(\boldsymbol{E})\boldsymbol{X} && \in \mathbb{R}^{d\times N}. \end{aligned} \tag{3.11}$$

Note that we avoid the use of adjacency matrix $\boldsymbol{A}$, which is different from GAO and hGAO. When computing the coefficient matrix $\boldsymbol{E}$, the similarity score between two feature maps $\boldsymbol{X}_{i:}$ and $\boldsymbol{X}_{j:}$ are calculated by $e_{ij} = \sum_{k=1}^{N} X_{ik} \times X_{jk}$. It can be seen that features within the same node communicate with each other, and there is no communication among features located in different nodes. This means we do not need the connectivity information provided by adjacency matrix $\boldsymbol{A}$, thereby avoiding the dependency on the adjacency matrix used in node-wise attention operators.

Table 3.1: Comparison of time and space complexities among GAO, hGAO, and cGAO in terms of time and space complexities.

Operator	Time Complexity	Space Complexity
GAO	$O(N^2 \times d)$	$O(N^2)$
hGAO	$O(N \times \log N \times k + N \times k \times d^2)$	$O(N^2)$
cGAO	$O(N \times d^2)$	$O(d^2)$

This saves computational resources related to operations with the adjacency matrix.

The computational cost of cGAO is $O(Nd^2)$, which is lower than that of GAO if $d < N$. When applying attention operators on graph data, we can control the number of feature maps d, but it is hard to reduce the number of nodes in graphs. On large graphs with $d \ll N$, cGAO has computational advantages over GAO and hGAO, since its time complexity is only linear to the size of graphs. The space complexity of cGAO is $O(d^2)$, which is independent of graph size. This means the application of cGAO on large graphs does not suffer from memory issues, which is especially useful on memory limited devices such as GPUs and mobile devices. Table 3.1 provides theoretical comparisons among GAO, hGAO and cGAO in terms of the time and space complexities. Therefore, cGAO enables efficient parallel training by removing the dependency on the adjacency matrix in graphs and significantly reduces the usage of computational resources.

3.3.3 The Proposed Graph Attention Networks

To use our hGAO and cGAO, we design a basic module known as the graph attention module (GAM). The GAM consists of two operators; those are, a graph attention operator and a graph convolutional network (GCN) layer [14]. We combine these two operators to enable efficient information propagation within graphs. For GAO and hGAO, they aggregate information from neighboring nodes by taking weighted sum of feature vectors from adjacent nodes. But there exists a situation that weights of some neighboring nodes are close to zero, preventing the information propagation of these nodes. In cGAO, the attention operator is applied among channels and does not involve information propagation among nodes. To overcome this limitation, we use a GCN layer, which applies the same weights to neighboring nodes and aggregate information from

Table 3.2: Statistics of datasets used in graph classification tasks under inductive learning settings. We use the D&D, PROTEINS, COLLAB, MUTAG, PTC, and IMDB-M datasets.

Dataset	Total	Train	Test	Nodes	Degree	Classes
MUTAG	188	170	18	17.9	2.2	2
PTC	344	310	34	25.6	2.0	2
PROTEINS	1113	1002	111	39.1	3.7	2
D&D	1178	1061	117	284.3	5.0	2
IMDB-M	1500	1350	150	13.0	10.1	3
COLLAB	5000	4500	500	74.5	66.0	3

all adjacent nodes. Note that we can use any graph attention operator such as GAO, hGAO and cGAO. To facilitate feature reuse and gradients back-propagation, we add a skip connection by concatenating inputs and outputs of the GCN layer.

Based on GAM, we design graph attention networks, denoted as GANet, for network embedding learning. In GANet, we first apply a GCN layer, which acts as a graph embedding layer to produce low-dimensional representations for nodes. In some data like citation networks dataset [14], nodes usually have very high-dimensional feature vectors. After the GCN layer, we stack multiple GAMs depending on the complexity of the graph data. As each GAM only aggregates information from neighboring nodes, stacking more GAMs can collect information from a larger parts of the graph. Finally, a GCN layer is used to produce designated number of output feature maps. The outputs can be directly used as predictions of node classification tasks. We can also add more operations to produce predictions for graph classification tasks. Figure 3.2 provides an example of our GANet. Based on this network architecture, we denote the networks using GAO, hGAO and cGAO as GANet, hGANet and cGANet, respectively.

3.4 Experimental Studies

In this section, we evaluate our proposed graph attention networks on node classification and graph classification tasks. We first compare our hGAO and cGAO with GAO in terms of computation resources such as computational cost and memory usage. Next, we compare our hGANet and cGANet with prior state-of-the-art models under inductive and transductive learning settings. Per-

Table 3.3: Statistics of datasets used in node classification tasks under transductive learning settings. We use the Cora, Citeseer, and Pubmed datasets.

Dataset	Nodes	Features	Train	Valid	Test	Degree	Classes
Cora	2708	1433	140	500	1000	4	7
Citeseer	3327	3703	120	500	1000	5	6
Pubmed	19717	500	60	500	1000	6	3

formance studies among GAO, hGAO, and cGAO are conducted to show that our hGAO and cGAO achieve better performance than GAO. We also conduct some performance studies to investigate the selection of some hyper-parameters.

3.4.1 Datasets

We conduct experiments on graph classification tasks under inductive learning settings and node classification tasks under transductive learning settings. Under inductive learning settings, training and testing data are separate. The test data are not accessible during training time. The training process will not learn about graph structures of the test data. For graph classification tasks under inductive learning settings, we use the MUTAG [18], PTC [18], PROTEINS [44], D&D [45], IMDB-M [46], and COLLAB [46] datasets to fully evaluate our proposed methods. MUTAG, PTC, PROTEINS and D&D are four benchmarking bioinformatics datasets. MUTAG and PTC are much smaller than PROTEINS and D&D in terms of number of graphs and average nodes in graphs. Compared to large datasets, evaluations on small datasets can help investigate the risk of over-fitting, especially for deep learning based methods. COLLAB, IMDB-M are two social network datasets. For these datasets, we follow the same settings as in [47], which employs 10-fold cross validation [48] with 9 folds for training and 1 fold for testing. The statistics of these datasets are summarized in Table 3.2.

Unlike inductive learning settings, the unlabeled data and graph structure are accessible during the training process under transductive learning settings. To be specific, only a small part of nodes in the graph are labeled while the others are not. For node classification tasks under transductive learning settings, we use three benchmarking datasets; those are Cora [20], Citeseer, and

Table 3.4: Comparison of results among GAO, hGAO, and cGAO on different graph sizes in terms of the number of MAdd, memory usage, and CPU prediction time. The input sizes are describe by "number of nodes $\times$ number of features". The prediction time is the total execution time on CPU.

Input	Layer	MAdd	Cost Saving	Memory	Memory Saving	Time	Speedup
$1k \times 48$	GAO	100.61m	0.00%	4.98MB	0.00%	8.19ms	1.0×
	hGAO	37.89m	62.34%	4.98MB	0.00%	5.61ms	1.4×
	cGAO	**9.21m**	**90.84%**	**0.99MB**	**80.12%**	**0.82ms**	**9.9**×
$10k \times 48$	GAO	9.65b	0.00%	0.41GB	0.00%	0.95s	1.0×
	hGAO	0.46b	95.14%	0.41GB	0.00%	0.37s	2.5×
	cGAO	**0.09b**	**99.04%**	**9.61MB**	**97.65%**	**0.017s**	**52.7**×
$20k \times 48$	GAO	38.49b	0.00%	1.62GB	0.00%	12.78s	1.0×
	hGAO	1.13b	97.04%	1.62GB	0.00%	4.55s	2.8×
	cGAO	**0.18b**	**99.52%**	**19.2MB**	**98.81%**	**0.029s**	**430.3**×

Pubmed [14]. These datasets are citation networks. Each node in the graph represents a document while an edge indicates a citation relationship. The graphs in these datasets are attributed and the feature vector of each node is generated by bag-of-word representations. The dimensions of feature vectors of three datasets are different depending on the sizes of dictionaries. The statistics of these datasets are summarized in Table 3.3. Following the same experimental settings in [14], we use 20 nodes, 500 nodes, and 500 nodes for training, validation, and testing, respectively.

3.4.2 Experimental Setup

In this section, we describe the experimental setup for inductive learning and transductive learning tasks. For inductive learning tasks, we adopt the model architecture of DGCNN [47]. DGCNN consists of four parts; those are graph convolution layers, soft pooling, 1-D convolution layers and dense layers. We replace graph convolution layers with our hGANet described in Section 3.3.3 and the other parts remain the same. The hGANet contains a starting GCN layer, four GAMs and an ending GCN layer. Each GAM is composed of a hGAO, and a GCN layer. The starting GCN layer outputs 48 feature maps. Each hGAO and GCN layer within GAMs outputs 12 feature maps. The final GCN layer produces 97 feature maps as the original graph convolution layers in DGCNN. The skip connections using concatenation is employed between the input and output feature maps of each GAM. The hyper-parameter k is set to 8 in each hGAO, which means each node in a graph selects 8 most important neighboring nodes to compute the response. We apply dropout [23] with the keep rate of 0.5 to the feature matrix in every GCN layer. For experiments on cGANet, we use the same settings.

For transductive learning tasks, we use our hGANet to perform node classification predictions. Since the feature vectors for nodes are generated using the bag-of-words method, they are high-dimensional sparse features. The first GCN layer acts like an embedding layer to reduce them into low-dimensional features. To be specific, the first GCN layer outputs 48 feature maps to produce 48 embedding features for each node. For different datasets, we stack different number of GAMs. Specifically, we use 4, 2, and 3 GAMs for Cora, Citeseer, and Pubmed, respectively. Each hGAO and GCN layer in GAMs outputs 16 feature maps. The last GCN layer produces the prediction on each node in the graph. We apply dropout with the keep rate of 0.12 on feature matrices in each layer. We also set k to 8 in all hGAOs. We employ identity activation function as [49] for all layers in the model. To avoid over-fitting, we apply L_2 regularization with $\lambda = 0.0001$. All trainable weights are initialized with Glorot initialization [25]. We use Adam optimizer [24] for training.

3.4.3 Comparison of Computational Efficiency

According to the theoretical analysis in Section 3.3, our proposed hGAO and cGAO have efficiency advantages over GAO in terms of the computational cost and memory usage. The advantages are expected to be more obvious as the increase of the number of nodes in a graph. In this section, we conduct simulated experiments to evaluate these theoretical analysis results. To reduce the influence of external factors, we use the network with a single graph attention operator and apply TensorFlow profile tool [50] to report the number of multiply-adds (MAdd), memory usage, and CPU inference time on simulated graph data.

The simulated data are create with the shape of "number of nodes $\times$ number of feature maps". For all simulated experiments, each node on the input graph has 48 features. We test three graph sizes; those are 1000, 1,0000, and 20,000, respectively. All tested graph operators output 48 feature maps including GAO, hGAO, and cGAO. For hGAOs, we set $k = 8$ in all experiments, which is the value of hyper-parameter k tuned on graph classification tasks. We report the number of MAdd,

Table 3.5: Comparison of results of node classification experiments with prior state-of-the-art models on the Cora, Citeseer, and Pubmed datasets.

Models	Cora	Citeseer	Pubmed
DeepWalk [26]	67.2%	43.2%	65.3%
Planetoid [21]	75.7%	64.7%	77.2%
Chebyshev [27]	81.2%	69.8%	74.4%
GCN [14]	81.5%	70.3%	79.0%
GAT [15]	$83.0 \pm 0.7\%$	$72.5 \pm 0.7\%$	$79.0 \pm 0.3\%$
hGANet	**$83.5 \pm 0.7\%$**	**$72.7 \pm 0.6\%$**	**$79.2 \pm 0.4\%$**

Table 3.6: Comparison of results of graph classification experiments with prior state-of-the-art models in terms of accuracies on the D&D, PROTEINS, COLLAB, MUTAG, PTC, and IMDB-M datasets. “-” denotes the result not available.

Models	D&D	PROTEINS	COLLAB	MUTAG	PTC	IMDB-M
GRAPHSAGE [28]	75.42%	70.48%	68.25%	-	-	-
PSCN [18]	76.27%	75.00%	72.60%	88.95%	62.29%	45.23%
SET2SET [51]	78.12%	74.29%	71.75%	-	-	-
DGCNN [47]	79.37%	76.26%	73.76%	85.83%	58.59%	47.83%
DiffPool [52]	80.64%	76.25%	75.48%	-	-	-
cGANet	80.86%	78.23%	76.96%	89.00%	63.53%	48.93%
hGANet	**81.71%**	**78.65%**	**77.48%**	**90.00%**	**65.02%**	**49.06%**

memory usage, and CPU inference time.

The comparison results are summarized in Table 3.4. On the graph with 20,000 nodes, our cGAO and hGAO provide 430.31 and 2.81 times speedup compared to GAO. In terms of the memory usage, cGAO can save 98.81% compared to GAO and hGAO. When comparing across different graph sizes, the effects of speedup and memory saving are more apparent as the graph size increases. This is consistent with our theoretical analysis on hGAO and cGAO. Our hGAO can save computational cost compared to GAO. cGAO achieves great computational resources reduction, which makes it applicable on large graphs. Note that the speed up of hGAO over GAO is not as apparent as the computational cost saving due to the practical implementation limitations.

Table 3.7: Comparison of results of graph classification experiments among GAO, hGAO, and cGAO in terms of accuracies on the D&D, PROTEINS, COLLAB, MUTAG, PTC, and IMDB-M datasets. “OOM” denotes out of memory.

Models	**D&D**	**PROTEINS**	**COLLAB**	**MUTAG**	**PTC**	**IMDB-M**
GANet	OOM	77.92%	76.06%	87.22%	62.94%	48.89%
cGANet	80.86%	78.23%	76.96%	89.00%	63.53%	48.93%
hGANet	**81.71%**	**78.65%**	**77.48%**	**90.00%**	**65.02%**	**49.06%**

3.4.4 Results on Inductive Learning Tasks

In this section, we evaluate our methods on graph classification tasks under inductive learning settings. To compare our proposed cGAOs with hGAO and GAO, we replace hGAOs with cGAOs in hGANet, denoted as cGANet. We compare our models with prior sate-of-the-art models on D&D, PROTEINS, COLLAB, MUTAG, PTC, and IMDB-M datasets, which serve as the benchmarking datasets for graph classification tasks. The results are summarized in Table 3.6.

From the results, we can observe that the our hGANet consistently outperforms DiffPool [52] by margins of 0.90%, 1.40%, and 2.00% on D&D, PROTEINS, and COLLAB datasets, which contain relatively big graphs in terms of the average number of nodes in graphs. Compared to DGCNN, the performance advantages of our hGANet are even larger. The superior performances on large benchmarking datasets demonstrate that our proposed hGANet is promising since we only replace graph convolution layers in DGCNN. The performance boosts over the DGCNN are consistently and significant, which indicates the great capability on feature extraction of hGAO compared to GCN layers.

On datasets with smaller graphs, our GANets outperform prior state-of-the-art models by margins of 1.05%, 2.71%, and 1.23% on MUTAG, PTC, and IMDB-M datasets. The promising performances on small datasets prove that our methods improve the ability of high-level feature extraction without incurring the problem of over-fitting. cGANet outperforms prior state-of-the-art models but has lower performances than hGANet. This indicates that cGAO is also effective on feature extraction but not as powerful as hGAO. The attention on only important adjacent nodes incurred by using hGAOs helps to improve the performance on graph classification tasks.

3.4.5 Results on Transductive Learning Tasks

Under transductive learning settings, we evaluate our methods on node classification tasks. We compare our hGANet with prior state-of-the-art models on Cora, Citeseer, and Pubmed datasets in terms of the node classification accuracy. The results are summarized in Table 3.5. From the results, we can observe that our hGANet achieves consistently better performances than GAT, which is the prior state-of-the-art model using graph attention operator. Our hGANet outperforms GAT [15] on three datasets by margins of 0.5%, 0.2%, and 0.2%, respectively. This demonstrates that our hGAO has performance advantage over GAO by attending less but most important adjacent nodes, leading to better generalization and performance.

3.4.6 Comparison of cGAO and hGAO with GAO

Besides comparisons with prior state-of-the-art models, we conduct experiments under inductive learning settings to compare our hGAO and cGAO with GAO. To be fair, we replace all hGAOs with GAOs in hGANet employed on graph classification tasks, which results in GANet. GAOs output the same number of feature maps as the corresponding hGAOs. Like hGAOs, we apply linear transformations on key and value matrices. This means GANets have nearly the same number of parameters with hGANets, which additionally contain limited number of projection vectors in hGAOs. We adopt the same experimental setups as hGANet. We compare our hGANet and cGANet with GANet on all six datasets for graph classification tasks described in Section 3.4.1. The comparison results are summarized in Table 3.7.

The results show that our cGAO and hGAO have significantly better performances than GAO. Notably, GANet runs out of memory when training on D&D dataset with the same experimental setup as hGANet. This demonstrates that hGAO has memory advantage over GAO in practice although they share the same space complexity. cGAO outperforms GAO on all six datasets but has slightly lower performances than hGAO. Considering cGAO dramatically saves computational resources, cGAO is a good choice when facing large graphs. Since there is no work that realizes

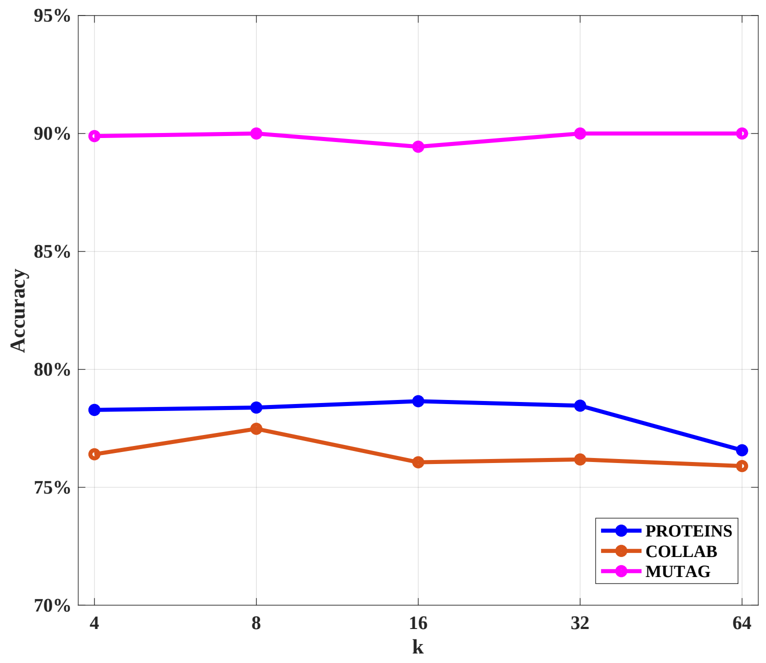

Figure 3.3: Results of employing different k values in hGAOs. We evaluate our hGANet on PROTEINS, COLLAB, and MUTAG datasets under inductive learning settings. We use the same experimental setups described in Section 3.4.2. We report the graph classification accuracies in this figure. We can see that the best performances is achieved when $k = 8$.

the hard attention operator in [31] on graph data, we do not provide comparisons with it in this work.

3.4.7 Performance Study of k in hGAO

Since k is an important hyper-parameter in hGAO, we conduct experiments to investigate the impact of different k values on hGANet. Based on hGANet, we vary the values of k in hGAOs with choices of 4, 8, 16, 32, and 64, which are reasonable selections for k. We report performances of hGANets with different k values on graph classification tasks on PROTEINS, COLLAB, and MUTAG datasets, which cover both large and small datasets.

The performance changes of hGANets with different k values are plotted in Figure 3.3. From the figure, we can see that hGANets achieve the best performances on all three datasets when $k = 8$. The performances start to decrease as the increase of k values. On PROTEINS and COLLAB datasets, the performances of hGANets with $k = 64$ are significantly lower than those with $k = 8$. This indicates that larger k values make the query node to attend more adjacent nodes in hGAO, which leads to worse generalization and performance.

4. GRAPH FEATURE LEARNING VIA LINE GRAPH STRUCTURES

In the previous section, we discussed how to learn high-level features by mocking regular deep learning methods such as convolution and attention operations. In this section, we investigate graph deep learning methods that learn new graph features using unique graph structures like line graph structures.

4.1 Introduction

Graph neural networks [53, 54, 28] have shown to be competent in solving challenging tasks in the field of network embedding. Many tasks have been significantly advanced by graph deep learning methods such as node classification tasks [14, 15, 49], graph classification tasks [52, 47], link prediction tasks [55, 56], and community detection tasks [57]. Currently, most graph neural networks capture the relationships among nodes through message passing operations. In a single layer-wise propagation, each node receives features from its neighbors in the graph and transforms them into its new feature representations [27, 58]. Thus, message passing operations need to rely on the graph topology information.

Recently, some works [57] use extra graph structures such as line graphs to enhance message passing operations in graph neural networks from different graph perspectives. A line graph is a graph that is derived from an original graph to represent connectivity between edges in the original graph. Since line graphs can faithfully encode the topology information, message passing operations on line graphs can enhance network embeddings in graph neural networks. However, graph neural networks that leverage line graph structures need to deal with two challenging issues; those are bias and inefficiency. Topology information in original graphs is faithfully encoded in line graphs but in a biased way. In particular, node features are either overstated or understated depending on their degrees. Besides, line graphs can be much bigger graphs than original graphs depending on the graph density. Message passing operations of graph neural networks on line graphs lead to significant use of computational resources.

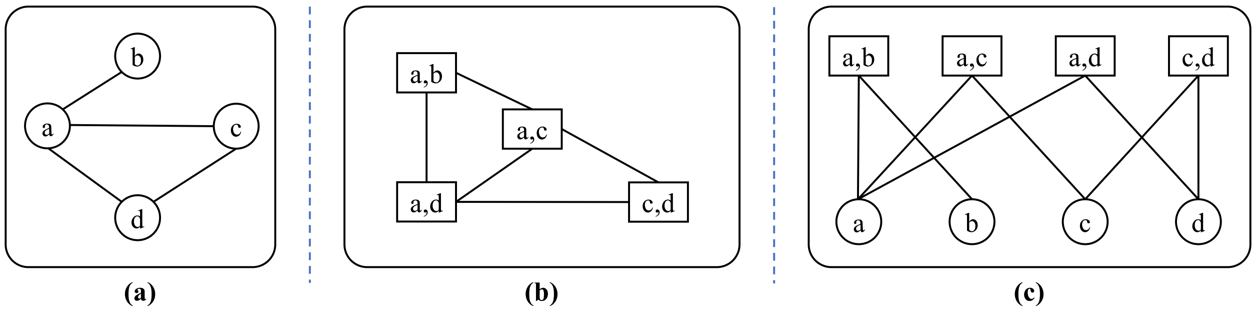

Figure 4.1: An illustration of an undirected graph (a), its corresponding line graph (b), and its incidence graph (c).

In this work, we propose to construct a weighted line graph that can correct biases in encoded topology information of line graphs. To this end, we assign each edge in a line graph a normalized weight such that each node in the line graph has a weighted degree of 2. In this weighted line graph, the dynamics of node features are the same as those in its original graph. Based on our weighted line graph, we propose a weighted line graph convolution layer (WLGCL) that performs a message passing operation on both original graph structures and weighted line graph structures. To address inefficiency issues existing in graph neural networks that use line graph structures, we further propose to implement our WLGCL via an incidence matrix, which can dramatically reduce the usage of computational resources. Based on our WLGCL, we build a family of weighted line graph convolutional networks (WLGCNs). We evaluate our methods on graph classification tasks and show that WLGCNs consistently outperform previous state-of-the-art models. Experiments on simulated data demonstrate the efficiency advantage of our implementation.

4.2 Background and Related Work

In this section, we describe the line graph and its applications in graph neural networks for network embedding learnings.

In graph theory, a line graph is a graph derived from an undirected graph. It represents the connectivity among edges in the original graph. Given a graph $\mathbb{G}$, the corresponding line graph $L(\mathbb{G})$ is constructed by using edges in $\mathbb{G}$ as vertices in $L(\mathbb{G})$. Two nodes in $L(\mathbb{G})$ are adjacent if they share a common end node in the graph $\mathbb{G}$ [59]. Note that the edges (a, b) and (b, a) in an undirected graph $\mathbb{G}$ correspond to the same vertex in the line graph $L(\mathbb{G})$. The Whitney graph isomorphism theorem [60] stated that a line graph has a one-to-one correspondence to its original graph. This theorem guarantees that the line graph can faithfully encode the topology information in the original graph.

An incidence graph is a bipartite graph that describes the incidence relationships among nodes and edges [61]. The two disjoint and independent nodes sets in the incidence graph are vertices and edges in $\mathbb{G}$, respectively. Besides, an incidence matrix is a matrix with each row and each column corresponding to a vertex and an edge, respectively. It shows the connectivity relationships between nodes and edges. Figure 4.1 provides an illustration of an undirected graph, its line graph, and its incidence graph. The above discussions are mainly based on undirected graphs, but they can be easily extended to other types of graphs.

Recently, [57] proposes to use the line graph structure to enhance the message passing operations in graph neural networks. Since the line graph can faithfully encode the topology information, the message passing on the line graph can enhance the network embeddings in graph neural networks. In graph neural networks that use line graph structures, features are passed and transformed in both the original graph structures and the line graph structures, thereby leading to better feature learnings and performances.

4.3 Weighted Line Graph Convolutional Networks

In this work, we propose the weighted line graph that addresses the bias in the line graph when encoding graph topology information. Based on our weighted line graph, we propose the weighted line graph convolution layer (WLGCL) for better feature learning by leveraging line graph structures. Besides, graph neural networks using line graphs consume excessive computational resources. To solve the inefficiency issue, we propose to use the incidence matrix to implement the WLGCL, which can dramatically reduce the usage of computational resources.

4.3.1 Benefit and Bias of Line Graph Representations

In this section, we describe the benefit and bias of using line graph representations.

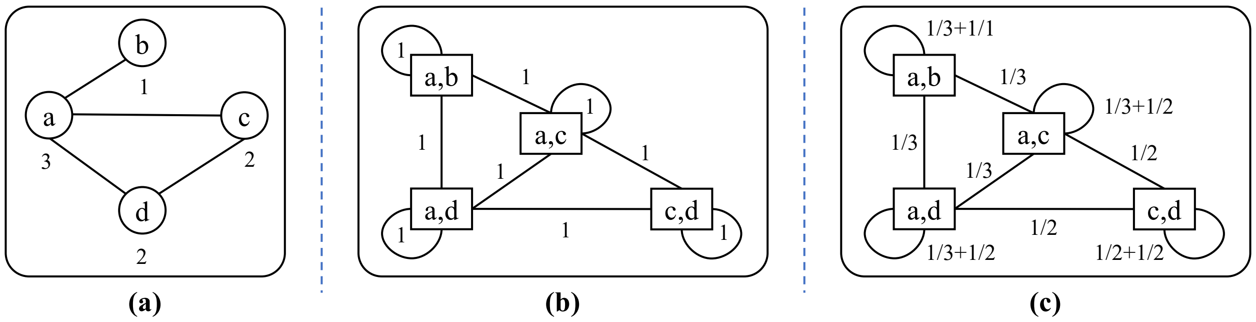

Figure 4.2: An illustration of a graph (a), its corresponding line graph (b), and its weighted line graph (c). Here, we consider a graph with 4 nodes and 4 edges as illustrated in (a). The numbers show the node degrees in the graph. In figure (b), a line graph is constructed with self-loops. Each node corresponds to an edge in the original graph. In the regular line graph, the weight of each edge is 1. Figure (c) illustrates the weighted line graph constructed as described in Section 4.3.2. The weight of each edge is assigned as defined in Eq. (4.1).

Benefit. In message-passing operations, edges are usually given equal importance and edge features are not well explored. This can constrain the capacity of GNNs, especially on graphs with edge features. In the chemistry domain, a compound can be converted into a graph, where atoms are nodes and chemical bonds are edges. On such kinds of graphs, edges have different properties and thus different importance. However, message-passing operations underestimate the importance of edges. To address this issue, the line graph structure can be used to leverage edge features and different edge importance [62, 57, 63]. The line graph, by its nature, enables graph neural networks to encode and propagate edge features in the graph. The line graph neural networks that take advantage of line graph structures have shown to be promising on graph-related tasks [57, 64, 65]. By encoding node and edge features simultaneously, line graph neural networks enhance the feature learning on graphs.

Bias. According to the Whitney graph isomorphism theorem, the line graph $L(\mathbb{G})$ encodes the topology information of the original graph $\mathbb{G}$, but the dynamics and topology of $\mathbb{G}$ are not correctly represented in $L(\mathbb{G})$ [66]. As described in the previous section, each edge in the graph $\mathbb{G}$ corresponds to a vertex in the line graph $L(\mathbb{G})$. The features of each edge contain features of its two end nodes. A vertex with a degree d in the original graph $\mathbb{G}$ will generate $d \times (d-1)/2$ edges in the

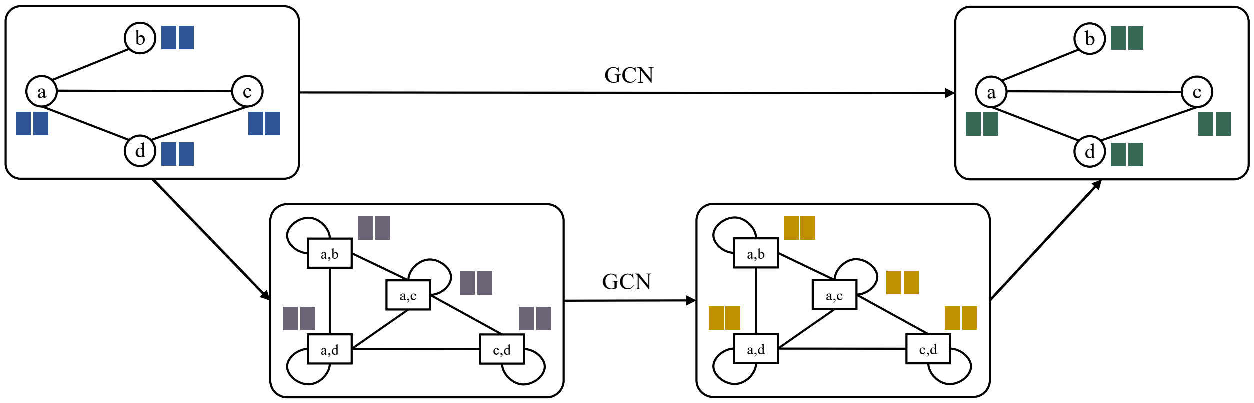

Figure 4.3: An illustration of our proposed weighted line graph convolution layer. We consider an input graph with 4 nodes and each node contains 2 features. Based on the input graph, we firstly construct the weighted line graph with features as described in Section 4.3.2. Then we apply two GCN layers on the original graph and the weighted line graph, respectively. The edge features in the line graph are transformed back into node features and combined with features in the original graph.

line graph $L(\mathbb{G})$. The message passing frequency of this node's features will change from $O(d)$ in the original graph to $O(d^2)$ in the line graph. From this point, the line graph encodes the topology information in the original graph but in a biased way. In the original graph, a node's features will be passed to d neighbors. But in the corresponding line graph, the information will be passed to $d \times (d-1)/2$ nodes. The topology structure in the line graph $L(\mathbb{G})$ will overstate the importance of features for nodes with high degrees in the graph. On the contrary, the nodes with smaller degrees will be relatively understated, thereby leading to biased topology information encoded in the line graph. Note that popular adjacency matrix normalization methods [14, 15, 67, 68] cannot address this issue.

4.3.2 Weighted Line Graph

In the previous section, we show that the line graph $L(\mathbb{G})$ constructed from the original graph $\mathbb{G}$ encodes biased graph topology information. To address this issue, we propose to construct a weighted line graph that assigns normalized weights to edges. In a regular line graph $L(\mathbb{G})$, each edge is assigned an equal weight of 1, thereby leading to a biased encoding of the graph topology information. To correct the bias, we need to normalize edge weights in the line graph.

Considering each edge in $\mathbb{G}$ has two ends, it is intuitive to normalize the weighted degree of the corresponding node in $L(\mathbb{G})$ to 2. To this end, the weight for an edge in the adjacency matrix $\boldsymbol{F}$ of $L(\mathbb{G})$ is computed as:

$$F_{(a,b),(b,c)} = \begin{cases} \frac{1}{D_b} & \text{if } a \neq c \\ \frac{1}{D_b} + \frac{1}{D_a}, & \text{if } a = c \end{cases} \tag{4.1}$$

where a, b, and c are nodes in the graph $\mathbb{G}$, (a, b) and (b, c) are edges in the graph $\mathbb{G}$ that are connected by the node b. D_b is the degree of the node b in the graph $\mathbb{G}$. To facilitate the message passing operation, we add self-loops on the weighted line graph $WL(\mathbb{G})$. The weights for self-loop edges computed by the second case consider the fact that they are self-connected by both ends. Figure 4.2 illustrates an example of a graph and its corresponding weighted line graph.

Theorem 1. *Given the edge weights in the weighted line graph $WL(\mathbb{G})$ defined by Eq. (4.1), the weighted degree for a node (a, b) in $WL(\mathbb{G})$ is 2.*

Proof. Given nodes a and b with degrees D_a and D_b in a graph $\mathbb{G}$, a node (a, b) in the corresponding weighted line graph $WL(\mathbb{G})$ connects to $D_a - 1$ and $D_b - 1$ nodes through a and b in $\mathbb{G}$, respectively. The weighted degree of the node (a, b) is computed by summing up the weights of edges that connect (a, b) to other nodes through a and b, and the weight of its self-loop:

$$\begin{aligned} WLD_{(a,b)} &= \sum_{i=1}^{D_a - 1} \frac{1}{D_a} + \sum_{j=1}^{D_b - 1} \frac{1}{D_b} + \left(\frac{1}{D_a} + \frac{1}{D_b} \right) \\ &= \sum_{i=1}^{D_a} \frac{1}{D_a} + \sum_{j=1}^{D_b} \frac{1}{D_b} = 2. \end{aligned} \tag{4.2}$$

This completes the proof. □

By constructing the weighted line graph with normalized edge weights defined in Eq. (4.1), each node (a, b) has a weighted degree of 2. Given a node a with a degree of d, it has d related edges in $\mathbb{G}$ and d related nodes in $L(\mathbb{G})$. The message passing frequency of node a's features in the weighted line graph $WL(\mathbb{G})$ is $\sum_{i=1}^{d} 2 = O(d)$, which is consistent with that in the original graph $\mathbb{G}$. Thus, the weighted line graph faithfully encodes the topology information of the original graph in an unbiased way.

4.3.3 Weighted Line Graph Convolution Layer

In this section, we propose the weighted line graph convolution layer (WLGCL) that leverages our proposed weighted line graph for feature representations learnings. In this layer, node features are passed and aggregated in both the original graph $\mathbb{G}$ and the corresponding weighted line graph $WL(\mathbb{G})$.

Suppose an undirected attributed graph $\mathbb{G}$ has N nodes and E edges. Each node in the graph contains C features. In the layer ℓ, an adjacency matrix $\boldsymbol{A}^{(\ell)} \in \mathbb{R}^{N\times N}$ and a feature matrix $\boldsymbol{X}^{(\ell)} \in \mathbb{R}^{N\times C}$ are used to represent the graph connectivity and node features, respectively. Here, we construct the adjacency matrix $\boldsymbol{F}^{(\ell)} \in \mathbb{R}^{E\times E}$ of the corresponding weighted line graph. The layer-wise propagation rule of the weighted line graph convolution layer ℓ is defined as:

$$\boldsymbol{Y}^{(\ell)} = \boldsymbol{B}^{(\ell)^T}\boldsymbol{X}^{(\ell)}, \qquad \in\mathbb{R}^{E\times C} \tag{4.3}$$

$$\boldsymbol{Y}^{(\ell)} = \boldsymbol{F}^{(\ell)}\boldsymbol{Y}^{(\ell)}, \qquad \in\mathbb{R}^{E\times C} \tag{4.4}$$

$$\boldsymbol{K}_L^{(\ell)} = \boldsymbol{B}^{(\ell)}\boldsymbol{Y}^{(\ell)}, \qquad \in\mathbb{R}^{N\times C} \tag{4.5}$$

$$\boldsymbol{K}^{(\ell)} = \boldsymbol{A}^{(\ell)}\boldsymbol{X}^{(\ell)}, \qquad \in\mathbb{R}^{N\times C} \tag{4.6}$$

$$\boldsymbol{X}^{(\ell+1)} = \boldsymbol{K}^{(\ell)}\boldsymbol{W}^{(\ell)} + \boldsymbol{K}_L^{(\ell)}\boldsymbol{W}_L^{(\ell)}, \qquad \in\mathbb{R}^{N\times C'} \tag{4.7}$$

where $\boldsymbol{W}^{(\ell)} \in \mathbb{R}^{C\times C'}$ and $\boldsymbol{W}_L^{(\ell)} \in \mathbb{R}^{C\times C'}$ are matrices of trainable parameters. $\boldsymbol{B}^{(\ell)} \in \mathbb{R}^{N\times E}$ is the incidence matrix of the graph $\mathbb{G}$ that shows the connectivity between nodes and edges.

To enable message passing on the line graph $L(\mathbb{G})$, each edge in the graph $\mathbb{G}$ needs to have features. However, edge features are not available on some graphs. To address this issue, we first compute features for an edge (a, b) by summing up features of its two end nodes: $Y_{(a,b)}^{(\ell)} = X_a^{(\ell)} + X_b^{(\ell)}$. Here, we use the summation operation to ensure the permutation invariant property in this layer. Eq. (4.3) computes features for each edge by using the incidence matrix of the graph $\mathbb{G}$,

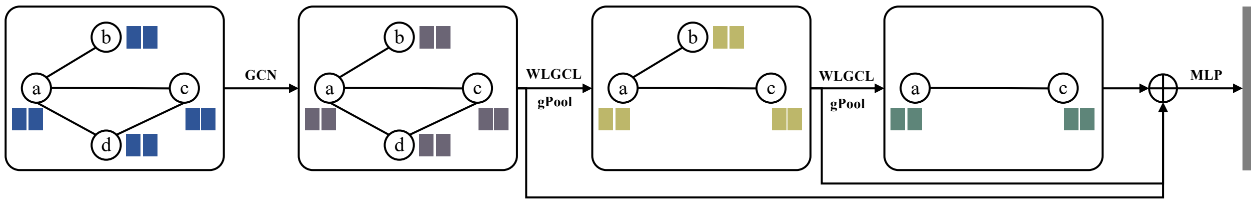

Figure 4.4: An illustration of the weighted line graph convolution network. The input graph is an undirected attributed graph. Each node in the graph contains two features. Here, we use a GCN layer to produce low-dimensional continuous feature representations. In each of the following two blocks, we use a layer and a layer for feature learning and graph coarsening, respectively. We use a multi-layer perceptron network for classification.

which results in the feature matrix $\boldsymbol{Y}^{(\ell)}$ for the line graph. Then, we perform message passing and aggregation on the line graph in Eq. (4.4). With updated edges features, Eq. (4.5) generates new nodes features with edge features $\boldsymbol{Y}^{(\ell)}$. Eq. (4.6) performs feature passing and aggregation on the graph G, which leads to aggregated nodes features $\boldsymbol{K}^{(\ell)}$. In Eq. (4.7), aggregated features from the graph $\mathbb{G}$ and the line graph $L(\mathbb{G})$ are transformed and combined, which produces the output feature matrix $\boldsymbol{X}^{(\ell+1)}$. Note that we can apply popular adjacency matrix normalization methods [14] on the adjacency matrix $\boldsymbol{A}^{(\ell)}$, the line graph adjacency matrix $\boldsymbol{F}^{(\ell)}$, and the incidence matrix $\boldsymbol{B}^{(\ell)}$.

In the WLGCL, we use the line graph structure as a complement to the original graph structure, thereby leading to enhanced feature learnings. Here, we use a simple feature aggregation method as used in GCN [14]. Other complicated and advanced feature aggregation methods such as GAT [15] can be easily applied by changing Eq. (4.4) and Eq. (4.6) accordingly. Figure 4.3 provides an illustration of our WLGCL.

4.3.4 Weighted Line Graph Convolution Layer via Incidence Matrix

In this section, we propose to implement the WLGCL using the incidence matrix, which can significantly reduce the usage of computational resources while taking advantage of the line graph structure.

One practical challenge of using a line graph structure is that it consumes excessive computational resources in terms of memory usage and execution time. To use a line graph in a graph neural network, we need to store its adjacency matrix, compute edge features, and perform message passing operation. Our proposed WLGCL also faces this challenge. Space and time complexities of Eq. (4.4), which plays the dominating role, are $O(E^2) = O(N^4)$ and $O(E^2C) = O(N^4C)$, respectively. To address this issue, we propose to use the incidence matrix $\boldsymbol{B}$ to compute the weighted line graph adjacency matrix $\boldsymbol{F}$. The adjacency matrix $\boldsymbol{F}$ can be accurately computed with the following theorem.

Theorem 2. *Given an undirected graph, its incidence matrix $\boldsymbol{B} \in \mathbb{R}^{N\times E}$, and its degree matrix $\boldsymbol{D} \in \mathbb{R}^{N}$, the adjacency matrix $\boldsymbol{F} \in \mathbb{R}^{E\times E}$ of the weighted line graph with weights defined by Eq. (4.1) can be exactly computed by*

$$\boldsymbol{F} = \boldsymbol{B}^T diag\left(\boldsymbol{D}\right)^{-1}\boldsymbol{B}, \tag{4.8}$$

where diag$(\cdot)$ takes a vector as input and constructs a squared diagonal matrix using the vector elements as the main diagonal elements.

Proof. We construct a weighted incidence matrix by normalizing the weights as $\hat{B}_{i,(i,j)} = 1/D_i$. Thus, the weighted incidence matrix is computed as $\hat{\boldsymbol{B}} = \text{diag}\left(\boldsymbol{D}\right)^{-1}\boldsymbol{B}$. In the incidence graph, each edge is connected to its two end nodes. Thus, each column in the incidence matrix $\boldsymbol{B}_{:,(a,b)}$ has two non-zero entries; those are $B_{a,(a,b)}$ and $B_{b,(a,b)}$. The same rule applies to the weighted incidence matrix $\hat{\boldsymbol{B}}$. Based on this observation, we have

$$\begin{aligned}
&\left(\boldsymbol{B}^T\hat{\boldsymbol{B}}\right)_{(a,b),(b,c)} = \sum_{i=1}^{N} B^T_{(a,b),i} \times \hat{B}_{i,(b,c)} \\
&= B^T_{(a,b),a}\hat{B}_{a,(b,c)} + B^T_{(a,b),b}\hat{B}_{b,(b,c)} \\
&= \begin{cases} \frac{1}{D_b} & \text{if } a \neq c \\ \frac{1}{D_b} + \frac{1}{D_a} & \text{if } a = c \end{cases} = F_{(a,b),(b,c)}.
\end{aligned} \tag{4.9}$$

This completes the proof. □

Based on the results from Theorem 2, we can update the equations (Eq. (4.3-4.5)) to generate

$\boldsymbol{K}_L^{(\ell)}$ in the WLGCL by replacing the adjacency matrix $\boldsymbol{F}$ with Eq. (4.8):

$$\begin{aligned}\boldsymbol{K}_L^{(\ell)} &= \boldsymbol{B}^{(\ell)}\boldsymbol{F}^{(\ell)}\boldsymbol{B}^{(\ell)^T}\boldsymbol{X}^{(\ell)} \\ &= \boldsymbol{B}^{(\ell)}\boldsymbol{B}^{(\ell)^T}\text{diag}\left(\boldsymbol{D}\right)^{-1}\boldsymbol{B}^{(\ell)}\boldsymbol{B}^{(\ell)^T}\boldsymbol{X}^{(\ell)} \\ &= \boldsymbol{H}^{(\ell)}\text{diag}\left(\boldsymbol{D}\right)^{-1}\boldsymbol{H}^{(\ell)}\boldsymbol{X}^{(\ell)},\end{aligned} \tag{4.10}$$

where $\boldsymbol{H}^{(\ell)} = \boldsymbol{B}^{(\ell)}\boldsymbol{B}^{(\ell)^T}$ only needs to be computed once. With computed $\boldsymbol{K}_L^{(\ell)}$, we output the new feature matrix $\boldsymbol{X}^{(\ell+1)}$ using equations Eq. (4.6) and Eq. (4.7).

By using the implementation in Eq. (4.10), space and time complexities of the WLGCL are reduced to $O(N \times E) = O(N^3)$ and $O(N^2 \times E) + O(N^2 \times C) = O(N^4)$, respectively. Compared to the naive WLGCL implementation, they are reduced by a factor of N and C, respectively. In the experimental study part, we show that the WLGCL implemented as Eq. (4.10) dramatically saves the computational resources compared to the naive implementation. Notably, the results in Eq. (4.8) can be applied to other graph neural networks that leverage the benefits of line graph structures.

4.3.5 Weighted Line Graph Convolutional Networks

In this section, we build a family of weighted line graph convolutional networks (WLGCNets) that utilize our proposed WLGCLs. In WLGCNets, an embedding layer such as a fully-connected layer or GCN layer is firstly used to learn low-dimensional representations for nodes in the graph. Then we stack multiple blocks, each of which consists of a WLGCL and a pooling layer [67]. Here, the WLGCL encodes high-level features while the pooling layer outputs a coarsened graph. We use the gPool layer to produce a coarsened graph that helps to retain original graph structure information. To deal with the variety of graph sizes in terms of the number of nodes, we apply global readout operations on the outputs including maximization, averaging and summation [69]. The outputs of the first GCN layer and all blocks are stacked together in the feature dimension and fed into a multi-layer perceptron network for classification. Figure 4.4 provides an example of our WLGCNets.

Table 4.1: Comparison of WLGCNet and previous state-of-the-art models on graph classification datasets. We compare our networks with WL [70], PSCN [18], DGCNN, SAGPool [71], DIFF-POOL, g-U-Net, and GIN. We report the graph classification accuracies (%) on PROTEINS, D&D, IMDB-MULTI, REDDIT-BINARY, REDDIT-MULTI5K, COLLAB, and REDDIT-MULTI12K datasets.

	PROTEINS	**D&D**	**IMDBM**	**RDTB**	**RDT5K**	**COLLAB**	**RDT12K**
graphs	1113	1178	1500	2000	4999	5000	11929
nodes	39.1	284.3	13	429.6	508.5	74.5	391.4
classes	2	2	3	2	5	3	11
WL	75.0 ± 3.1	78.3 ± 0.6	50.9 ± 3.8	81.0 ± 3.1	52.5 ± 2.1	78.9 ± 1.9	44.4 ± 2.1
DGCNN	75.5 ± 0.9	79.4 ± 0.9	47.8 ± 0.9	-	-	73.8 ± 0.5	41.8 ± 0.6
PSCN	75.9 ± 2.8	76.3 ± 2.6	45.2 ± 2.8	86.3 ± 1.6	49.1 ± 0.7	72.6 ± 2.2	41.3 ± 0.8
DIFFPOOL	76.3	80.6	-	-	-	75.5	47.1
SAGPool	71.9	76.5	-	-	-	-	-
g-U-Net	77.6 ± 2.6	82.4 ± 2.9	51.8 ± 3.7	85.5 ± 1.3	48.2 ± 0.8	77.5 ± 2.1	44.5 ± 0.6
GIN	76.2 ± 2.8	82.0 ± 2.7	52.3 ± 2.8	92.4 ± 2.5	57.5 ± 1.5	80.6 ± 1.9	-
WLGCNet	$\mathbf{78.9 \pm 4.2}$	$\mathbf{83.8 \pm 2.8}$	$\mathbf{56.1 \pm 3.6}$	$\mathbf{94.1 \pm 2.2}$	$\mathbf{58.2 \pm 3.2}$	$\mathbf{83.1 \pm 7.9}$	$\mathbf{50.3 \pm 1.5}$

4.4 Experimental Study

In this section, we evaluate our proposed WLGCL and WLGCNet on graph classification tasks. We demonstrate the effectiveness of our methods by comparing our networks with previous state-of-the-art models in terms of the graph classification accuracy. The performances on small datasets show that our methods will not increase the risk of the over-fitting problem. We conduct ablation experiments to demonstrate the contributions of our methods. Besides, we evaluate the efficiency of our implementation of the WLGCL in terms of the usage of computational resources.

4.4.1 Experimental Setup

We describe the experimental setup for graph classification tasks. In this work, we mainly evaluate our methods on graph classification datasets such as social network datasets and bioinformatics datasets. The node features are created using one-hot encodings and fed into the networks. In WLGCNets, we use GCN layers as the graph embedding layers. After the first GCN layer, we stack three blocks as described in Section 4.3.5. The outputs of the GCN layer and WLGCLs in three blocks are processed by a readout function and concatenated as the network output. The readout function performs three global pooling operations; those are maximization, averaging, and summation. The network outputs are fed into a classifier to produce predictions. Here, we use a two-layer feed-forward network with 512 units in the hidden layer as the classifier. We apply dropout [23] on the network and the classifier.

We use an Adam optimizer [24] with a learning rate of 0.001 to train WLGCNets. To prevent over-fitting, we apply the L_2 regularization on trainable parameters with a weight decay rate of 0.0008. All models are trained for 200 epochs using one NVIDIA GeForce RTX 2080 Ti GPU on an Ubuntu 18.04 system.

4.4.2 Performance Study

To evaluate our methods and WLGCNets, we conduct experiments on graph classification tasks using seven datasets; those are IMDB-BINARY (IMDBB), D&D [45], IMDB-MULTI (IMDBM), REDDIT-BINARY (RDTB), REDDIT-MULTI5K (RDT5K), COLLAB, and REDDIT-MULTI12K (RDT12K) [46]. REDDIT datasets are benchmarking large graph datasets used for evaluating graph neural networks in the community. On the datasets without node features such as RDT12K, we use one-hot encodings of node degrees as node features [69]. To produce less biased evaluation results, we follow the practices in [69, 47] and perform 10-fold cross-validation on training datasets. We use the average accuracy across 10 fold testing results with variances.

We report the graph classification accuracy along with performances of previous state-of-the-art models. The results are summarized in Table 4.1. We can observe from the results that our proposed WLGCNets significantly outperform previous models by margins of 3.4%, 1.8%, 3.8%, 1.7%, 0.7%, 2.5%, 3.2% on IMDB-BINARY, D&D, IMDB-MULTI, REDDIT-BINARY, REDDIT-MULTI5K, COLLAB, and REDDIT-MULTI12K datasets, respectively. The promising results, especially on large benchmarking datasets such as REDDIT-MULTI12K, demonstrate the effectiveness of our proposed methods and models for network embeddings. Note that our WLGCNet uses the gPool layer from the g-U-Net. The superior performances of WLGCNets over the g-U-Net demonstrate the performance gains are from our proposed WLGCLs.

Table 4.2: Comparison of WLGCNet and previous state-of-the-art models on relatively small datasets. We report the graph classification accuracies (%) on MUTAG, PTC, and PROTEINS datasets.

	MUTAG	**PTC**	**PROTEINS**
graphs	188	344	1113
nodes	17.9	25.5	39.1
classes	2	2	2
WL	90.4 ± 5.7	59.9 ± 4.3	75.0 ± 3.1
DGCNN	85.8 ± 1.7	58.6 ± 2.4	75.5 ± 0.9
PSCN	92.6 ± 4.2	60.0 ± 4.8	75.9 ± 2.8
DIFFPOOL	-	-	76.3
SAGPool	-	-	71.9
g-U-Net	87.2 ± 7.8	64.7 ± 6.8	77.6 ± 2.6
GIN	90.0 ± 8.8	64.6 ± 7.0	76.2 ± 2.8
WLGCNet	$\mathbf{93.0 \pm 5.8}$	$\mathbf{72.7 \pm 6.0}$	$\mathbf{78.9 \pm 4.2}$

4.4.3 Results on Small Datasets

In the previous section, we evaluate our methods on benchmarking datasets that are relatively large in terms of the number of graphs and the number of nodes in graphs. To provide a comprehensive evaluation, we conduct experiments on relatively small datasets to evaluate the risk of over-fitting of our methods. Here, we use three datasets; those are MUTAG [72], PTC [73], and PROTEINS [44]. These datasets are bioinformatics datasets with categorical features on nodes. We follow the same experimental settings as in Section 4.4.2.

The results in terms of the graph classification accuracy are summarized in Table 4.2 with performances of previous state-of-the-art models. We can observe from the results that our WLGCNet outperforms previous models by margins of 0.4%, 6.0%, and 1.3% on MUTAG, PTC, and PROTEINS, respectively. This demonstrates that our proposed models that take advantage of line graph structures will not increase the risk of the over-fitting problem even on small datasets.

4.4.4 Ablation Study of Weighted Line Graph Convolution Layers

In this section, we conduct ablation studies based on WLGCNets to demonstrate the contribution of our WLGCLs to the entire network. To explore the advantage of line graph structures,

Table 4.3: Ablation study of WLGCNet. We compare WLGCNet with the network using the same architecture as WLGCNet with GCN layers (denoted as WLGCNet$_g$), the network using the same architecture as WLGCNet with regular line graph convolution layers (denoted as WLGCNet$_l$). We report the graph classification accuracies (%) on REDDIT-BINARY, REDDIT-MULTI5K, and REDDIT-MULTI12K datasets.

	RDTB	**RDT5K**	**RDT12K**
WLGCNet$_g$	93.2 ± 1.5	56.9 ± 2.2	49.1 ± 1.5
WLGCNet$_l$	93.6 ± 2.0	57.3 ± 3.0	49.6 ± 2.8
WLGCNet	**94.1 ± 2.2**	**58.2 ± 3.2**	**50.3 ± 1.5**

we construct a network that removes all layers using line graphs. Based on the WLGCNet, we replace WLGCLs by GCNs using the same number of trainable parameters, which we denote as WLGCNet$_g$. To compare our weighted line graph with the regular line graph, we modify our WLGCLs to use regular line graph structures. We denote the resulting network as WLGCNet$_l$. We evaluate these networks on three datasets; those are REDDIT-BINARY, REDDIT-MULTI5K, and REDDIT-MULTI12K datasets.

The results in terms of the graph classification accuracy are summarized in Table 4.3. We can observe from the results that both WLGCNet and WLGCNet$_l$ achieve better performances than WLGCNet$_g$, which demonstrates the benefits of utilizing line graph structures on graph neural networks. When comparing WLGCNet with WLGCNet$_l$, WLGCNet outperforms WLGCNet$_l$ by margins of 0.5%, 0.5%, and 0.7% on REDDIT-BINARY, REDDIT-MULTI5K, and REDDIT-MULTI12K datasets, respectively. This indicates that our proposed WLGCL utilizes weighted line graph structures with unbiased topology information encoded, thereby leading to better performances.

4.4.5 Computational Efficiency Study

In Section 4.3.4, we propose an efficient implementation of WLGCL using the incidence matrix, which can save dramatic computational resources compared to the naive one. Here, we conduct experiments on simulated data to evaluate the efficiency of our methods. To this end, we build networks that contain a single layer. This helps to remove the influence of other expected fac-

Table 4.4: Comparison of WLGCL and the WLGCL using native implementation (denoted as WLGCL$_n$). We evaluate them on simulated data with different graph sizes in terms of the number of nodes and the number of edges. All layers output 64 feature channels. We report the number of multiply-adds (MAdd), the amount of memory usage, and the CPU execution time. We describe the input graph size in the format of "number of nodes / number of edges".

Input	Operator	MAdd	Saving	Memory	Saving	Time	Speedup
$1000/50000$	WLGCL$_n$	166.47B	0.00%	9.5GB	0.00%	15.73s	1.0×
	WLGCL	51.13B	69.28%	0.19GB	97.94%	0.63s	26.2×
$1000/100000$	WLGCL$_n$	652.87B	0.00%	37.63GB	0.00%	56.62s	1.0×
	WLGCL	101.13B	84.51%	0.38GB	98.98%	1.21s	47.2×
$1000/150000$	WLGCL$_n$	1,459.27B	0.00%	86.71GB	0.00%	134.36s	1.0×
	WLGCL	151.13B	89.64%	0.57GB	99.34%	1.82s	74.6×
$2000/150000$	WLGCL$_n$	1,478.66B	0.00%	99.87GB	0.00%	278.8s	1.0×
	WLGCL	608.52B	58.85%	1.13GB	98.86%	6.19s	45.1×

tors. To fully explore the efficiency of our proposed methods, we conduct experiments on graphs of different sizes in terms of the number of nodes. Since WLGCL takes advantage of line graph structures, the graph density has a significant impact on the layer efficiency. Here, the graph density is defined by $2E/(N \times (N-1))$. To investigate the impact of the graph density, we conduct experiments on graphs with the same size but different numbers of edges.

By using the TensorFlow profile tool [50], we report the computational resources used by networks including the number of multiply-adds (MAdd), the amount of memory usage, and the CPU execution time. The comparison results are summarized in Table 4.4. We can observe from the results that the WLGCLs with our proposed implementation use significantly less computational resources than WLGCLs with naive implementation in terms of the memory usage and CPU execution time. By comparing the results on first three inputs, the advantage on efficiency of our method over the naive implementation becomes much larger as the increase of the graph density with the same graph size. By comparing results of the last two inputs with the same number of edges but different graph sizes, we can observe that the efficiency advantage of our proposed methods remains the same. This shows that the graph density is a key factor that influences the usage of computational resources, especially on dense graphs.

The experimental results on simulated data demonstrate that our implementation of WLGCL

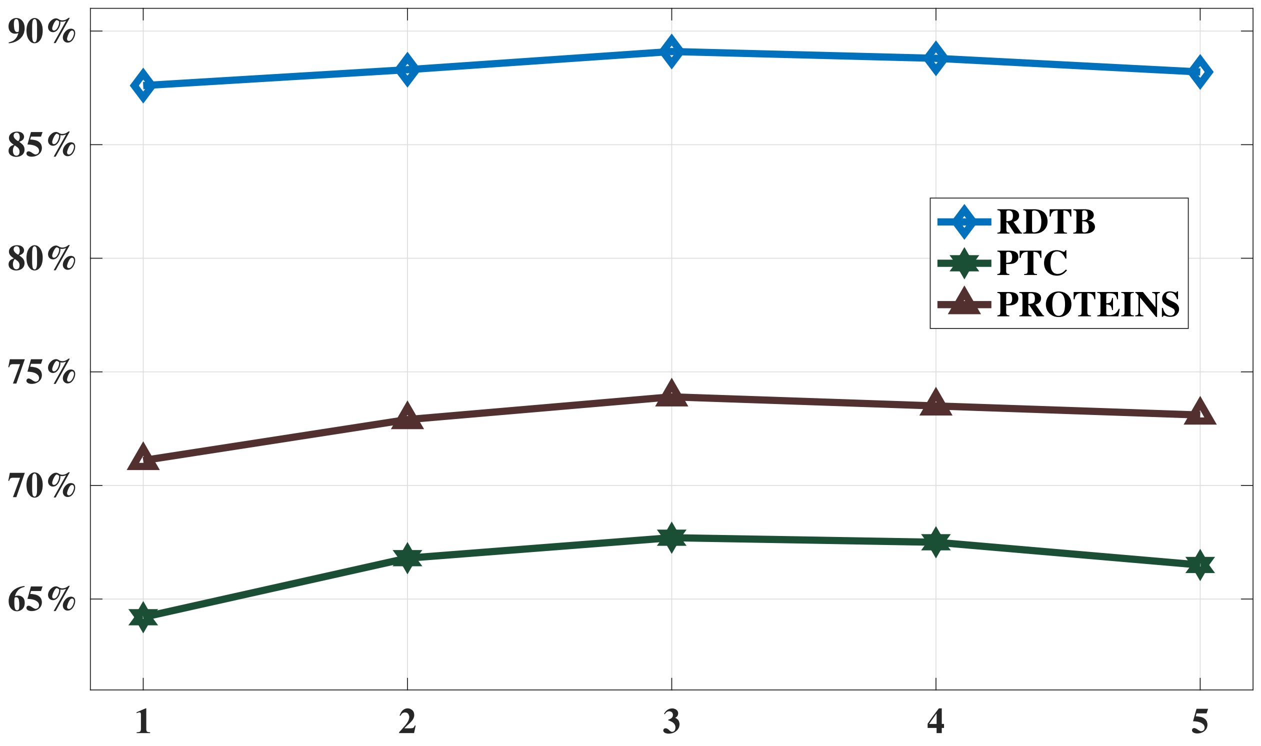

Figure 4.5: Comparison of WLGCNets with different depths. We evaluate these networks on PTC, PROTEINS, and REDDIT-BINARY datasets. We report the graph classification accuracies in this figure.

via the incidence matrix can effectively alleviate the inefficiency issue in graph neural networks that leverage line graph structures.

4.4.6 Network Depth Study

Network depth in terms of the number of blocks is an important hyper-parameter in the WLGCNet. In previous experiments, we use three blocks in WLGCNets based on our empirical experiences. In this section, we investigate the impact of the network depth in WLGCNets on network embeddings. Based on our WLGCNet, we vary the network depth from 1 to 5, which covers a reasonable range. We evaluate these networks on PTC, PROTEINS, and REDDIT-BINARY datasets and report the graph classification accuracies. Figure 4.5 plots the results of WLGCNets with different numbers of blocks. We can observe from the figure that the best performances are achieved on WLGCNets with three blocks on all three datasets. When the network depth increases, the performances decrease, which indicates the over-fitting problem in deeper networks.

5. FEATURE-AWARE GRAPH STRUCTURE LEARNING

In the previous sections, we discussed how to learn high-level features without changing graph structures. In this section, we move forward to discuss graph deep learning methods that learn new graph structures[1].

5.1 Introduction

Convolutional neural networks (CNNs) [42] have demonstrated great capability in various challenging artificial intelligence tasks, especially in fields of computer vision [5, 19] and natural language processing [16]. One common property behind these tasks is that both images and texts have grid-like structures. Elements on feature maps have locality and order information, which enables the application of convolutional operations [27].

In practice, many real-world data can be naturally represented as graphs such as social and biological networks. Due to the great success of CNNs on grid-like data, applying them on graph data [53, 54] is particularly appealing. Recently, there have been many attempts to extend convolutions to graph data (GNNs) [14, 15, 49]. One common use of convolutions on graphs is to compute node representations [28, 52]. With learned node representations, we can perform various tasks on graphs such as node classification and link prediction.

Images can be considered as special cases of graphs, in which nodes lie on regular 2D lattices. It is this special structure that enables the use of convolution and pooling operations on images. Based on this relationship, node classification and embedding tasks have a natural correspondence with pixel-wise prediction tasks such as image segmentation [74, 75, 58]. In particular, both tasks aim to make predictions for each input unit, corresponding to a pixel on images or a node in graphs. In the computer vision field, pixel-wise prediction tasks have achieved major advances recently. Encoder-decoder architectures like the U-Net [76] are state-of-the-art methods for these tasks. It is thus

[1]Reprinted with permission from "Graph U-Nets." by Hongyang Gao and Shuiwang Ji, 2019, *International Conference on Machine Learning*, vol. 1, pp. 2083-2092, Copyright 2019 by PMLR, and "Learning graph pooling and hybrid convolutional operations for text representations." by Hongyang Gao, Yongjun Chen, and Shuiwang Ji, 2019, *The World Wide Web Conference*, vol. 1, pp. 2743-2749, Copyright 2019 by ACM.

highly interesting to develop U-Net-like architectures for graph data. In addition to convolutions, pooling and up-sampling operations are essential building blocks in these architectures. However, extending these operations to graph data is highly challenging. Unlike grid-like data such as images and texts, nodes in graphs have no spatial locality and order information as required by regular pooling operations.

To bridge the above gap, we propose novel graph pooling (gPool) and unpooling (gUnpool) operations in this section. Based on these two operations, we propose U-Net-like architectures for graph data. The gPool operation samples some nodes to form a smaller graph based on their scalar projection values on a trainable projection vector. As an inverse operation of gPool, we propose a corresponding graph unpooling (gUnpool) operation, which restores the graph to its original structure with the help of locations of nodes selected in the corresponding gPool layer. Based on the gPool and gUnpool layers, we develop graph U-Nets, which allow high-level feature encoding and decoding for network embedding. Experimental results on node classification and graph classification tasks demonstrate the effectiveness of our proposed methods as compared to previous methods.

In this section, we also work on graphs converted from text data, the words are treated as nodes in the graphs. By maintaining the order information in nodes' feature matrices, we can apply convolutional operations to feature matrices. Based on this observation, we develop a new graph convolutional layer, known as the hybrid convolutional (hConv) layer. Based on gPool and hConv layers, we develop a shallow but effective architecture for text modeling tasks [77]. Results on text classification tasks demonstrate the effectiveness of our proposed methods as compared to previous CNN models.

5.2 Related Work

In this section, we discuss some graph structure learning operations and their applications on text data.

5.2.1 Graph Structure Learning Operations

Recently, there has been a rich line of research on graph neural networks [78]. Inspired by the first order graph Laplacian methods, [14] proposed graph convolutional networks (GCNs), which achieved promising performance on graph node classification tasks. GCNs essentially perform aggregation and transformation on node features without learning trainable filters. [28] tried to sample a fixed number of neighboring nodes to keep the computational footprint consistent. [15] proposed to use attention mechanisms to enable different weights for neighboring nodes. [79] used relational graph convolutional networks for link prediction and entity classification. Some studies applied GNNs to graph classification tasks [80, 81, 47]. [82] discussed possible ways of applying deep learning on graph data. [83] and [84] proposed to use spectral networks for large-scale graph classification tasks. Some studies also applied graph kernels on traditional computer vision tasks [85, 86, 87].

In addition to convolution, some studies tried to extend pooling operations to graphs. [27] proposed to use binary tree indexing for graph coarsening, which fixes indices of nodes before applying 1-D pooling operations. [88] used deterministic graph clustering algorithm to determine pooling patterns. [52] used an assignment matrix to achieve pooling by assigning nodes to different clusters of the next layer.

5.2.2 GCN Applications on Text Modeling

Before applying graph-based methods on text data, we need to convert texts to graphs. In this section, we discuss related literatures on converting texts to graphs and use of GCNs on text data.

Many graph representations of texts have been explored to capture the inherent topology and dependence information between words. In [89], a rule-based classifier is employed to map each tag onto graph nodes and edges. The tags are acquired by the part-of-speech (POS) tagging techniques. In [90] a concept interaction graph representation is proposed for capturing complex interactions among sentences and concepts in documents. The graph-of-word representation (GoW) [91] attempts to capture co-occurrence relationships between words known as terms.

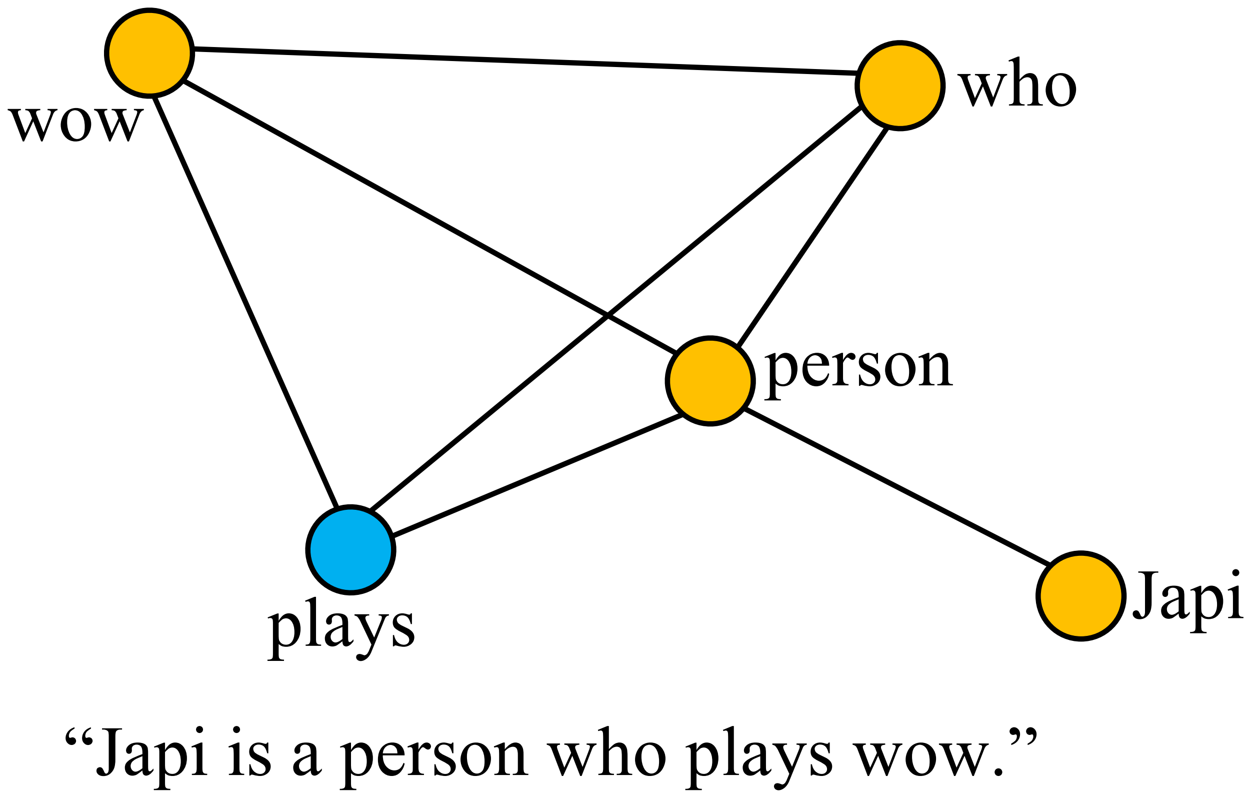

Figure 5.1: Example of converting text to a graph using the graph-of-words method. For this text, we use noun, adjective, and verb as terms for node selection. The words of “Japi", “person", “who", “plays", and “wow" are selected as nodes in the graph. We employ a sliding window size of 4 for edge building. For instance, there is an undirected edge between “Japi” and “person”, since they can be covered in the same sliding window in the original text.

It was initially applied to text ranking task and has been widely used in many NLP tasks such as information retrieval [92], text classification [93, 94], keyword extraction [95, 96] and document similarity measurement [97].

Before applying graph-based text modeling methods, we need to convert texts to graphs. Here, we employ the graph-of-words [91] method for its effectiveness and simplicity. The conversion starts with the phase preprocessing such as tokenization and text cleaning. After preprocessing, each text is encoded into an unweighted and undirected graph in which nodes represent selected terms and edges represent co-occurrences between terms within a fixed sliding window. A term is a group of words clustered based on their part-of-speech tags such as noun and adjective. The choice of sliding window size depends on the average lengths of processed texts. Figure 5.1 provides an example of this method.

In addition to graph data, some studies attempted to apply graph-based methods to grid-like data such as texts. Compared to traditional recurrent neural networks such as LSTM [33], GCNs have the advantage of considering long-term dependencies by edges in graphs. In [98], a variant of GCNs is applied to the task of sentence encoding and achieved better performance than LSTM. GCNs have also been used in neural machine translation tasks [99]. Although graph convolutional operations have been extensively developed and explored, pooling operations on graphs are not well studied currently.

5.3 Structure Learning Operations

In this section, we introduce the graph pooling (gPool) layer, graph unpooling (gUnpool) layer, and hybrid convolutional layer. Based on these new layers, we develop the graph U-Nets for node classification tasks and graph networks for graph classifications.

5.3.1 Graph Pooling Layer

Pooling layers play important roles in CNNs on grid-like data. They can reduce sizes of feature maps and enlarge receptive fields, thereby giving rise to better generalization and performance [100]. On grid-like data such as images, feature maps are partitioned into non-overlapping rectangles, on which non-linear down-sampling functions like maximum are applied. In addition to local pooling, global pooling layers [101] perform down-sampling operations on all input units, thereby reducing each feature map to a single number. In contrast, k-max pooling layers [102] select the k-largest units out of each feature map.

However, we cannot directly apply these pooling operations to graphs. In particular, there is no locality information among nodes in graphs. Thus the partition operation is not applicable on graphs. The global pooling operation will reduce all nodes to one single node, which restricts the flexibility of networks. The k-max pooling operation outputs the k-largest units that may come from different nodes in graphs, resulting in inconsistency in the connectivity of selected nodes.

In this section, we propose the graph pooling (gPool) layer to enable down-sampling on graph data. In this layer, we adaptively select a subset of nodes to form a new but smaller graph. To this end, we employ a trainable projection vector $\mathbf{p}$. By projecting all node features to 1D, we can perform k-max pooling for node selection. Since the selection is based on 1D footprint of each node, the connectivity in the new graph is consistent across nodes. Given a node i with its feature vector $\mathbf{x}_i$, the scalar projection of $\mathbf{x}_i$ on $\mathbf{p}$ is $y_i = \mathbf{x_i}\mathbf{p}/\|\mathbf{p}\|$. Here, y_i measures how much information of node i can be retained when projected onto the direction of $\mathbf{p}$. By sampling nodes, we wish to preserve as much information as possible from the original graph. To achieve this, we select nodes with the largest scalar projection values on $\mathbf{p}$ to form a new graph.

Suppose there are N nodes in a graph $\mathbb{G}$ and each of which contains C features. The graph can be represented by two matrices; those are the adjacency matrix $A^\ell \in \mathbb{R}^{N\times N}$ and the feature matrix $X^\ell \in \mathbb{R}^{N\times C}$. Each non-zero entry in the adjacency matrix A represents an edge between two nodes in the graph. Each row vector $\mathbf{x}_i^\ell$ in the feature matrix X^ℓ denotes the feature vector of node i in the graph. The layer-wise propagation rule of the graph pooling layer ℓ is defined as:

$$
\begin{aligned}
\mathbf{y} &= X^\ell \mathbf{p}^\ell / \|\mathbf{p}^\ell\|, \\
\text{idx} &= \text{rank}(\mathbf{y}, k), \\
\tilde{\mathbf{y}} &= \text{sigmoid}(\mathbf{y}(\text{idx})), \\
\tilde{X}^\ell &= X^\ell(\text{idx}, :), \\
A^{\ell+1} &= A^\ell(\text{idx}, \text{idx}), \\
X^{\ell+1} &= \tilde{X}^\ell \odot \left(\tilde{\mathbf{y}}\mathbf{1}_C^T\right),
\end{aligned}
\tag{5.1}
$$

where k is the number of nodes selected in the new graph. $\text{rank}(\mathbf{y}, k)$ is the operation of node ranking, which returns indices of the k-largest values in $\mathbf{y}$. The idx returned by $\text{rank}(\mathbf{y}, k)$ contains the indices of nodes selected for the new graph. $A^\ell(\text{idx}, \text{idx})$ and $X^\ell(\text{idx}, :)$ perform the row and/or column extraction to form the adjacency matrix and the feature matrix for the new graph. $\mathbf{y}(\text{idx})$ extracts values in $\mathbf{y}$ with indices idx followed by a sigmoid operation. $\mathbf{1}_C \in \mathbb{R}^C$ is a vector of size C with all components being 1, and $\odot$ represents the element-wise matrix multiplication.

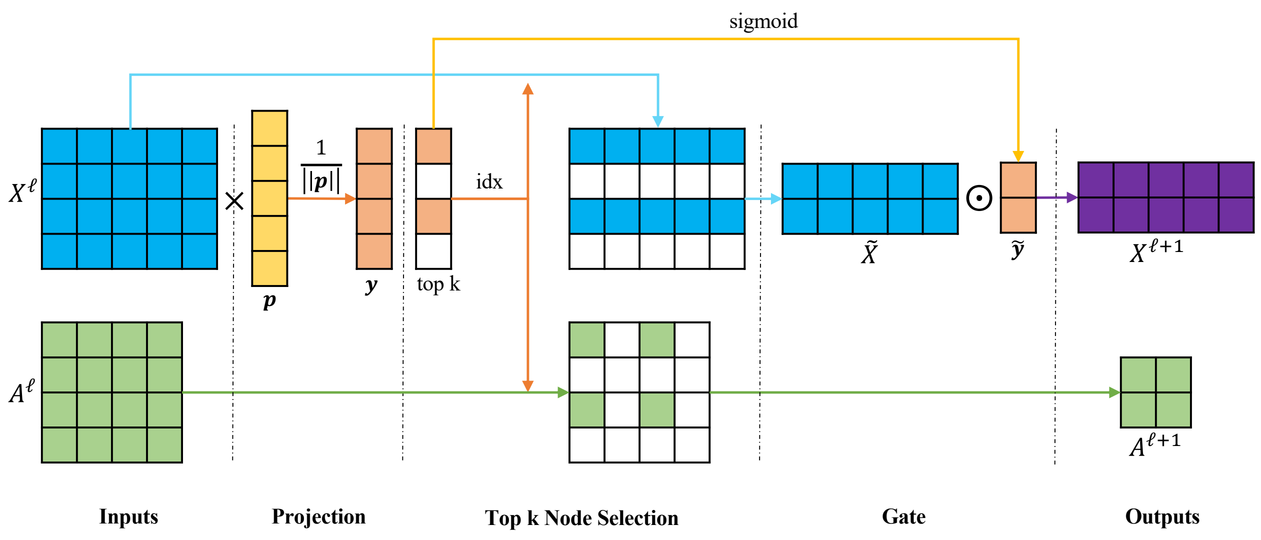

Figure 5.2: An illustration of the proposed graph pooling layer with $k = 2$. $\times$ and $\odot$ denote matrix multiplication and element-wise product, respectively. We consider a graph with 4 nodes, and each node has 5 features. By processing this graph, we obtain the adjacency matrix $A^\ell \in \mathbb{R}^{4\times4}$ and the input feature matrix $X^\ell \in \mathbb{R}^{4\times5}$ of layer ℓ. In the projection stage, $\mathbf{p} \in \mathbb{R}^5$ is a trainable projection vector. By matrix multiplication and sigmoid$(\cdot)$, we obtain $\mathbf{y}$ that are scores estimating scalar projection values of each node to the projection vector. By using $k = 2$, we select two nodes with the highest scores and record their indices in the top-k-node selection stage. We use the indices to extract the corresponding nodes to form a new graph, resulting in the pooled feature map $\tilde{X}^\ell$ and new corresponding adjacency matrix $A^{\ell+1}$. At the gate stage, we perform element-wise multiplication between $\tilde{X}^\ell$ and the selected node scores vector $\tilde{\mathbf{y}}$, resulting in $X^{\ell+1}$. This graph pooling layer outputs $A^{\ell+1}$ and $X^{\ell+1}$.

X^ℓ is the feature matrix with row vectors $\mathbf{x}_1^\ell, \mathbf{x}_2^\ell, \cdots, \mathbf{x}_N^\ell$, each of which corresponds to a node in the graph. We first compute the scalar projection of X^ℓ on $\mathbf{p}^\ell$, resulting in $\mathbf{y} = [y_1, y_2, \cdots, y_N]^T$ with each y_i measuring the scalar projection value of each node on the projection vector $\mathbf{p}^\ell$. Based on the scalar projection vector $\mathbf{y}$, rank$(\cdot)$ operation ranks values and returns the k-largest values in $\mathbf{y}$. Suppose the k-selected indices are $i_1, i_2, \cdots, i_k$ with $i_m < i_n$ and $1 \leq m < n \leq k$. Note that the index selection process preserves the position order information in the original graph. With indices idx, we extract the adjacency matrix $A^\ell \in \mathbb{R}^{k\times k}$ and the feature matrix $\tilde{X}^\ell \in \mathbb{R}^{k\times C}$ for the new graph. Finally, we employ a gate operation to control information flow. With selected indices idx, we obtain the gate vector $\tilde{y} \in \mathbb{R}^k$ by applying sigmoid to each element in the extracted scalar projection vector. Using element-wise matrix product of $\tilde{X}^\ell$ and $\tilde{\mathbf{y}}\mathbf{1}_C^T$, information of selected nodes is controlled. The ith row vector in $X^{\ell+1}$ is the product of the ith row vector in X^{ℓ} and the ith scalar value in $\tilde{y}$.

Notably, the gate operation makes the projection vector $\mathbf{p}$ trainable by back-propagation [42]. Without the gate operation, the projection vector $\mathbf{p}$ produces discrete outputs, which makes it not trainable by back-propagation. Figure 5.2 provides an illustration of our proposed graph pooling layer. Compared to pooling operations used in grid-like data, our graph pooling layer employs extra training parameters in projection vector $\mathbf{p}$. We will show that the extra parameters are negligible but can boost performance.

In our proposed gPool layer, we sample some important nodes to form a new graph for high-level feature encoding. Since related edges are removed when removing nodes in gPool, the nodes in the pooled graph might become isolated. This may influence the information propagation in subsequent layers, especially when GCN layers are used to aggregate information from neighboring nodes. We need to increase connectivity among nodes in the pooled graph. To address this problem, we propose to use the k^{th} graph power $\mathbb{G}^k$ to increase the graph connectivity. This operation builds links between nodes whose distances are at most k hops [103]. In this part, we employ $k = 2$ since there is a GCN layer before each gPool layer to aggregate information from its first-order neighboring nodes. Formally, we replace the fifth equation in Eq 5.1 by:

$$\begin{aligned} A^2 &= A^{\ell} A^{\ell}, \\ A^{\ell+1} &= A^2(\mathrm{idx}, \mathrm{idx}), \end{aligned} \tag{5.2}$$

where $A^2 \in \mathbb{R}^{N \times N}$ is the 2^{nd} graph power. Now, the graph sampling is performed on the augmented graph with better connectivity.

5.3.2 Graph Unpooling Layer

Up-sampling operations are important for encoder-decoder networks such as U-Net. The encoders of networks usually employ pooling operations to reduce feature map size and increase receptive field. While in decoders, feature maps need to be up-sampled to restore their original

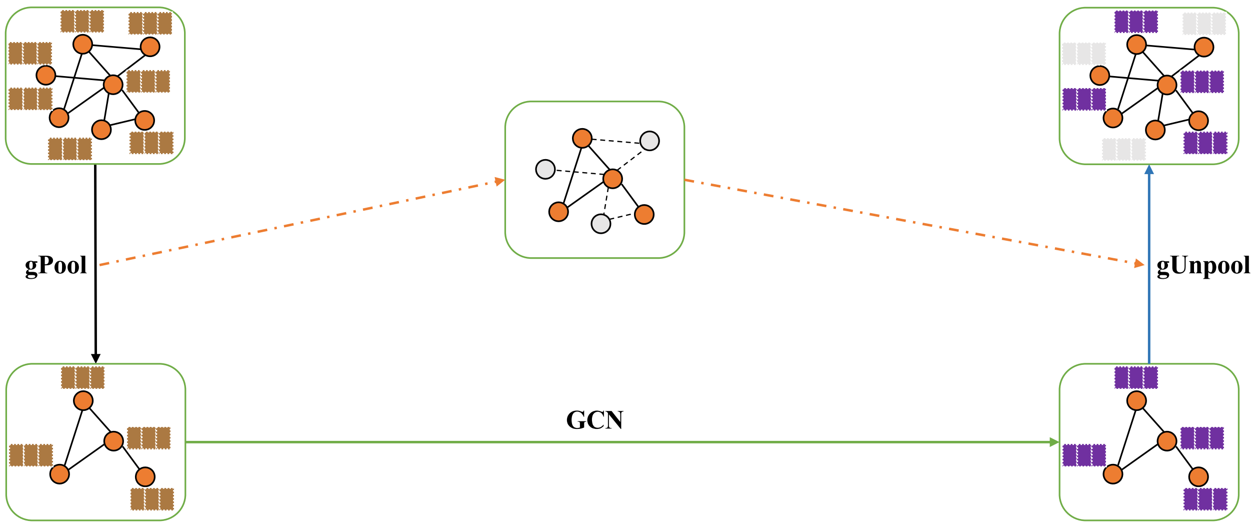

Figure 5.3: An illustration of the proposed graph unpooling (gUnpool) layer. In this example, a graph with 7 nodes is down-sampled using a gPool layer, resulting in a coarsened graph with 4 nodes and position information of selected nodes. The corresponding gUnpool layer uses the position information to reconstruct the original graph structure by using empty feature vectors for unselected nodes.

resolutions. On grid-like data like images, there are several up-sampling operations such as the deconvolution [104, 105] and unpooling layers [77]. However, such operations are not currently available on graph data.

To enable up-sampling operations on graph data, we propose the graph unpooling (gUnpool) layer, which performs the inverse operation of the gPool layer and restores the graph into its original structure. To achieve this, we record the locations of nodes selected in the corresponding gPool layer and use this information to place nodes back to their original positions in the graph. Formally, we propose the layer-wise propagation rule of graph unpooling layer as

$$X^{\ell+1} = \text{distribute}(0_{N\times C}, X^{\ell}, \text{idx}), \tag{5.3}$$

where $\text{idx} \in \mathbb{Z}^{*k}$ contains indices of selected nodes in the corresponding gPool layer that reduces the graph size from N nodes to k nodes. $X^{\ell} \in \mathbb{R}^{k\times C}$ are the feature matrix of the current graph, and $0_{N\times C}$ are the initially empty feature matrix for the new graph. $\text{distribute}(0_{N\times C}, X^{\ell}, \text{idx})$ is the operation that distributes row vectors in X^ℓ into $0_{N\times C}$ feature matrix according to their corresponding indices stored in idx. In $X^{\ell+1}$, row vectors with indices in idx are updated by row vectors in X^ℓ, while other row vectors remain zero. Figure 5.3 provides an example of our proposed gUnpool layer. Like U-Net on images, we use skip connections to fuse features of the same graph structure from encoder part for low-level spatial information, then employ a GCN layer to populate empty feature vectors in $X^{\ell+1}$.

5.3.3 Hybrid Convolutional Layer

It follows from the analysis in Section 5.2.2 that GCN layers only perform convolutional operations on each node. There is no trainable spatial filters as in regular convolution layers. GCNs do not have the power of automatic feature extraction as achieved by CNNs. This limits the capability of GCNs, especially in the field of graph modeling. In traditional graph data, there is no ordering information among nodes. In addition, the different numbers of neighbors for each node in the graph prohibit convolutional operations with a kernel size larger than 1. Although we attempt to modeling texts as graph data, they are essentially grid-like data with order information among nodes, thereby enabling the application of regular convolutional operations.

To take advantage of convolutional operations with trainable filters, we propose the hybrid convolutional layer (hConv), which combines GCN operations and regular 1-D convolutional operations to achieve the capability of automatic feature extraction. Formally, we propose the hConv layer to be defined as

$$\begin{aligned} X_1^{\ell+1} &= \text{conv}(X^\ell), \\ X_2^{\ell+1} &= \text{gcn}(A^\ell, X^\ell), \\ X^{\ell+1} &= \left[X_1^{\ell+1}, X_2^{\ell+1}\right], \end{aligned} \tag{5.4}$$

where conv$(\cdot)$ denotes a regular 1-D convolutional operation, and the gcn$(\cdot,\cdot)$ operation is defined in Eq. 2.1. For the feature matrix X^ℓ, we treat the column dimension as the channel dimension, such that the 1-D convolutional operation can be applied along the row dimension. Using the 1-D convolutional operation and the GCN operation, we obtain two intermediate outputs; those are

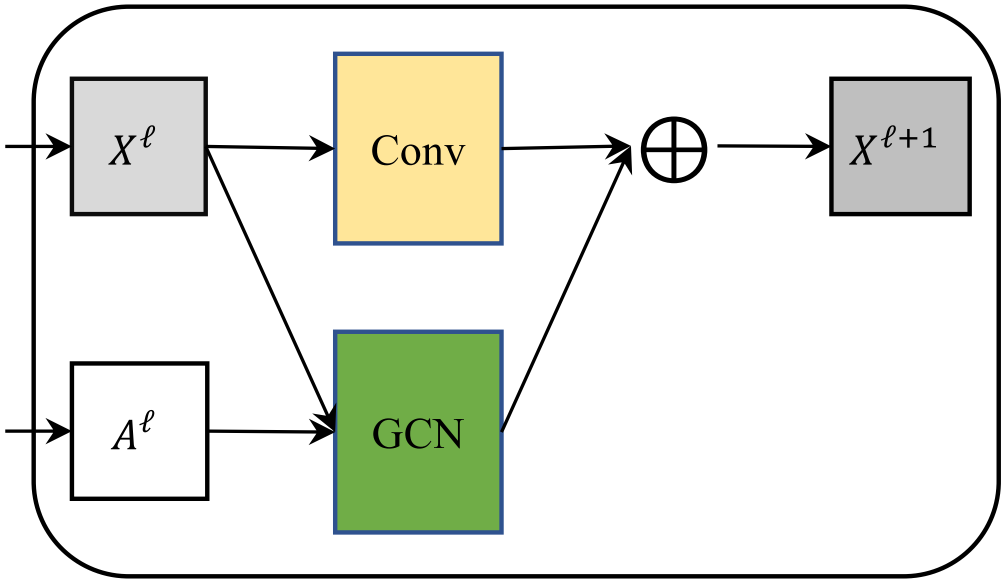

Figure 5.4: An illustration of the hybrid convolutional layer. $\oplus$ denotes matrix concatenation along the row dimension. In this layer, A^ℓ and X^ℓ are the adjacency matrix and the node feature matrix, respectively. A regular 1-D convolutional operation is used to extract high-level features from sentence texts. The GCN operation is applied at the graph level for feature extraction. The two intermediate outputs are concatenated together to form the final output $X^{\ell+1}$.

$X_1^{\ell+1}$ and $X_2^{\ell+1}$. These two matrices are concatenated together as the layer output $X^{\ell+1}$. Figure 5.4 illustrates an example of the hConv layer.

We argue that the integration of GCN operations and 1-D convolutional operations in the hConv layer is especially applicable to graph data obtained from texts. By representing texts as an adjacency matrix A^ℓ and a node feature matrix X^ℓ of layer ℓ, each node in the graph is essentially a word in the text. We retain the order information of nodes from their original relative positions in texts. This indicates that the feature matrix X^ℓ is organized as traditional grid-like data with order information retained. From this point, we can apply an 1-D convolutional operation with kernel sizes larger than 1 on the feature matrix X^ℓ for high-level feature extraction.

The combination of the GCN operation and the convolutional operation in the hConv layer can take the advantages of both of them and overcome their respective limitations. In convolutional layers, the receptive fields of units on feature maps increase very slow since small kernel sizes are usually used to avoid massive number of parameters. In contrast, GCN operations can help to

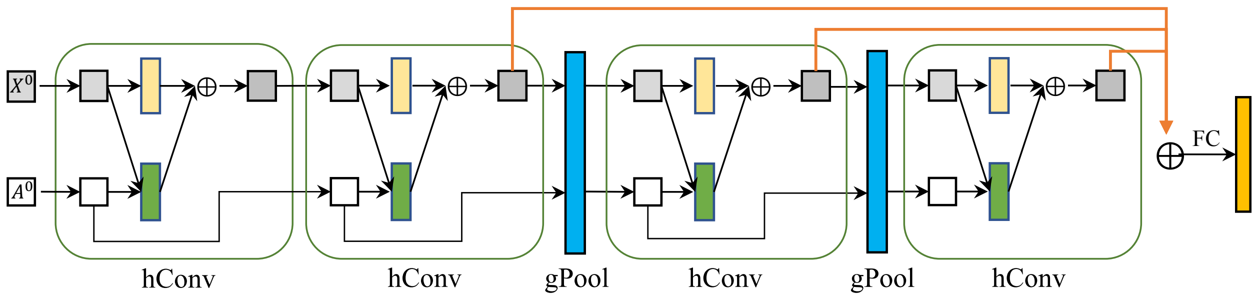

Figure 5.5: The architecture of the hConv-gPool-Net. $\oplus$ denotes the concatenation operation of feature maps. The inputs of the network are an adjacency matrix A^0 and a feature matrix X^0. We stack four hConv layers for feature extraction. In the second and the third hConv layers, we employ the gPool layers to reduce the number of nodes in graphs by half. Starting from the second hConv layer, a global max-pooling layer is applied to the output feature maps of each hConv layer. The outputs of these pooling layers are concatenated together. Finally, we employ a fully-connected layer for predictions. To obtain the other three networks discussed in Section 5.3.5, we can simply replace the hConv layers with GCN layers or remove gPool layers based on this network architecture.

increase the receptive fields quickly by means of edges between terms in sentences corresponding to nodes in graphs. At the same time, GCN operations are not able to automatically extract high-level features as they do not have trainable spatial filters as used in convolutional operations. From this point, the hConv layer is especially useful when working on text-based graph data such as sentences and documents.

5.3.4 Graph U-Nets Architecture for Node Classification

It is well-known that encoder-decoder networks like U-Net achieve promising performance on pixel-wise prediction tasks, since they can encode and decode high-level features while maintaining local spatial information. Similar to pixel-wise prediction tasks [106, 76], node classification tasks aim to make a prediction for each input unit. Based on our proposed gPool and gUnpool layers, we propose our graph U-Nets (g-U-Nets) architecture for node classification tasks.

In our graph U-Nets (g-U-Nets), we first apply a graph embedding layer to convert nodes into low-dimensional representations, since original inputs of some dataset like Cora [20] usually have very high-dimensional feature vectors. After the graph embedding layer, we build the encoder by stacking several encoding blocks, each of which contains a gPool layer followed by a GCN

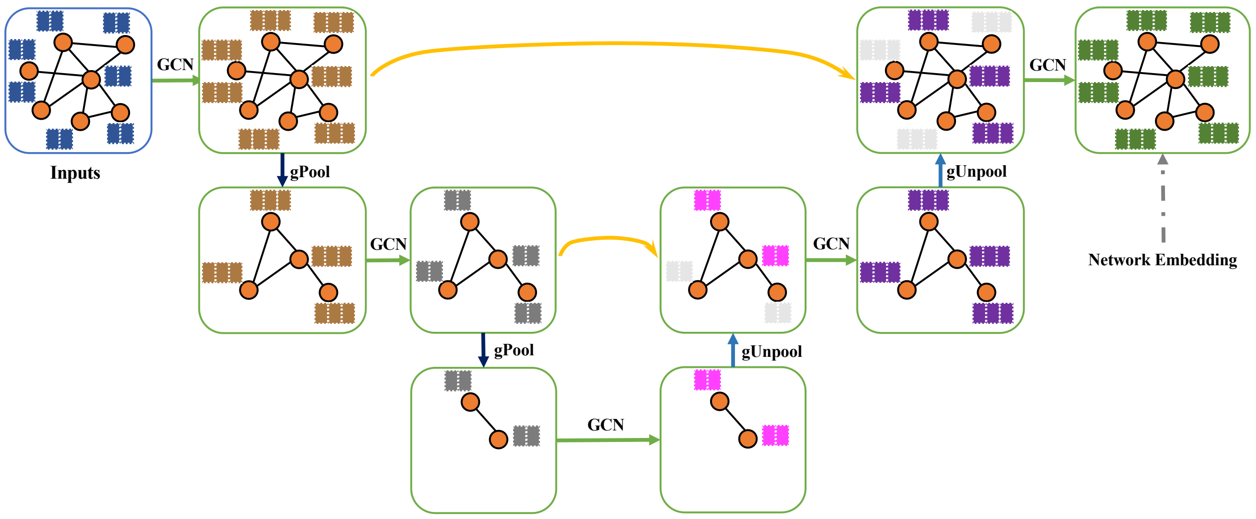

Figure 5.6: An illustration of the proposed graph U-Nets (g-U-Nets). In this example, each node in the input graph has two features. The input feature vectors are transformed into low-dimensional representations using a GCN layer. After that, we stack two encoder blocks, each of which contains a gPool layer and a GCN layer. In the decoder part, there are also two decoder blocks. Each block consists of a gUnpool layer and a GCN layer. For blocks in the same level, encoder block uses skip connection to fuse the low-level spatial features from the encoder block. The output feature vectors of nodes in the last layer are network embedding, which can be used for various tasks such as node classification and link prediction.

layer. gPool layers reduce the size of graph to encode higher-order features, while GCN layers are responsible for aggregating information from each node's first-order information. In the decoder part, we stack the same number of decoding blocks as in the encoder part. Each decoder block is composed of a gUnpool layer and a GCN layer. The gUnpool layer restores the graph into its higher resolution structure, and the GCN layer aggregates information from the neighborhood. There are skip-connections between corresponding blocks of encoder and decoder layers, which transmit spatial information to decoders for better performance. The skip-connection can be either feature map addition or concatenation. Finally, we employ a GCN layer for final predictions before the softmax function. Figure 5.6 provides an illustration of a sample g-U-Nets with two blocks in encoder and decoder. Notably, there is a GCN layer before each gPool layer, thereby enabling gPool layers to capture the topological information in graphs implicitly.

In GCN, the adjacency matrix before normalization is computed as $\hat{A} = A + I$ in which a self-loop is added to each node in the graph. When performing information aggregation, the same weight is given to node's own feature vector and its neighboring nodes. Here, we wish to give a higher weight to node's own feature vector, since its own feature should be more important for prediction. To this end, we change the calculation to $\hat{A} = A + 2I$ by imposing larger weights on self loops in the graph, which is common in graph processing. All experiments for this part use this modified version of GCN layer for better performance.

5.3.5 Graph Network Architectures for Text Classification

Based on our proposed gPool and hConv layers, we design four network architectures, including our baseline with a FCN-like [77] architecture. FCN has been shown to be very effective for image semantic segmentation. It allows final linear classifiers to make use of features from different layers. Here, we design four architectures based on our proposed gPool and hConv layers.

- **GCN-Net:** We establish a baseline method by using GCN layers to build a network without any hConv or gPool layers. In this network, we stack 4 standard GCN layers as feature extractors. Starting from the second layer, a global max-pooling layer [107] is applied to each layer's output. The outputs of these pooling layers are concatenated together and fed into a fully-connected layer for final predictions. This network serves as a baseline model for experimental studies.

- **GCN-gPool-Net:** In this network, we add our proposed gPool layers to GCN-Net. Starting from the second layer, we add a gPool layer after each GCN layer except for the last one. In each gPool layer, we select the hyper-parameter k to reduce the number of nodes in the graph by a factor of two. All other parts of the network remain the same as those of GCN-Net.

- **hConv-Net:** For this network, we replace all GCN layers in GCN-Net by our proposed hConv layers. To ensure the fairness of comparison among these networks, the hConv layers output the same number of feature maps as the corresponding GCN layers. Suppose the original ith GCN layer outputs n_{out} feature maps. In the corresponding hConv layer, both

Table 5.1: Summary of datasets used in our node classification experiments. The Cora, Citeseer, and Pubmed datasets [21, 22] are used for transductive learning experiments.

Dataset	Nodes	Features	Classes	Train	Valid	Test	Degree
Cora	2708	1433	7	140	500	1000	4
Citeseer	3327	3703	6	120	500	1000	5
Pubmed	19717	500	3	60	500	1000	6

the GCN operation and the convolutional operation output $n_{out}/2$ feature maps. By concatenating those intermediate outputs, the ith hConv layer also outputs n_{out} feature maps. The remaining parts of the network remain the same as those in GCN-Net.

- **hConv-gPool-Net:** Based on the hConv-Net network, we add gPool layers after each hConv layer except for the first and the last layers. We employ the same principle for the selection of hyper-parameter k as that in GCN-gPool-Net. The remaining parts of the network remain the same. Note that gPool layers maintain the order information of nodes in the new graph, thus enabling the application of 1-D convolutional operations in hConv layers afterwards. Figure 5.5 provides an illustration of the hConv-gPool-Net network.

5.4 Experimental Study on Graph Data

In this section, we evaluate our gPool and gUnpool layers based on the g-U-Nets proposed in Section 5.3.4. We compare our networks with previous state-of-the-art models on node classification and graph classification tasks. Experimental results show that our methods achieve new state-of-the-art results in terms of node classification accuracy and graph classification accuracy. Some ablation studies are performed to examine the contributions of the proposed gPool layer, gUnpool layer, and graph connectivity augmentation to performance improvements. We conduct studies on the relationship between network depth and node classification performance. We investigate if additional parameters involved in gPool layers can increase the risk of over-fitting. Our code is publicly available[2].

[2]`https://github.com/divelab/gunet/`

Table 5.2: Summary of datasets used in our inductive learning experiments. The D&D [45], PROTEINS [44], and COLLAB [46] datasets are used for inductive learning experiments.

Dataset	**Graphs**	**Nodes (max)**	**Nodes (avg)**	**Classes**
D&D	1178	5748	284.32	2
PROTEINS	1113	620	39.06	2
COLLAB	5000	492	74.49	3

5.4.1 Datasets

In experiments, we evaluate our networks on node classification tasks under transductive learning settings and graph classification tasks under inductive learning settings.

Under transductive learning settings, unlabeled data are accessible for training, which enables the network to learn about the graph structure. To be specific, only part of nodes are labeled while labels of other nodes in the same graph remain unknown. We employ three benchmark datasets for this setting; those are Cora, Citeseer, and Pubmed [14], which are summarized in Table 5.1. These datasets are citation networks, with each node and each edge representing a document and a citation, respectively. The feature vector of each node is the bag-of-word representation whose dimension is determined by the dictionary size. We follow the same experimental settings in [14]. For each class, there are 20 nodes for training, 500 nodes for validation, and 1000 nodes for testing.

Under inductive learning settings, testing data are not available during training, which means the training process does not use graph structures of testing data. We evaluate our methods on relatively large graph datasets selected from common benchmarks used in graph classification tasks [52, 18, 47]. We use protein datasets including D&D [45] and PROTEINS [44], the scientific collaboration dataset COLLAB [46]. These data are summarized in Table 5.2.

5.4.2 Experimental Setup

We describe the experimental setup for both transductive and inductive learning settings. For transductive learning tasks, we employ our proposed g-U-Nets proposed in Section 5.3.4. Since nodes in the three datasets are associated with high-dimensional features, we employ a GCN layer

Table 5.3: Results of transductive learning experiments in terms of node classification accuracies on Cora, Citeseer, and Pubmed datasets. g-U-Nets denotes our proposed graph U-Nets model.

Models	**Cora**	**Citeseer**	**Pubmed**
DeepWalk [26]	67.2%	43.2%	65.3%
Planetoid [21]	75.7%	64.7%	77.2%
Chebyshev [27]	81.2%	69.8%	74.4%
GCN [14]	81.5%	70.3%	79.0%
GAT [15]	$83.0 \pm 0.7\%$	$72.5 \pm 0.7\%$	$79.0 \pm 0.3\%$
g-U-Nets (Ours)	$\mathbf{84.4 \pm 0.6\%}$	$\mathbf{73.2 \pm 0.5\%}$	$\mathbf{79.6 \pm 0.2\%}$

to reduce them into low-dimensional representations. In the encoder part, we stack four blocks, each of which consists of a gPool layer and a GCN layer. We sample 2000, 1000, 500, 200 nodes in the four gPool layers, respectively. Correspondingly, the decoder part also contains four blocks. Each decoder block is composed of a gUnpool layer and a GCN layer. We use addition operation in skip connections between blocks of encoder and decoder parts. Finally, we apply a GCN layer for final prediction. For all layers in the model, we use identity activation function [49] after each GCN layer. To avoid over-fitting, we apply L_2 regularization on weights with $\lambda = 0.001$. Dropout [23] is applied to both adjacency matrices and feature matrices with keep rates of 0.8 and 0.08, respectively.

For inductive learning tasks, we follow the same experimental setups in [47] using our g-U-Nets architecture as described in transductive learning settings for feature extraction. Since the sizes of graphs vary in graph classification tasks, we sample proportions of nodes in four gPool layers; those are 90%, 70%, 60%, and 50%, respectively. The dropout keep rate imposed on feature matrices is 0.3.

5.4.3 Performance Study

Under transductive learning settings, we compare our proposed g-U-Nets with other state-of-the-art models in terms of node classification accuracy. We report node classification accuracies on datasets Cora, Citeseer, and Pubmed, and the results are summarized in Table 5.3. We can observe from the results that our g-U-Nets achieves consistently better performance than other

Table 5.4: Comparisons with other models in terms of graph classification accuracy (%) on social network datasets including COLLAB, IMDB-BINARY, IMDB-MULTI, REDDIT-BINARY, REDDIT-MULTI5K and REDDIT-MULTI12K datasets.

	COLLAB	**IMDB-B**	**IMDB-M**	**RDT-B**	**RDT-M5K**	**RDT-M12K**
# graphs	5000	1000	1500	2000	4999	11929
# nodes	74.5	19.8	13.0	429.6	508.5	391.4
# classes	3	2	3	2	5	11
WL	78.9 ± 1.9	73.8 ± 3.9	50.9 ± 3.8	81.0 ± 3.1	52.5 ± 2.1	44.4 ± 2.1
PSCN	72.6 ± 2.2	71.0 ± 2.2	45.2 ± 2.8	86.3 ± 1.6	49.1 ± 0.7	41.3 ± 0.8
DGCNN	73.8	70.0	47.8	-	-	41.8
DIFFPOOL	75.5	-	-	-	-	47.1
g-U-Net	77.5 ± 2.1	75.4 ± 3.0	51.8 ± 3.7	85.5 ± 1.3	48.2 ± 0.8	44.5 ± 0.6

networks. For baseline values listed for node classification tasks, they are the state-of-the-art on these datasets. Our proposed model is composed of GCN, gPool, and gUnpool layers without involving more advanced graph convolution layers like GAT. When compared to GCN directly, our g-U-Nets significantly improves performance on all three datasets by margins of 2.9%, 2.9%, and 0.6%, respectively. Note that the only difference between our g-U-Nets and GCN is the use of encoder-decoder architecture containing gPool and gUnpool layers. These results demonstrate the effectiveness of g-U-Nets in network embedding.

We compare our g-U-Nets with other state-of-the-art models in terms of graph classification accuracy. The comparison results are summarized in Table 5.4. We can observe from the results that our g-U-Net significantly outperform other models on most social network datasets by margins of 2.0%, 1.6%, 0.9% on COLLAB, IMDB-BINARY, and IMDB-MULTI datasets, respectively. The promising performances, especially on large datasets such as REDDIT, demonstrate the effectiveness of our methods.

5.4.4 Ablation Study of gPool and gUnpool layers

Although GCNs have been reported to have worse performance when the network goes deeper [14], it may also be argued that the performance improvement over GCN in Table 5.3 is due to the use of a deeper network architecture. In this section, we investigate the contributions of gPool

Table 5.5: Comparison of g-U-Nets with and without gPool or gUnpool layers in terms of node classification accuracy on Cora, Citeseer, and Pubmed datasets.

Models	**Cora**	**Citeseer**	**Pubmed**
g-U-Nets without gPool or gUnpool	$82.1 \pm 0.6\%$	$71.6 \pm 0.5\%$	$79.1 \pm 0.2\%$
g-U-Nets (Ours)	$\mathbf{84.4 \pm 0.6\%}$	$\mathbf{73.2 \pm 0.5\%}$	$\mathbf{79.6 \pm 0.2\%}$

Table 5.6: Comparison of g-U-Nets with and without graph connectivity augmentation in terms of node classification accuracy on Cora, Citeseer, and Pubmed datasets.

Models	**Cora**	**Citeseer**	**Pubmed**
g-U-Nets without augmentation	$83.7 \pm 0.7\%$	$72.5 \pm 0.6\%$	$79.0 \pm 0.3\%$
g-U-Nets (Ours)	$\mathbf{84.4 \pm 0.6\%}$	$\mathbf{73.2 \pm 0.5\%}$	$\mathbf{79.6 \pm 0.2\%}$

and gUnpool layers to the performance of g-U-Nets. We conduct experiments by removing all gPool and gUnpool layers from our g-U-Nets, leading to a network with only GCN layers with skip connections. Table 5.5 provides the comparison results between g-U-Nets with and without gPool or gUnpool layers. The results show that g-U-Nets have better performance over g-U-Nets without gPool or gUnpool layers by margins of 2.3%, 1.6% and 0.5% on Cora, Citeseer, and Pubmed datasets, respectively. These results demonstrate the contributions of gPool and gUnpool layers to performance improvement. When considering the difference between the two models in terms of architecture, g-U-Nets enable higher level feature encoding, thereby resulting in better generalization and performance.

5.4.5 Graph Connectivity Augmentation Study

In the above experiments, we employ gPool layers with graph connectivity augmentation by using the 2^{nd} graph power. Here, we conduct experiments on node classification tasks to investigate the benefits of graph connectivity augmentation based on g-U-Nets. We remove the graph connectivity augmentation from gPool layers while keeping other settings the same for fairness of comparisons. Table 5.6 provides comparison results between g-U-Nets with and without graph connectivity augmentation. The results show that the absence of graph connectivity augmenta-

Table 5.7: Comparison of different network depths in terms of node classification accuracy on Cora, Citeseer, and Pubmed datasets. Based on g-U-Nets, we experiment with different network depths in terms of the number of blocks in encoder and decoder parts.

Depth	**Cora**	**Citeseer**	**Pubmed**
2	$82.6 \pm 0.6\%$	$71.8 \pm 0.5\%$	$79.1 \pm 0.3\%$
3	$83.8 \pm 0.7\%$	$72.7 \pm 0.7\%$	$79.4 \pm 0.4\%$
4	$\mathbf{84.4 \pm 0.6\%}$	$\mathbf{73.2 \pm 0.5\%}$	$\mathbf{79.6 \pm 0.2\%}$
5	$84.1 \pm 0.5\%$	$72.8 \pm 0.6\%$	$79.5 \pm 0.3\%$

Table 5.8: Comparison of the g-U-Nets with and without gPool or gUnpool layers in terms of the node classification accuracy and the number of parameters on Cora dataset.

Models	**Accuracy**	**#Params**	**Ratio of increase**
g-U-Nets without gPool or gUnpool	$82.1 \pm 0.6\%$	75,643	0.00%
g-U-Nets (Ours)	$\mathbf{84.4 \pm 0.6\%}$	75,737	0.12%

tion will cause consistent performance degradation on all of three datasets. This demonstrates that graph connectivity augmentation via 2^{nd} graph power can help with the graph connectivity and information transfer among nodes in sampled graphs.

5.4.6 Network Depth Study of Graph U-Nets

Since the network depth in terms of the number of blocks in encoder and decoder parts is an important hyper-parameter in the g-U-Nets, we conduct experiments to investigate the relationship between network depth and performance in terms of node classification accuracy. We use different network depths on node classification tasks and report the classification accuracies. The results are summarized in Table 5.7. We can observe from the results that the performance improves as network goes deeper until a depth of 4. The over-fitting problem happens in deeper networks and prevents networks from improving when the depth goes beyond that. In image segmentation, U-Net models with depth 3 or 4 are commonly used [108, 109], which is consistent with our choice in experiments. This indicates the capacity of gPool and gUnpool layers in receptive field enlargement and high-level feature encoding even working with very shallow networks.

Table 5.9: Summary of datasets used in our experiments. The #words denotes the average number of words of the data samples for each dataset. These numbers help the selection of the sliding window size used in converting texts to graphs.

Datasets	**#Train**	**#Test**	**#Classes**	**#Words**
AG's News	120,000	7,600	4	45
DBPedia	560,000	70,000	14	55
Yelp Polarity	560,000	38,000	2	153
Yelp Full	650,000	50,000	5	155

Table 5.10: Results of text classification experiments in terms of classification error rate on the AG's News, DBPedia, Yelp Review Polarity, and Yelp Review Full datasets. The first two methods are the state-of-the-art models without using any unsupervised data. The last four networks are proposed in this work.

Models	**AG's News**	**DBPedia**	**Yelp Polarity**	**Yelp Full**
Word-level CNN [110]	8.55%	1.37%	4.60%	39.58%
Char-level CNN [110]	9.51%	1.55%	4.88%	37.95%
GCN-Net	8.64%	1.69%	7.74%	42.60%
GCN-gPool-Net	8.09%	1.44%	5.82%	41.83%
hConv-Net	7.49%	1.02%	4.45%	37.81%
hConv-gPool-Net	**7.09%**	**0.92%**	**4.37%**	**36.27%**

5.4.7 Parameter Study of Graph Pooling Layers

Since our proposed gPool layer involves extra parameters, we compute the number of additional parameters based on our g-U-Nets. The comparison results between g-U-Nets with and without gPool or gUnpool layers on dataset Cora are summarized in Table 5.8. From the results, we can observe that gPool layers in U-Net model only adds 0.12% additional parameters but can promote the performance by a margin of 2.3%. We believe this negligible increase of extra parameters will not increase the risk of over-fitting. Compared to g-U-Nets without gPool or gUnpool layers, the encoder-decoder architecture with our gPool and gUnpool layers yields significant performance improvement.

5.5 Experimental Study on Text Data

In this section, we evaluate our gPool layer and hConv layer based on the four networks proposed in Section 5.3.5. We compare the performances of our networks with that of previous state-of-the-art models. The experimental results show that our methods yield improved performance in terms of classification accuracy. We also perform some ablation studies to examine the contributions of the gPool layer and the hConv layer to the performance. The number of extra parameters in gPool layers is shown to be negligible and will not increase the risk of over-fitting. Our code is publicly available[3].

5.5.1 Datasets

In this work, we evaluate our methods on four datasets, including the AG's News, Dbpedia, Yelp Polarity, and Yelp Full [110] datasets. **AG's News** is a news dataset containing four topics: World, Sports, Business and Sci/Tech. The task is to classify each news into one of the topics. **Dbpedia** dataset contains 14 classes. It is constructed by choosing 14 non-overlapping classes from the DBPedia 2014 dataset [111]. Each sample contains a title and an abstract corresponding to a Wikipedia article. **Yelp Polarity** dataset is obtained from the Yelp Dataset Challenge in 2015 [110]. Each sample is a piece of review text with a binary label (negative or positive). **Yelp Full** dataset is obtained from the Yelp Dataset Challenge in 2015, which is for sentiment classification [110]. It contains five classes corresponding to the movie review star ranging from 1 to 5. The summary of these datasets are provided in Table 5.9. For all datasets, we tokenize the textual document and convert words to lower case. We remove stop-words and all punctuation in texts. Based on cleaned texts, we build the graph-of-word representations for texts.

5.5.2 Text to Graph Conversion

We use the graph-of-words method to convert texts into graph representations that include an adjacency matrix and a feature matrix. We select nouns, adjective, and verb as terms, meaning a word appears in the graph if it belongs to one of the above categories. We use a sliding window to

[3]`https://github.com/divelab/hConv-gPool-Net/`

Table 5.11: Comparison in terms of the network depth and the text classification error rate on the AG's News dataset. The depth listed here is calculated by counting the number of convolutional and fully-connected layers in networks.

Models	**Depth**	**Error Rate**
Word-level CNN	9	8.55%
Character-level CNN	9	9.51%
GCN-Net	5	8.64%
GCN-gPool-Net	5	8.09%
hConv-Net	5	7.49%
hConv-gPool-Net	5	**7.09%**

decide if two terms have an edge between them. If the distance between two terms is less than the window size, an undirected edge between these two terms is added. In the generated graph, nodes are the terms appear in texts, and edges are added using the sliding window. We use a window size of 4 for the AG's News and DBpedia datasets and 10 for the other two datasets, depending on their average words in training samples. The maximum numbers of nodes in graphs for the AG's News, DBPedia, Yelp Polarity, Yelp Full datasets are 100, 100, 300, and 256, respectively.

To produce the feature matrix, we use word embedding and position embedding features. For word embedding features, the pre-trained fastText word embedding vectors [112] are used, and it contains more than 2 million pre-trained words vectors. Compared to other pre-trained word embedding vectors such as GloVe [113], using the fastText helps us to avoid massive unknown words. On the AG's News dataset, the number of unknown words with the fastText is only several hundred, which is significantly smaller than the number using GloVe. In addition to word embedding features, we also employ position embedding method proposed in [114]. We encode the positions of words in texts into one-hot vectors and concatenate them with word embedding vectors. We obtain the feature matrix by stacking word vectors of nodes in the row dimension.

5.5.3 Experimental Setup

For our proposed networks, we employ the same settings with minor adjustments to accommodate the different datasets. As discussed in Section 5.3.5, we stack four GCN or hConv layers

Table 5.12: Comparison between the GCN-Net and GCN-gPool-Net in term of parameter numbers and text classification error rates on the AG's News dataset.

Models	Error rate	# Params	Ratio of increase
GCN-Net	8.64%	1,554,820	0.00%
GCN-gPool-Net	**8.09%**	1,555,719	0.06%

for GCN-based networks or hConv-based networks. For the networks using gPool layers, we add gPool layers after the second and the third GCN or hConv layers. Four GCN or hConv layers output 1024, 1024, 512, and 256 feature maps, respectively. We use this decreasing number of feature maps, since GCNs help to enlarge the receptive fields very quickly. We do not need more high-level features in deeper layers. The kernel sizes used by convolutional operations in hConv layers are all 3×1. For all layers, we use the ReLU [115] for nonlinearity. For all experiments, the following settings are shared. For training, the Adam optimizer [24] is used for 60 epochs. The learning rate starts at 0.001 and decays by 0.1 at the 30^th and the 50^th epoch. We employ the dropout with a keep rate of 0.55 [23] and batch size of 256. These hyper-parameters are tuned on the AG's News dataset, and then ported to other datasets.

5.5.4 Performance Study on Text Classification

We compare our proposed methods with other state-of-the-art models on text classification, and the experimental results are summarized in Table 5.10. We can see from the results that our hConv-gPool-Net outperforms both word-level CNN and character-level CNN by at least a margin of 1.46%, 0.45%, 0.23%, and 3.31% on the AG's News, DBPedia, Yelp Polarity, and Yelp Full datasets, respectively. The performance of GCN-Net with only GCN layers cannot compete with that of word-level CNN and char-level CNN primarily due to the lack of automatic high-level feature extraction. By replacing GCN layers using our proposed hConv layers, hConv-Net achieves better performance than the two CNN models across four datasets. This demonstrates the promising performance of our hConv layer by employing regular convolutional operations for automatic feature extraction. By comparing the GCN-Net with GCN-gPool-Net, and hConv-Net with hConv-gPool-Net, we observe that our proposed gPool layers promote both models' performance by at least a margin of 0.4%, 0.1%, 0.08%, and 1.54% on the AG's News, DBPedia, Yelp Polarity, and Yelp Full datasets. The margins tend to be larger on harder tasks. This observation demonstrates that our gPool layer helps to enlarge receptive fields and reduce spatial dimensions of graphs, resulting in better generalization and performance.

5.5.5 Network Depth Study

In addition to performance study, we also conduct experiments to evaluate the relationship between performance and network depth in terms of the number of convolutional and fully-connected layers in models. The results are summarized in Table 5.11. We can observe from the results that our models only require 5 layers, including 4 convolutional layers and 1 fully-connected layer. Both word-level CNN and character-level CNN models need 9 layers in their networks, which are much deeper than ours. Our hConv-gPool-Net achieves the new state-of-the-art performance with fewer layers, demonstrating the effectiveness of gPool and hConv layers. Since GCN and gPool layers enlarge receptive fields quickly, advanced features are learned in shallow layers, leading to shallow networks but better performance.

5.5.6 Parameter Study of gPool Layer

Since gPool layers involve extra trainable parameters in projection vectors, we study the number of parameters in gPool layers in the GCN-gPool-Net that contains two gPool layers. The results are summarized in Table 5.12. We can see from the results that gPool layers only needs 0.06% additional parameters compared to GCN-Net. We believe that this negligible increase of parameters will not increase the risk of over-fitting. With negligible additional parameters, gPool layers can yield a performance improvement of 0.54%.

6. TOPOLOGY-AWARE GRAPH STRUCTURE LEARNING

In the previous sections, we discussed how to learn new graph structures based on node features. In this section, we discuss graph deep learning methods that learn new graph structures by considering graph topology information.

6.1 Introduction

Pooling operations have been widely applied in various fields such as computer vision [6, 8, 19], and natural language processing [110]. Pooling operations can effectively reduce dimensional sizes [6, 116] and enlarge receptive fields [3]. The application of regular pooling operations depends on the well-defined spatial locality in grid-like data such as images and texts. However, it is still challenging to perform pooling operations on graph data. In particular, there is no spatial locality information or order information among the nodes in graphs [52, 49].

Some works try to overcome this limitation with two kinds of methods; those are node clustering [52] and primary nodes sampling [67, 47]. The node clustering methods create graphs with super-nodes by learning a nodes assignment matrix. The adjacency matrix of the learned graphs in node clustering methods are softly connected. These methods suffer from the over-fitting problem and need auxiliary link prediction tasks to stabilize the training [52]. The primary nodes sampling methods like top-k pooling [67, 47] rank the nodes in a graph and sample top-k nodes to form the sampled graph. It uses a small number of additional trainable parameters and is shown to be more powerful on various graph-related machine learning tasks [67]. However, the top-k pooling layer does not explicitly incorporate the topology information in a graph when computing ranking scores, which may cause performance loss. When generating ranking scores, only node features are used, which ignores the graph topology information. This can generate a coarsened graph with isolated nodes.

In this work, we propose a novel topology-aware pooling (TAP) layer that explicitly encodes the topology information when computing ranking scores. We use an attention operator to compute similarity scores between each node and its neighboring nodes. The average similarity score of a node is used as its ranking score in the selection process. To avoid isolated nodes problem in our TAP layer, we further propose a graph connectivity term for computing the ranking scores of nodes. The graph connectivity term uses degree information as a bias term to encourage the layer to select highly connected nodes to form the sampled graph. Based on the TAP layer, we develop topology-aware pooling networks for network embedding learning. Experimental results on graph classification tasks demonstrate that our proposed networks with TAP layer consistently outperform previous models. The comparison results between our TAP layer and other pooling layers based on the same network architecture demonstrate the effectiveness of our method compared to other pooling methods.

6.2 Background and Related Work

In this section, we describe graph pooling operations and attention operators.

6.2.1 Graph Pooling Operations

The pooling operations on graph data mainly include two categories; those are node clustering and node sampling. DIFFPOOL [52] realizes graph pooling operation by clustering nodes into super-nodes. By learning an assignment matrix, DIFFPOOL softly assigns each node to different clusters in the new graph with specified probabilities. The pooling operations under this category retain and encode all nodes information into the new graph. One challenge of methods in this category is that they may increase the risk of over-fitting by training another network to learn the assignment matrix. In addition, the new graph is mostly connected where each edge value represents the strength of connectivity between two nodes. The connectivity pattern in the new graph may greatly differ from that of the original graph.

The node sampling methods mainly select a fixed number k of the most important nodes to form a new graph. In SortPool [47], the same feature of each node is used for ranking and k nodes with the largest values in this feature are selected to form the coarsened graph. Top-k pooling [67] generates the ranking scores by using a trainable projection vector that projects feature vectors of

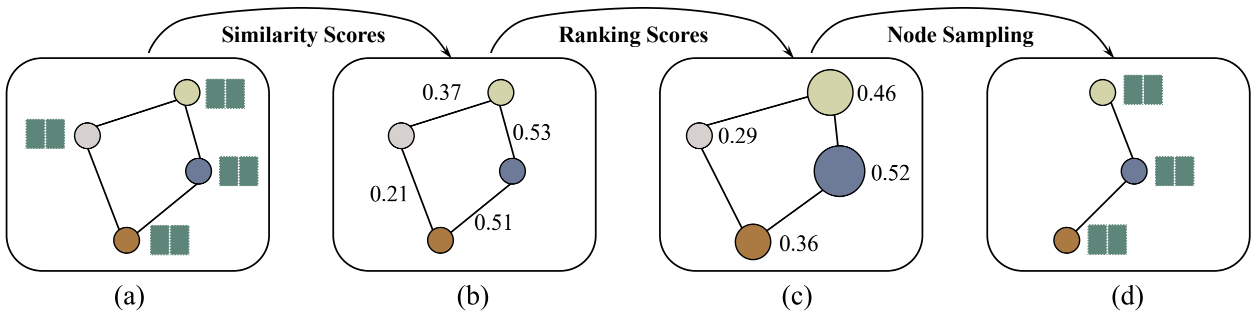

Figure 6.1: An illustration of the proposed topology-aware pooling layer that selects $k = 3$ nodes. This graph contains four nodes, each of which has 2 features. Given the input graph, we firstly use an attention operator to compute similarity scores between every pair of connected nodes. Here, we use self-attention without linear transformation for notation simplicity. In graph (b), we label each edge by the similarity score between its two ends. Then we compute the ranking score of each node by taking the average of the similarity scores between it and its neighboring nodes. In graph (c), we label each node by its ranking score and bigger node indicates a higher ranking score. By selecting the nodes with the $k = 3$ largest ranking scores, the coarsened graph is shown in graph (d).

nodes into scalar values. k nodes with the largest scalar values are selected to form the coarsened graph. These methods involve none or a very small number of extra trainable parameters, thereby avoiding the risk of over-fitting. However, these methods suffer from one limitation that they do not explicitly consider the topology information during pooling. Both SortPool and top-k pooling select nodes based on scalar values that do not explicitly incorporate topology information. In this work, we propose a pooling operation that explicitly encodes topology information in ranking scores, thereby leading to an improved operation.

6.2.2 Attention Operators

Attention operator has shown to be effective in challenging tasks in various fields such as computer vision [31, 117, 35] and natural language processing [40, 16, 17]. Attention operator is capable of capturing long-range relationships, thereby leading to better performances [30]. The inputs to an attention operator consist of three matrices; those are a query matrix $\boldsymbol{Q} \in \mathbb{R}^{d\times m}$, a key matrix $\boldsymbol{K} \in \mathbb{R}^{d\times n}$, and a value matrix $\boldsymbol{V} \in \mathbb{R}^{p\times n}$. The attention operator computes the response of each query vector in $\boldsymbol{Q}$ by attending it to all key vectors in $\boldsymbol{K}$. It uses the resulting coefficient vector to take a weighted sum over value vectors in $\boldsymbol{V}$. The layer-wise operation of an attention operation is defined as $\boldsymbol{O} = \boldsymbol{V}\text{softmax}(\boldsymbol{K}^T\boldsymbol{Q})$. When the attention operator is applied to graph, each node only attends to its neighboring nodes [15]. Self-attention can also produce an attention mask to control information flow on selected nodes in pooling operation [71]. In our proposed pooling operation, we employ an attention operator to compute ranking scores that explicitly encode topology information.

6.3 Topology-Aware Pooling Layers and Networks

In this work, we propose the topology-aware pooling (TAP) layer that uses attention operators to encode topology information in ranking scores for node selection. We also propose a graph connectivity term in the computation of ranking scores, which encourages better graph connectivity in the coarsened graph. Based on our TAP layer, we propose the topology-aware pooling networks for network representation learning.

6.3.1 Topology-Aware Pooling Layer

Pooling layers have shown to be important on grid-like data with regard to reducing feature map sizes and enlarging receptive fields [100, 118]. On graph data, two kinds of pooling layers have been proposed; those are node clustering [52] and primary nodes sampling [67, 47]. A primary nodes sampling method, known as top-k pooling [67], uses a projection vector to generate ranking scores for each node in the graph. The graph is created by choosing nodes with k-largest scores. However, the sampling process relies on the projection values that are generated by node features and a trainable projection vector. Top-k pooling does not explicitly consider the topology information in the graph, thereby leading to constrained network capability.

In this section, we propose the topology-aware pooling (TAP) layer that performs primary nodes sampling by considering the graph topology. In this layer, we generate the ranking scores based on local information. To this end, we employ an attention operator to compute the similarity scores between each node and its neighboring nodes. The ranking score for a node i is the mean value of the similarity scores with its neighboring nodes. The resulting ranking score for a node indicates the similarity between this node and its neighboring nodes. If a node has a high ranking score, it can highly represent a local graph that consists of it and its neighboring nodes. By choosing nodes with the highest ranking scores, we can retain the maximum information in the sampled graph.

Suppose there are N nodes in a graph $\mathbb{G}$, each of which contains C features. In layer ℓ, we use two matrices to represent the graph; those are the adjacency matrix $\boldsymbol{A}^{(\ell)} \in \mathbb{R}^{N\times N}$ and the feature matrix $\boldsymbol{X}^{(\ell)} \in \mathbb{R}^{N\times C}$. The non-zero entries in $\boldsymbol{A}^{(\ell)}$ represent edges in the graph. The ith row in $\boldsymbol{X}^{(\ell)}$ denotes the feature vector of node i. The layer-wise forward propagation rule of the TAP in the layer ℓ is defined as

$$\boldsymbol{K} = \boldsymbol{X}^{(\ell)}\boldsymbol{W}^{(\ell)}, \qquad \in\mathbb{R}^{N\times C} \tag{6.1}$$

$$\boldsymbol{E} = \boldsymbol{X}^{(\ell)}\boldsymbol{K}^{T}, \qquad \in\mathbb{R}^{N\times N} \tag{6.2}$$

$$\tilde{\boldsymbol{E}} = \boldsymbol{E} \circ \boldsymbol{A}^{(\ell)}, \qquad \in\mathbb{R}^{N\times N} \tag{6.3}$$

$$\boldsymbol{d} = \sum_{j=1}^{N} \boldsymbol{A}^{(\ell)}_{:j}, \qquad \in\mathbb{R}^{N} \tag{6.4}$$

$$\boldsymbol{s} = \text{sigmoid}\left(\frac{\sum_{j=1}^{N} \tilde{\boldsymbol{E}}_{:j}}{\boldsymbol{d}}\right), \qquad \in\mathbb{R}^{N} \tag{6.5}$$

$$\boldsymbol{idx} = \text{Ranking}_k(\boldsymbol{s}), \qquad \in\mathbb{R}^{k} \tag{6.6}$$

$$\boldsymbol{A}^{(\ell+1)} = \boldsymbol{A}^{(\ell)}(\boldsymbol{idx}, \boldsymbol{idx}), \qquad \in\mathbb{R}^{k\times k} \tag{6.7}$$

$$\boldsymbol{X}^{(\ell+1)} = \boldsymbol{X}^{(\ell)}(\boldsymbol{idx}, :)\text{diag}(\boldsymbol{s}(\boldsymbol{idx})), \qquad \in\mathbb{R}^{k\times C} \tag{6.8}$$

where $\boldsymbol{W}^{(\ell)} \in \mathbb{R}^{C\times C}$ is a trainable weight matrix, $\boldsymbol{A}^{(\ell)}_{:j}$ is the jth column of matrix $\boldsymbol{A}^{(\ell)}$, $\circ$ denotes the element-wise matrix multiplication, $\tilde{\boldsymbol{E}}_{:j}$ is the jth column of matrix $\tilde{\boldsymbol{E}}$, k is the number of nodes selected in the sampled graph, and diag$(\cdot)$ constructs a diagonal matrix using input vector as diagonal elements. Ranking$_k$ operator ranks the scores and return the indices of the k-largest values in $\boldsymbol{s}$, which represent the indices of selected nodes to form the coarsened graph. Based on the node indices, we extract a new adjacency matrix and a new feature matrix from the original graph.

To compute attention scores, we perform a linear transformation on feature matrix $\boldsymbol{X}^{(\ell)}$ in Eq. (6.1), which results in the key matrix $\boldsymbol{K}$. We use the input feature matrix as the query matrix. The similarity score matrix $\boldsymbol{E}$ is obtained by the matrix multiplication between $\boldsymbol{X}^{(\ell)}$ and $\boldsymbol{K}$ in Eq. (6.2). Each value e_{ij} in $\boldsymbol{E}$ measures the similarity between node i and node j. Since $\boldsymbol{E}$ contains similarity scores for nodes that are not directly connected, we use the adjacency matrix $\boldsymbol{A}^{(\ell)}$ as a mask to set these entries in $\boldsymbol{E}$ to zeros in Eq. (6.3), resulting in $\tilde{\boldsymbol{E}}$. We compute the degree of each node in Eq. (6.4). The ranking score of a node is computed in Eq. (6.5) by taking the average of similarity scores between this node and its neighboring nodes followed by a sigmoid operation. The sigmoid operation also serves as a gate conversion function that converts the values in the range between 0 and 1. Here, we perform element-wise division between two vectors. The resulting score vector is $\boldsymbol{s} = [s_1, s_2, \ldots, s_N]^T$ where s_i represents the ranking score of node i. Ranking_k is an operator that selects the k-largest values and returns the corresponding node indices. In Eq. (6.6), we use Ranking_k to select the k-most important nodes with indices in $\boldsymbol{idx}$. Using indices $\boldsymbol{idx}$, we extract a new adjacency matrix $\boldsymbol{A}^{(\ell+1)}$ in Eq. (6.7) and a new feature matrix $\boldsymbol{X}^{(\ell+1)}$ in Eq. (6.8) from the original graph. Here, we use the ranking scores $\boldsymbol{s}(\boldsymbol{idx})$ as gates to control information flow and enable the gradient back-propagation for the trainable parameters in the transformation matrix $\boldsymbol{W}^{(\ell)}$ [67].

This method can be considered as a local-voting, global-ranking process. In our TAP layer, the ranking scores are derived from the similarity scores of each node with its neighboring nodes, thereby encoding the topology information of each node in its ranking score. This can be considered as a local voting process that each node gets its votes from the local neighborhood. If a node is important in the graph, it will receive higher similarity scores on average. When performing global ranking, the nodes that get the highest votes from local neighborhoods are selected such that maximum information in the graph can be retained. Figure 6.1 provides an illustration of our proposed TAP layer. Compared to the top-k pooling [67], our TAP layer considers topology information in the graph, thereby leading to a better coarsened graph.

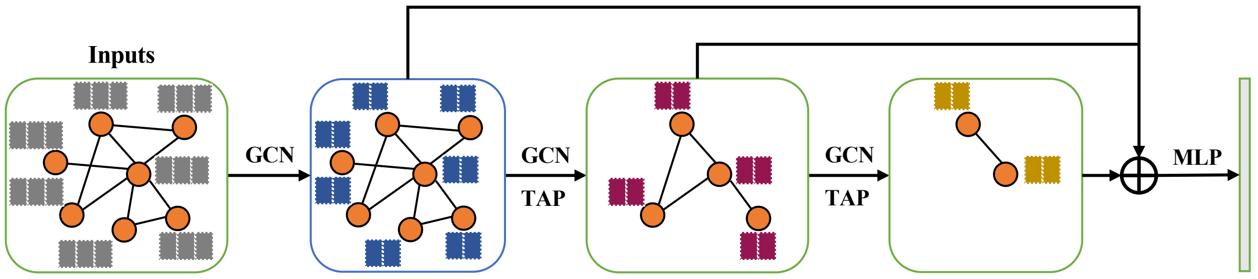

Figure 6.2: An illustration of the topology-aware pooling network. $\oplus$ denotes the concatenation operation of feature vectors. Each node in the input graph contains three features. We use a GCN layer to transform the feature vectors into low-dimensional representations. We stack two blocks, each of which consists of a GCN layer and a TAP layer. A global reduction operation such as max-pooling is applied to the outputs of the first GCN layer and TAP layers. The resulting feature vectors are concatenated and fed into the final multi-layer perceptron for prediction.

6.3.2 Graph Connectivity Term

Our proposed TAP layer computes the ranking scores by using similarity scores between nodes in the graph, thereby regarding topology information in the graph. However, the coarsened graph generated by the TAP layer may suffer from the problem of isolated nodes. In sparsely connected graphs, some nodes have a very small number of neighboring nodes or even only themselves. Suppose node i only connects to itself. The ranking score of node i is the similarity score to itself, which may result in high ranking scores in the graph. The resulting graph can be very sparsely connected, which completely lose the original graph structure. In the extreme situation, the coarsened graph can contain only isolated nodes without any connectivity. This can inevitably hurt the model performance.

To overcome the limitation of TAP layer and encourage better connectivity in the selected graph, we propose to add a graph connectivity term to the computation of ranking scores. To this end, we use node degrees as an indicator for graph connectivity and add degree values to their ranking scores such that densely-connected nodes are preferred during nodes selection. By using the node degree as the graph connectivity term, the ranking score of node i is computed as

$$s_i = \text{sigmoid}\left(\frac{\sum_{j=1}^{N} \tilde{E}_{ij}}{d_i}\right) + \lambda \frac{d_i}{N}, \tag{6.9}$$

where d_i is the degree of node i, and λ is a hyperparameter that sets the importance of the graph connectivity term to the computation of ranking scores. The graph connectivity term can overcome the limitation of the TAP layer. The computation of ranking scores now considers nodes degrees and gives rise to better connectivity in the resulting graph. A better connected coarsened graph is expected to retain more graph structure information, thereby leading to better model performances.

6.3.3 Topology-Aware Pooling Networks

Based on our proposed TAP layer, we build a family of networks known as topology-aware pooling networks (TAPNets) for graph classification tasks. In TAPNets, we firstly apply a graph embedding layer to produce low-dimensional representations of nodes in the graph, which helps to deal with some datasets with very high-dimensional input feature vectors. There are multiple choices for this graph embedding layer such as fully-connected layer and GCN layer. Here, we use a GCN layer [14] for node embedding for the sake of the performance. After the embedding layer, we stack several blocks, each of which consists of a GCN layer for high-level feature extraction and a TAP layer for graph coarsening. The output of each TAP layer is fed into the next GCN layer.

In the ith TAP layer, we use a hyperparameter $k^{(i)}$ to control the number nodes in the sampled graph. We feed the output feature matrices of the graph embedding layer and TAP layers to a classifier. Suppose we stack m blocks and all GCN and TAP layers output h feature maps. Given an input graph with the adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{N \times N}$ and the feature matrix $\boldsymbol{X} \in \mathbb{R}^{N \times C}$, our TAPNet outputs a list of feature matrices $[\boldsymbol{Y}_0, \boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_m]$. Here, $\boldsymbol{Y}_0 \in \mathbb{R}^{N \times h}$ is the output of the graph embedding layer and $\boldsymbol{Y}_i \in \mathbb{R}^{k^{(i)} \times h}$ is the output of TAP layer in the ith block. Here, we gather outputs from all blocks.

In TAPNets, we use a multi-layer perceptron as the classifier. We first transform network outputs to a one-dimensional vector. Specificity, the resulting vector $\boldsymbol{z} = [\boldsymbol{y}_0^T, \boldsymbol{y}_1^T, \ldots, \boldsymbol{y}_m^T]^T$ where $\boldsymbol{y}_i$ is transformed from $\boldsymbol{Y}_{\boldsymbol{i}}$. Global max and average pooling operations are two popular ways for the transformation, which reduce the spatial size of feature matrices to 1 using max and average functions, respectively. Recently, [69] proposed to use the summation function that results in

Table 6.1: Comparisons between TAPNets and other models in terms of graph classification accuracy (%) on social network datasets including COLLAB, IMDB-BINARY, IMDB-MULTI, REDDIT-BINARY, REDDIT-MULTI5K and REDDIT-MULTI12K datasets.

	COLLAB	**IMDB-B**	**IMDB-M**	**RDT-B**	**RDT-M5K**	**RDT-M12K**
# graphs	5000	1000	1500	2000	4999	11929
# nodes	74.5	19.8	13.0	429.6	508.5	391.4
# classes	3	2	3	2	5	11
WL	78.9 ± 1.9	73.8 ± 3.9	50.9 ± 3.8	81.0 ± 3.1	52.5 ± 2.1	44.4 ± 2.1
PSCN	72.6 ± 2.2	71.0 ± 2.2	45.2 ± 2.8	86.3 ± 1.6	49.1 ± 0.7	41.3 ± 0.8
DGCNN	73.8	70.0	47.8	-	-	41.8
DIFFPOOL	75.5	-	-	-	-	47.1
g-U-Net	77.5 ± 2.1	75.4 ± 3.0	51.8 ± 3.7	85.5 ± 1.3	48.2 ± 0.8	44.5 ± 0.6
GIN	80.6 ± 1.9	75.1 ± 5.1	52.3 ± 2.8	92.4 ± 2.5	**57.5 ± 1.5**	-
TAPNet (ours)	**84.6 ± 1.7**	**79.5 ± 4.1**	**55.6 ± 2.9**	**94.1 ± 1.9**	57.1 ± 1.3	**49.2 ± 1.6**

promising performances. In TAPNets, we concatenate transformation output vectors produced by the global pooling operations using max, averaging, and summation, respectively. The resulting feature vector is fed into the classifier. Figure 6.2 illustrates a sample TAPNet with two blocks.

Note that our TAP layers can also be applied to node classification tasks by replacing top-k pooling layers in graph U-Nets [67].

6.3.4 Auxiliary Link Prediction Objective

Multi-task learning has shown to be effective across various machine learning tasks [119, 52]. It can leverage useful information in multiple related tasks, thereby leading to better generalization and performance. In this section, we propose to add an auxiliary link prediction objective during training by using a by-product of our TAP layer. In Eq. (6.2), we compute the similarity scores $\boldsymbol{E}$ between every pair of nodes in the graph. By applying an element-wise sigmoid$(\cdot)$ on $\boldsymbol{E}$, we can obtain a link probability matrix $\hat{\boldsymbol{E}}^{(\ell)} \in \mathbb{R}^{N\times N}$ with each element $\hat{e}_{ij}$ measures the likelihood of a link between node i and node j in the graph.

With the adjacency matrix $\boldsymbol{A}^{(\ell)}$, we compute the auxiliary link prediction loss as

$$loss_{aux} = \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N} f\left(\hat{\boldsymbol{E}}_{ij}^{(\ell)}, \boldsymbol{A}_{ij}^{(\ell)}\right), \tag{6.10}$$

where $f(\cdot, \cdot)$ is a loss function that computes the distance between the link probability matrix $\hat{\boldsymbol{E}}^{(\ell)}$ and the adjacency matrix $\boldsymbol{A}^{(\ell)}$.

Note that the adjacency matrix used as the link prediction objective is directly derived from the original graph. Since the TAP layer extracts a sub graph from the original one, the connectivity between two nodes in the sampled graph is the same as that in the original graph. This means the adjacency matrices in deeper network are still using the original graph structure. Compared to auxiliary link prediction in DiffPool [52] that uses the learned adjacency matrix as objective, our method uses the original links, thereby providing more accurate information. This can also be clearly observed in the experimental studies in Sections 6.4.2 and 6.4.3.

6.4 Experimental Studies

In this section, we evaluate our methods and networks on graph classification tasks using bioinformatics and social network datasets. We conduct ablation experiments to evaluate the contributions of the TAP layer and each term in it to the overall network performance. Some experiments are performed to investigate how to choose the hyperparameter λ in the TAP layer.

6.4.1 Experimental Setup for Graph Classification Tasks

We evaluate our methods using social network datasets and bioinformatics datasets. They share the same experimental setups except for minor differences. The node features in social network networks are created using one-hot encodings of node degrees. The nodes in bioinformatics have categorical features. We use the TAPNet proposed in Section 6.3.3 that consists of one GCN layer and three blocks. The first GCN layer is used to learn low-dimensional representations of nodes in the graph. Each block is composed of one GCN layer and one TAP layer. All GCN and TAP layers output 48 feature maps. We use Leaky ReLU [120] with a slop of 0.01 to activate the outputs of GCN layers. The three TAP layers in the networks select numbers of nodes that are proportional to the nodes in the graph. We use the rates of 0.8, 0.6, and 0.4 in three TAP layers, respectively. We use $\lambda = 0.1$ to control the importance of the graph connectivity term in the computation of ranking scores.

Table 6.2: Comparisons between TAPNets and other models in terms of graph classification accuracy (%) on bioinformatics datasets including DD, PTC, MUTAG, and PROTEINS datasets.

	DD	**PTC**	**MUTAG**	**PROTEINS**
# graphs	1178	344	188	1113
# nodes	284.3	25.5	17.9	39.1
# classes	2	2	2	2
WL	78.3 ± 0.6	59.9 ± 4.3	90.4 ± 5.7	75.0 ± 3.1
PSCN	76.3 ± 2.6	60.0 ± 4.8	92.6 ± 4.2	75.9 ± 2.8
DGCNN	79.4 ± 0.9	58.6 ± 2.4	85.8 ± 1.7	75.5 ± 0.9
SAGPool	76.5	-	-	71.9
DIFFPOOL	80.6	-	-	76.3
g-U-Net	82.4 ± 2.9	64.7 ± 6.8	87.2 ± 7.8	77.6 ± 2.6
GIN	82.0 ± 2.7	64.6 ± 7.0	90.0 ± 8.8	76.2 ± 2.8
TAPNet (ours)	**84.2** ± **3.7**	**72.7** ± **6.0**	**93.0** ± **5.8**	**78.9** ± **4.2**

Dropout [23] is applied to the input feature matrices of GCN and TAP layers with keep rate of 0.7. We use a two-layer feed-forward network as the network classifier. Dropout with keep rate of 0.8 is applied to input features of two layers. We use ReLU activation function on the output of the first layer on DD, PTC, MUTAG, COLLAB, REDDIT-MULTI5K, and REDDIT-MULTI12K datasets. We use ELU [121] for other datasets. We train our networks using Adam optimizer [24] with a learning rate of 0.001. To avoid over-fitting, we use L_2 regularization with $\lambda = 0.0008$. All models are trained using one NVIDIA GeForce RTX 2080 Ti GPU. We will release our code in the final version.

6.4.2 Graph Classification Results on Social Network Datasets

We conduct experiments on graph classification tasks to evaluate our proposed methods and TAPNets. We use 6 social network datasets; those are COLLAB, IMDB-BINARY (IMDB-B), IMDB-MULTI (IMDB-M), REDDIT-BINARY (RDT-B), REDDIT-MULTI5K (RDT-M5K) and REDDIT-MULTI12K (RDT-M12K) [46] datasets. Note that REDDIT datasets are popular large datasets used for network embedding learning in terms of graph size and number of graphs [52, 69]. Since there is no feature for nodes in social networks, we create node features by following the practices in [69]. In particular, we use one-hot encodings of node degrees as feature vectors for

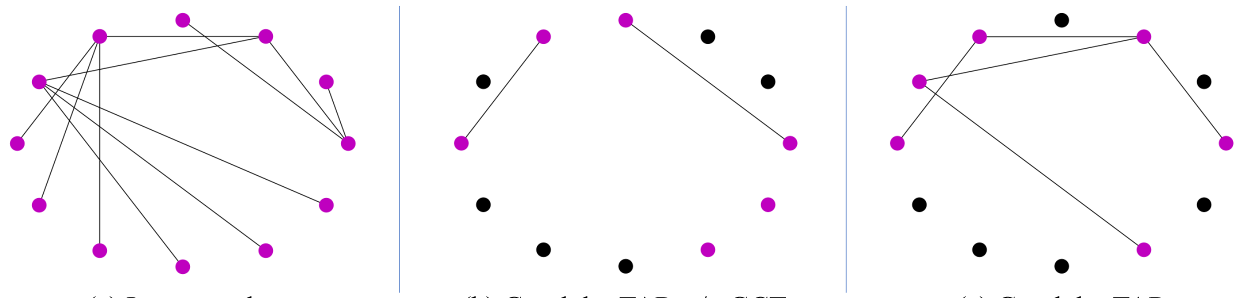
(a) Input graph (b) Graph by TAP w/o GCT (c) Graph by TAP

Figure 6.3: Illustrations of coarsened graphs generated by TAP and TAP w/o GCT. Here, GCT denotes the graph connection term. The input graph (a) contains 12 nodes. The pooling layers select 6 nodes to form new graphs. The nodes that are not selected are colored black. The new graph in (b) generated by TAP w/o GCT is sparsely connected. The new graph generated by TAP is illustrated in (c), which is shown to be much better connected.

nodes in social network datasets. On these datasets, we perform 10-fold cross-validation as in [47] with 9 folds for training and 1 fold for testing. To ensure fair comparisons, we do not use the auxiliary link prediction objective in these experiments.

We compare our TAPNets with other state-of-the-art models in terms of graph classification accuracy. The comparison results are summarized in Table 6.1. We can observe from the results that our TAPNets significantly outperform other models on most social network datasets by margins of 4.0%, 4.4%, 3.3%, 1.7%, and 2.1% on COLLAB, IMDB-BINARY, IMDB-MULTI, REDDIT-BINARY, and REDDIT-MULTI12K datasets, respectively. The promising performances, especially on large datasets such as REDDIT, demonstrate the effectiveness of our methods. Note that the superior performances over g-U-Net [67] show that our TAP layer can produce better-coarsened graph than that using the top-k pooling layer.

6.4.3 Graph Classification Results on Bioinformatics Datasets

We have shown the promising performances of our TAPNets on social network datasets. To fully evaluate our methods, we conduct experiments on graph classification tasks using 4 bioinformatics datasets; those are DD [45] , PTC [73] , MUTAG [72] , and PROTEINS [44] [69] datasets. Different from nodes in social network datasets, nodes in bioinformatics datasets have categorical features. In these experiments, we do not use the auxiliary link prediction objective. We

Table 6.3: Comparisons between different pooling operations based on the same TAPNet architecture in terms of the graph classification accuracy (%) on PTC, IMDB-MULTI, and REDDIT-BINARY datasets.

Model	**PTC**	**IMDB-M**	**RDT-B**
Net$_{\text{diff}}$	70.9	54.9	92.1
Net$_{\text{sort}}$	70.6	54.8	92.3
Net$_{\text{top-}k}$	71.5	55.2	92.8
TAPNet	**72.7**	**55.6**	**94.1**

compare our TAPNets with other state-of-the-art models in terms of graph classification accuracy without using the auxiliary link prediction term in loss function.

The comparison results are summarized in Table 6.2. We can observe from the results that our TAPNets achieve significantly better results than other models by margins of 1.2%, 8.1%, 3.0%, and 2.7% on DD, PTC, MUTAG, and PROTEINS datasets, respectively. Notably, some bioinformatics datasets such as PTC and MUTAG are much smaller than social network datasets in terms of number of graphs and number of nodes in graphs. The promising results on these small datasets demonstrate that our methods can achieve good generalization and performances without involving the risk of over-fitting.

SAGPool [71] uses a GCN layer to compute ranking scores and a self-attention operator to generate a mask to control the information flow. Here, our method employs an attention operator to compute ranking scores that better encodes the topology information in the graph. The superior performances over SAGPool on DD and PROTEINS datasets demonstrate that our methods can better capture the topology information, thereby leading to better performances.

6.4.4 Comparison with Other Graph Pooling Layers

It may be argued that our TAPNets achieve promising results by employing superior networks. In this section, we conduct experiments on the same TAPNet architecture to compare our TAP layer with other graph pooling layers; those are DIFFPOOL, sort pooling, and top-k pooling layers. We denote the networks with the TAPNet architecture while using these pooling layers as Net$_{\text{diff}}$, Net$_{\text{sort}}$, and Net$_{\text{top-}k}$, respectively. We evaluate them on PTC, IMDB-MULTI, and REDDIT-

Table 6.4: Comparisons among TAPNets with and without TAP layers, TAPNet without attention score term (AST), TAPNet without graph connection term (GCT), and TAPNet with auxiliary link prediction objective (AUX) in terms of the graph classification accuracy (%) on PTC, IMDB-MULTI, and REDDIT-BINARY datasets.

Model	**PTC**	**IMDB-M**	**RDT-B**
TAPNet w/o TAP	70.6	52.1	91.0
TAPNet w/o AST	71.2	54.8	91.5
TAPNet w/o GCT	72.0	55.1	93.0
TAPNet	72.7	55.6	94.1
TAPNet w AUX	**73.0**	**55.8**	**94.2**

BINARY datasets and summarize the results in Table 6.3. Note that these models use the same experimental setups to ensure fair comparisons. The results demonstrate the superior performance of our proposed TAP layer compared with other pooling layers using the same network architecture.

6.4.5 Ablation Studies

In this section, we investigate the contributions of TAP layer and its components in ranking score computation; those are the attention score term (AST) and the graph connectivity term (GCT). We remove TAP layers from TAPNet which we denote as TAPNet w/o TAP. To explore the contributions of terms in ranking scores computation, we separately remove ASTs and GCTs from all TAP layers in TAPNets. We denote the resulting models as TAPNet w/o AST and TAPNet w/o GCT, respectively. In addition, we add the auxiliary link prediction objective as described in Section 6.3.4 in training. We denote the TAPNet using auxiliary training objective as TAPNet w AUX. We evaluate these models on three datasets; those are PTC, IMDB-MULTI, and REDDIT-BINARY datasets.

The comparison results on these datasets are summarized in Table 6.4. The results show that TAPNets outperform TAPNets w/o TAP by margins of 2.1%, 3.5%, and 2.4% on PTC, IMDB-MULTI, and REDDIT-BINARY datasets, respectively. The better results of TAPNet over TAPNet w/o AST and TAPNet w/o GCT show the contributions of ASTs and GCTs to performances.

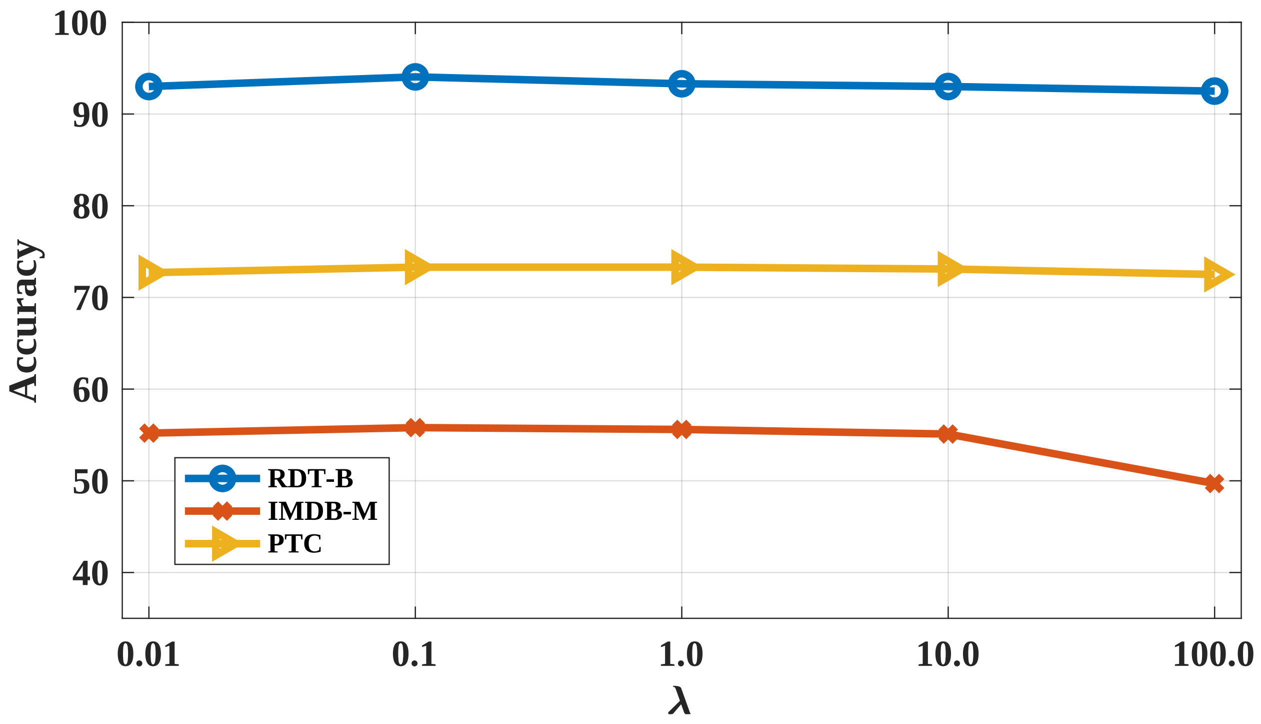

Figure 6.4: Comparison results of TAPNets using different λ values in TAP layers. We report graph classification accuracy (%) on PTC, IMDB-MULTI, and REDDIT-BINARY datasets.

It can be observed that TAPNet w AUX achieves better performances than TAPNet, which shows the effectiveness of the auxiliary link prediction objective. To fully study the impact of GCT on TAP layer, we visualize the coarsened graphs generated by TAP and TAP without GCT (denoted as TAP w/o GCT). We select a graph from PTC dataset and illustrate outcome graphs in Figure 6.3. We can observe from the figure that TAP produces a better-connected graph than that by TAP w/o GCT.

6.4.6 Parameter Study of GAP

Since TAP layer employs an attention operator to compute ranking scores, it involves extra trainable parameters to the overall network. Here, we conduct experiments to study the number of parameters in TAPNet. We remove the extra trainable parameters from TAP layers in two ways; those are removing TAP layers from the TAPNet and removing attention score terms (AST) from TAP layers. We denote the resulting two networks as TAPNet w/o TAP and TAPNet w/o AST,

Table 6.5: Comparisons among TAPNets with and without TAP layers, and TAPNet without attention score term (AST) in terms of the graph classification accuracy (%), and the number of parameters on REDDIT-BINARY dataset.

Model	**Accuracy**	**#Params**	**Ratio**
TAPNet w/o TAP	91.0	323,666	0.00%
TAPNet w/o AST	91.5	323,666	0.00%
TAPNet	**94.1**	330,578	2.13%

respectively.

The comparison results on REDDIT-BINARY dataset is summarized in Table 6.5. We can see from the results that TAP layers only need 2.13% additional trainable parameters. We believe the negligible usage of extra parameters will not increase the risk of over-fitting but can bring 1.9% performance improvement over TAPNet w/o TAP and TAPNet w/o AST on REDDIT-BINARY dataset. Also, the promising performances of TAPNets on small datasets like PTC and MUTAG in Table 6.2 show that TAP layers will not significantly increase the number of trainable parameters or cause the over-fitting problem.

6.4.7 Performance Study of λ

In Section 6.3.2, we propose to add the graph connectivity term into the computation of ranking scores to improve the graph connectivity in the coarsened graph. It can be seen that λ is an influential hyperparameter in the TAP layer. In this part, we study the impacts of different λ values on network performances. We select different λ values from the range of 0.01, 0.1, 1.0, 10.0, and 100.0 to cover a reasonable range of values. We evaluate TAPNets using different λ values on PTC, IMDB-MULTI, and REDDIT-BINARY datasets.

The results are shown in Figure 6.4. We can observe that the best performances on three datasets are achieved with $\lambda = 0.1$. When λ becomes larger, the performances of TAPNet models decrease. This indicates that the graph connectivity term is a plus term for generating reasonable ranking scores but it should not overwhelm the attention score term that encodes the topology information in ranking scores.

7. CONCLUSIONS AND FUTURE WORK

In Section 2, we discuss how to learning high-level node features without changing graph structures. In particular, we propose the learnable graph convolutional layer (LGCL), which transforms generic graphs to data of grid-like structures and enables the use of regular convolutional operations. The transformation is conducted through a novel k-largest node selection process, which uses the ranking between node feature values. Based on our LGCL, we build deeper networks, known as learnable graph convolutional networks (LGCNs), for node classification tasks on graphs. Experimental results show that the proposed LGCN models yield consistently better performance than prior methods under both transductive and inductive learning settings. Our LGCN models achieve new state-of-the-art results on four different datasets, demonstrating the effectiveness of LGCLs. In addition, we propose a sub-graph selection algorithm, resulting in the sub-graph training strategy, which can solve the problem of excessive requirements for memory and computational resources on large-scale graph data. With the sub-graph training, the proposed LGCN models are both effective and efficient. Our experiments indicate that the sub-graph training strategy brings a significant advantage in terms of training speed, with a negligible amount of performance loss. The new training strategy is very useful as it enables the use of more complex models efficiently.

In Section 3, we propose novel hGAO and cGAO which are attention operators on graph data. hGAO achieves the hard attention operation by selecting important nodes for the query node to attend. By employing a trainable projection vector, hGAO selects k-most important nodes for each query node based on their projection scores. Compared to GAO, hGAO saves computational resources and attends important adjacent nodes, leading to better generalization and performance. Furthermore, we propose the cGAO, which performs attention operators from the perspective of channels. cGAO removes the dependency on the adjacency matrix and dramatically saves computational resources compared to GAO and hGAO. Based on our proposed attention operators, we propose a new architecture that employs a densely connected design pattern to promote feature reuse. We evaluate our methods under both transductive and inductive learning settings. Experimental results demonstrate that our hGANets achieve improved performance compared to prior state-of-the-art networks. The comparison between our methods and GAO indicates that our hGAO achieves significant better performance than GAO. Our cGAO greatly saves computational resources and makes attention operators applicable on large graphs.

In Section 4, we consider the biased topology information encoding in graph neural networks that utilize line graph structures to enhance network embeddings. A line graph constructed from a graph can faithfully encode the topology information. However, the dynamics in the line graph are inconsistent with that in the original graph. On line graphs, the features of nodes with high degrees are more frequently passed in the graph, which causes understatement or overstatement of node features. To address this issue, we propose the weighted line graph that assigns normalized weights on edges such that the weighted degree of each node is 2. Based on the weighted line graph, we propose the weighted line graph layer that leverages the advantage of the weighted line graph structure. A practical challenge faced by graph neural networks on line graphs is that they consume excessive computational resources, especially on dense graphs. To address this limitation, we propose to use the incidence matrix to implement the WLGCL, which can dramatically save the computational resources. Based on the WLGCL, we build a family of weighted line graph convolutional networks. The experimental results on graph classification datasets and simulated data demonstrate the effectiveness and efficiency of our proposed methods and networks.

In Section 5, we introduce how to learn graph deep learning methods that can operate graph structures. In this section, we firstly propose novel gPool and gUnpool layers in g-U-Nets networks for network embedding. The gPool layer implements the regular global k-max pooling operation on graph data. It samples a subset of important nodes to enable high-level feature encoding and receptive field enlargement. By employing a trainable projection vector, gPool layers sample nodes based on their scalar projection values. Furthermore, we propose the gUnpool layer which applies unpooling operations on graph data. By using the position information of nodes in the original graph, gUnpool layer performs the inverse operation of the corresponding gPool layer and restores the original graph structure. Based on our gPool and gUnpool layers, we propose the graph U-Nets (g-U-Nets) architecture which uses a similar encoder-decoder architecture as regular U-Net on image data. Experimental results demonstrate that our g-U-Nets achieve performance improvements as compared to other GNNs on transductive learning tasks. To avoid the isolated node problem that may exist in sampled graphs, we employ the 2^{nd} graph power to improve graph connectivity. Ablation studies indicate the contributions of our graph connectivity augmentation approach.

Besides graph data, we use gPool and propose hConv layers in FCN-like graph convolutional networks for text modeling. The gPool layer achieves the effect of regular pooling operations on graph data to extract important nodes in graphs. By learning a projection vector, all nodes are measured through cosine similarity with the projection vector. The nodes with the k-largest scores are extracted to form a new graph. The scores are then applied to the feature matrix for information control, leading to the additional benefit of making the projection vector trainable. Since graphs are extracted from texts, we maintain the node orders as in the original texts. We propose the hConv layer that combines GCN and regular convolutional operations to enable automatic high-level feature extraction. Based on our gPool and hConv layers, we propose four networks for the task of text categorization. Our results show that the model based on gPool and hConv layers achieves new state-of-the-art performance compared to CNN-based models. gPool layers involve negligible number of parameters but bring significant performance boosts, demonstrating its contributions to model performance.

In Section 6, we propose a novel topology-aware pooling (TAP) layer that applies attention mechanism to explicitly encode the topology information in ranking scores. A TAP layer attends each node to its neighboring nodes and uses the average similarity score with its neighboring nodes as its ranking score. The primary nodes sampling based on these ranking scores can incorporate the topology information, thereby leading to a better-coarsened graph. Moreover, we propose to add a graph connectivity term to the computation of ranking scores to overcome the isolated problem which a TAP layer may suffer from. Based on the TAP layer, we develop topology-aware pooling networks (TAPNets) for network representation learning. We add an auxiliary link prediction objective to train our networks by employing the similarity score matrix generated in TAP layers. Experimental results on graph classification tasks using both bioinformatics and social network datasets demonstrate that our TAPNets achieve performance improvements as compared to previous models. Ablation Studies show the contributions of our TAP layers to network performances. We show that the trainable parameters involved in TAP layers will not cause over-fitting.

Based on this work, we discuss several possible directions for future work. First, our methods mainly address the node classification and graph classification problems. In practice, many other interesting tasks can be formulated as link prediction problems, where each edge has a label. Our current graph deep learning methods are not able to directly perform link prediction on graphs. Also, there is no specialized link prediction operations on image data. We need a layer to learn edge representations effectively, which is necessary for link prediction. Second, our methods are mainly following the operations on computer vision. For graph data, there can be some specialized deep learning operations on graph. We will explore these directions in the future.

REFERENCES

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems*, pp. 1097–1105, 2012.

[3] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, “Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 834–848, 2017.

[4] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards real-time object detection with region proposal networks,” in *Advances in Neural Information Processing systems*, pp. 91–99, 2015.

[5] K. He, G. Gkioxari, P. Dollár, and R. Girshick, “Mask R-CNN,” in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2961–2969, 2017.

[6] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *Proceedings of the International Conference on Learning Representations*, 2015.

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9, 2015.

[8] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

[9] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder–decoder approaches," in *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pp. 103–111, 2014.

[10] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 1412–1421, 2015.

[11] J. Gehring, M. Auli, D. Grangier, and Y. Dauphin, "A convolutional encoder model for neural machine translation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 123–135, 2017.

[12] T. Wang, D. J. Wu, A. Coates, and A. Y. Ng, "End-to-end text recognition with convolutional neural networks," in *Proceedings of the 21st International Conference on Pattern Recognition*, pp. 3304–3308, IEEE, 2012.

[13] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 855–864, ACM, 2016.

[14] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations*, 2017.

[15] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations*, 2017.

[16] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *International Conference on Learning Representations*, 2015.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, pp. 6000–6010, 2017.

[18] M. Niepert, M. Ahmed, and K. Kutzkov, "Learning convolutional neural networks for graphs," in *International Conference on Machine Learning*, pp. 2014–2023, 2016.

[19] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[20] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, “Collective classification in network data,” *AI Magazine*, vol. 29, no. 3, p. 93, 2008.

[21] Z. Yang, W. Cohen, and R. Salakhudinov, “Revisiting semi-supervised learning with graph embeddings,” in *International Conference on Machine Learning*, pp. 40–48, 2016.

[22] M. Zitnik and J. Leskovec, “Predicting multicellular function through multi-layer tissue networks,” *Bioinformatics*, vol. 33, no. 14, pp. i190–i198, 2017.

[23] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[24] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *The International Conference on Learning Representations*, 2015.

[25] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

[26] B. Perozzi, R. Al-Rfou, and S. Skiena, “Deepwalk: Online learning of social representations,” in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 701–710, 2014.

[27] M. Defferrard, X. Bresson, and P. Vandergheynst, “Convolutional neural networks on graphs with fast localized spectral filtering,” in *Advances in Neural Information Processing Systems*, pp. 3844–3852, 2016.

[28] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” in *Advances in Neural Information Processing Systems*, pp. 1024–1034, 2017.

[29] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.

[30] X. Wang, R. Girshick, A. Gupta, and K. He, “Non-local neural networks,” in *The IEEE Conference on Computer Vision and Pattern Recognition*, p. 4, 2018.

[31] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, “Show, attend and tell: Neural image caption generation with visual attention,” in *International Conference on Machine Learning*, pp. 2048–2057, 2015.

[32] H. Zhao, Y. Zhang, S. Liu, J. Shi, C. Change Loy, D. Lin, and J. Jia, “Psanet: Point-wise spatial attention network for scene parsing,” in *Proceedings of the European Conference on Computer Vision*, pp. 267–283, 2018.

[33] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[34] K. Gregor, I. Danihelka, A. Graves, D. Rezende, and D. Wierstra, “Draw: A recurrent neural network for image generation,” in *International Conference on Machine Learning*, pp. 1462–1471, 2015.

[35] G. Li, X. He, W. Zhang, H. Chang, L. Dong, and L. Lin, “Non-locally enhanced encoder-decoder network for single image de-raining,” in *Proceedings of the 26th ACM international conference on Multimedia*, pp. 1056–1064, 2018.

[36] M. Jaderberg, K. Simonyan, A. Zisserman, *et al.*, “Spatial transformer networks,” in *Advances in Neural Information Processing Systems*, pp. 2017–2025, 2015.

[37] S. Shankar, S. Garg, and S. Sarawagi, “Surprisingly easy hard-attention for sequence to sequence learning,” in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 640–645, 2018.

[38] B. Yu, H. Yin, and Z. Zhu, “St-unet: A spatio-temporal U-network for graph-structured time series modeling,” *arXiv preprint arXiv:1903.05631*, 2019.

[39] J. Ling and A. Rush, "Coarse-to-fine attention models for document summarization," in *Proceedings of the Workshop on New Frontiers in Summarization*, pp. 33–42, 2017.

[40] M. Malinowski, C. Doersch, A. Santoro, and P. Battaglia, "Learning visual question answering by bootstrapping hard attention," in *European Conference on Computer Vision*, pp. 3–20, Springer, 2018.

[41] F. Juefei-Xu, E. Verma, P. Goel, A. Cherodian, and M. Savvides, "Deepgender: Occlusion and low resolution robust facial gender classification via progressively trained convolutional neural networks with attention," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 68–77, 2016.

[42] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural networks: Tricks of the Trade*, pp. 9–48, Springer, 2012.

[43] Y. Rao, J. Lu, and J. Zhou, "Attention-aware deep reinforcement learning for video face recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3931–3940, 2017.

[44] K. M. Borgwardt, C. S. Ong, S. Schönauer, S. Vishwanathan, A. J. Smola, and H.-P. Kriegel, "Protein function prediction via graph kernels," *Bioinformatics*, vol. 21, no. suppl_1, pp. i47–i56, 2005.

[45] P. D. Dobson and A. J. Doig, "Distinguishing enzyme structures from non-enzymes without alignments," *Journal of Molecular Biology*, vol. 330, no. 4, pp. 771–783, 2003.

[46] P. Yanardag and S. Vishwanathan, "A structural smoothing framework for robust graph comparison," in *Advances in Neural Information Processing Systems*, pp. 2134–2142, 2015.

[47] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification," in *Proceedings of AAAI Conference on Artificial Inteligence*, 2018.

[48] C.-C. Chang and C.-J. Lin, "Libsvm: a library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, p. 27, 2011.

[49] H. Gao, Z. Wang, and S. Ji, “Large-scale learnable graph convolutional networks,” in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1416–1424, ACM, 2018.

[50] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, “Tensorflow: a system for large-scale machine learning,” in *OSDI*, vol. 16, pp. 265–283, 2016.

[51] O. Vinyals, S. Bengio, and M. Kudlur, “Order matters: Sequence to sequence for sets,” in *Proceedings of the International Conference on Learning Representations*, 2016.

[52] Z. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec, “Hierarchical graph representation learning with differentiable pooling,” in *Advances in Neural Information Processing Systems*, pp. 4800–4810, 2018.

[53] M. Gori, G. Monfardini, and F. Scarselli, “A new model for learning in graph domains,” in *Neural Networks, 2005. IJCNN’05. Proceedings. 2005 IEEE International Joint Conference on*, vol. 2, pp. 729–734, IEEE, 2005.

[54] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, “The graph neural network model,” *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[55] M. Zhang and Y. Chen, “Link prediction based on graph neural networks,” in *Advances in Neural Information Processing Systems*, pp. 5165–5175, 2018.

[56] K. Zhou, T. P. Michalak, M. Waniek, T. Rahwan, and Y. Vorobeychik, “Attacking similarity-based link prediction in social networks,” in *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pp. 305–313, 2019.

[57] Z. Chen, J. B. Estrach, and L. Li, “Supervised community detection with line graph neural networks,” in *7th International Conference on Learning Representations*, 2019.

[58] S. Jégou, M. Drozdzal, D. Vazquez, A. Romero, and Y. Bengio, “The one hundred layers tiramisu: Fully convolutional densenets for semantic segmentation,” in *IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 1175–1183, IEEE, 2017.

[59] M. C. Golumbic, *Algorithmic graph theory and perfect graphs*. Elsevier, 2004.

[60] B. D. Thatte, “Kocay’s lemma, whitney’s theorem, and some polynomial invariant reconstruction problems,” *The Electronic Journal of Combinatorics*, pp. R63–R63, 2005.

[61] J. L. Gross and J. Yellen, *Graph theory and its applications*. CRC press, 2005.

[62] X. Jiang, P. Ji, and S. Li, “Censnet: convolution with edge-node switching in graph neural networks,” in *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pp. 2656–2662, AAAI Press, 2019.

[63] S. Zhu, C. Zhou, S. Pan, X. Zhu, and B. Wang, “Relation structure-aware heterogeneous graph neural network,” in *IEEE International Conference on Data Mining*, pp. 1534–1539, IEEE, 2019.

[64] X. Xiong, K. Ozbay, L. Jin, and C. Feng, “Dynamic prediction of origin-destination flows using fusion line graph convolutional networks,” *arXiv preprint arXiv:1905.00406*, 2019.

[65] W. Yao, A. S. Bandeira, and S. Villar, “Experimental performance of graph neural networks on random instances of max-cut,” in *Wavelets and Sparsity XVIII*, vol. 11138, p. 111380S, International Society for Optics and Photonics, 2019.

[66] T. Evans and R. Lambiotte, “Line graphs, link partitions, and overlapping communities,” *Physical Review E*, vol. 80, no. 1, p. 016105, 2009.

[67] H. Gao and S. Ji, “Graph U-Nets,” in *International Conference on Machine Learning*, pp. 2083–2092, 2019.

[68] L. Gong and Q. Cheng, “Exploiting edge features for graph neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 9211–9219, 2019.

[69] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, “How powerful are graph neural networks?,” in *The 6th International Conference on Learning Representations*, 2018.

[70] N. Shervashidze, P. Schweitzer, E. J. v. Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels," *Journal of Machine Learning Research*, vol. 12, no. Sep, pp. 2539–2561, 2011.

[71] J. Lee, I. Lee, and J. Kang, "Self-attention graph pooling," in *Proceedings of The 36th International Conference on Machine Learning*, 2019.

[72] N. Wale, I. A. Watson, and G. Karypis, "Comparison of descriptor spaces for chemical compound retrieval and classification," *Knowledge and Information Systems*, vol. 14, no. 3, pp. 347–375, 2008.

[73] H. Toivonen, A. Srinivasan, R. D. King, S. Kramer, and C. Helma, "Statistical evaluation of the predictive toxicology challenge 2000–2001," *Bioinformatics*, vol. 19, no. 10, pp. 1183–1193, 2003.

[74] H. Noh, S. Hong, and B. Han, "Learning deconvolution network for semantic segmentation," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1520–1528, 2015.

[75] H. Gao and S. Ji, "Efficient and invariant convolutional neural networks for dense prediction," in *IEEE International Conference on Data Mining*, pp. 871–876, IEEE, 2017.

[76] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 234–241, Springer, 2015.

[77] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3431–3440, 2015.

[78] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," *arXiv preprint arXiv:1704.01212*, 2017.

[79] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, “Modeling relational data with graph convolutional networks,” in *European Semantic Web Conference*, pp. 593–607, Springer, 2018.

[80] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, “Convolutional networks on graphs for learning molecular fingerprints,” in *Advances in Neural Information Processing Systems*, pp. 2224–2232, 2015.

[81] H. Dai, B. Dai, and L. Song, “Discriminative embeddings of latent variable models for structured data,” in *International Conference on Machine Learning*, pp. 2702–2711, 2016.

[82] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, “Geometric deep learning: going beyond euclidean data,” *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.

[83] M. Henaff, J. Bruna, and Y. LeCun, “Deep convolutional networks on graph-structured data,” *arXiv preprint arXiv:1506.05163*, 2015.

[84] J. Bruna, W. Zaremba, A. Szlam, and Y. Lecun, “Spectral networks and locally connected networks on graphs,” in *International Conference on Learning Representations*, 2014.

[85] F. Gama, A. G. Marques, G. Leus, and A. Ribeiro, “Convolutional neural network architectures for signals supported on graphs,” *IEEE Transactions on Signal Processing*, vol. 67, no. 4, pp. 1034–1049, 2019.

[86] M. Fey, J. Eric Lenssen, F. Weichert, and H. Müller, “Splinecnn: Fast geometric deep learning with continuous b-spline kernels,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 869–877, 2018.

[87] F. Monti, D. Boscaini, J. Masci, E. Rodola, J. Svoboda, and M. M. Bronstein, “Geometric deep learning on graphs and manifolds using mixture model cnns,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5115–5124, 2017.

[88] M. Simonovsky and N. Komodakis, "Dynamic edge-conditioned filters in convolutional neural networks on graphs," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3693–3702, 2017.

[89] A. Bronselaer and G. Pasi, "An approach to graph-based analysis of textual documents," in *The 8th European Society for Fuzzy Logic and Technology*, pp. 634–641, Atlantis Press, 2013.

[90] B. Liu, T. Zhang, D. Niu, J. Lin, K. Lai, and Y. Xu, "Matching long text documents via graph convolutional networks," *arXiv preprint arXiv:1802.07459*, 2018.

[91] R. Mihalcea and P. Tarau, "Textrank: Bringing order into text," in *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, pp. 404–411, 2004.

[92] F. Rousseau and M. Vazirgiannis, "Graph-of-word and tw-idf: new approach to ad hoc ir," in *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*, pp. 59–68, ACM, 2013.

[93] F. Rousseau, E. Kiagias, and M. Vazirgiannis, "Text categorization as a graph classification problem," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, vol. 1, pp. 1702–1712, 2015.

[94] F. D. Malliaros and K. Skianis, "Graph-based term weighting for text categorization," in *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015*, pp. 1473–1479, 2015.

[95] F. Rousseau and M. Vazirgiannis, "Main core retention on graph-of-words for single-document keyword extraction," in *European Conference on Information Retrieval*, pp. 382–393, Springer, 2015.

[96] A. Tixier, F. Malliaros, and M. Vazirgiannis, "A graph degeneracy-based approach to keyword extraction," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 1860–1870, 2016.

[97] G. Nikolentzos, P. Meladianos, F. Rousseau, Y. Stavrakas, and M. Vazirgiannis, "Shortest-path graph kernels for document similarity," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pp. 1890–1900, 2017.

[98] D. Marcheggiani and I. Titov, "Encoding sentences with graph convolutional networks for semantic role labeling," *arXiv preprint arXiv:1703.04826*, 2017.

[99] J. Bastings, I. Titov, W. Aziz, D. Marcheggiani, and K. Sima'an, "Graph convolutional encoders for syntax-aware neural machine translation," *arXiv preprint arXiv:1704.04675*, 2017.

[100] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions," in *Proceedings of the International Conference on Learning Representations*, 2016.

[101] H. Zhao, Z. Lu, and P. Poupart, "Self-adaptive hierarchical sentence model," in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, pp. 4069–4076, 2015.

[102] P. Blunsom, E. Grefenstette, and N. Kalchbrenner, "A convolutional neural network for modelling sentences," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, 2014.

[103] S. P. Chepuri and G. Leus, "Subsampling for graph power spectrum estimation," in *IEEE Sensor Array and Multichannel Signal Processing Workshop*, pp. 1–5, IEEE, 2016.

[104] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5967–5976, IEEE, 2017.

[105] J. Zhao, M. Mathieu, R. Goroshin, and Y. Lecun, "Stacked what-where auto-encoders," *arXiv preprint arXiv:1506.02351*, 2015.

[106] Y. Gong, Y. Jia, T. Leung, A. Toshev, and S. Ioffe, "Deep convolutional ranking for multilabel image annotation," in *Proceedings of the International Conference on Learning Representations*, 2014.

[107] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.

[108] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, no. 12, pp. 2481–2495, 2017.

[109] Ö. Çiçek, A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger, "3d u-net: learning dense volumetric segmentation from sparse annotation," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 424–432, Springer, 2016.

[110] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Advances in Neural Information Processing Systems*, pp. 649–657, 2015.

[111] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. Van Kleef, S. Auer, *et al.*, "Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia," *Semantic Web*, vol. 6, no. 2, pp. 167–195, 2015.

[112] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, vol. 2, pp. 427–431, 2017.

[113] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pp. 1532–1543, 2014.

[114] D. Zeng, K. Liu, S. Lai, G. Zhou, and J. Zhao, "Relation classification via convolutional deep neural network," in *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pp. 2335–2344, 2014.

[115] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning*, pp. 807–814, 2010.

[116] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[117] J. Lu, J. Yang, D. Batra, and D. Parikh, "Hierarchical question-image co-attention for visual question answering," in *Advances In Neural Information Processing Systems*, pp. 289–297, 2016.

[118] J. Carreira, R. Caseiro, J. Batista, and C. Sminchisescu, "Semantic segmentation with second-order pooling," in *European Conference on Computer Vision*, pp. 430–443, Springer, 2012.

[119] S. Ruder, "An overview of multi-task learning in deep neural networks," *arXiv preprint arXiv:1706.05098*, 2017.

[120] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *arXiv preprint arXiv:1505.00853*, 2015.

[121] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv preprint arXiv:1511.07289*, 2015.