Towards Differentiable Graph Pooling with Structural and Feature Information

Qi Huang

CONTENTS

I Abstract 4

II Introduction 4

II-A Context . . . 4

II-B Challenges . . . 5

II-C Contribution . . . 6

III Related Work 7

III-A Notation . . . 7

III-B Traditional methods in graph pooling . . . 7

III-C Graph neural network . . . 9

III-D Recent efforts in extending GNNs to pooling operator . . . 10

IV Solution 12

IV-A Motivation & Goal . . . 12

IV-B *GraphSage* . . . 12

IV-C *DiffPool* . . . 13

IV-D Min-cut & Proximal Gradient Descent . . . 13

IV-E *DiffPoolProx* . . . 15

IV-F Dataset . . . 17

V Discussion 19

V-A Dataset . . . 19

V-B Training Details & Model Configurations . . . 20

V-C Results . . . 20

V-C1 Metrics . . . 20

V-C2 Visualization . . . 20

V-C3 Analysis . . . 21

V-C4 Challenges . . . 21

VI Conclusion 23

VII Acknowledgement 25

References 25

I. Abstract

Graph Pooling is a graph analytic method in the field of graph representation learning. Starting from an original graph and signal pair, the goal is to construct a smaller graph with transformed signals that retain the desired information as much as possible. Traditional methods fall short of utilizing feature information in signal-rich graphs, while more recently learning-based analytic methods discard most structural information. In this project, we proposed a graph-neural-network-based pooling network that computes the node assignment with node feature information, and refines it with graph structural information via proximal gradient descent. With graph classification as the downstream task, our proposed method achieves favorable performance against the baseline on both real and synthetic datasets.

II. Introduction

Over the past decade we witness the academia's resurgent interest in deep learning. Most of deep learning's impressive achievements are signal processing applications with data such as images, texts, and speech, etc. A commonality of these data formats is that their underlying domains have the Euclidean structure. However, in many other cases, the data's underlying domain is non-euclidean, like graph. Recently, there have been attempts in generalizing deep learning algorithm to "learn" signals with non-euclidean domain, particularly with graph. This capstone project aims at making contributions to this on-going research effort of graph representation learning. Specifically, we explore potential improvements on graph pooling, a technique used in building graph representation learning model. In the following introduction section, we will first motivate our research in graph representation learning, and graph pooling specifically. We then proceed to explain the specific issues in graph pooling that are hard to tackle, and give an overview to the research and practical implication of graph pooling.

A. Context

In Mathematics and Computer Science, a graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". Objects in the set are represented as nodes, and relations are represented as edges. Traditionally, graphs are studied as pure structures without any signal/data defined on them. Nevertheless, in recent years graphs have increasingly become signal-rich, with both vertices and edges contain rich feature information. A large amount of data produced by nature and human activities could be structured as graphs. Notable examples include molecular graphs, relational database

graphs, social networks as graphs, etc. In practice, people are also increasingly more interested in analyzing *graph structured data* instead of *graph per se*. Application-wise, there are numerous potential applications. Learning on drugs' molecular structure could help pharmaceutical researchers to search for new drug in an high dimensional space. Learning on social network, with users as vertices and conversations as edges, could help to recommend new friends. Deploying machine learning methods on financial transaction network could help detect suspicious transactions and evaluate users' credit risk.

One open research question in graph representation learning is how we "pool" a large graph to a smaller graph. By graph pooling, we refer to the process of constructing a smaller graph with transformed signals that retain the desired information as much as possible from the original (graph, signal) pair. Commonly used graph size measures include number of vertices, or number of edges.

This specific task has both research and application implications. On one hand, pooling and convolution are two building blocks for convolution neural network (CNN) in computer vision, and are keys to the success of modern CNN architecture. Pooling greatly reduces the amount of parameters, prevents the network from over-fitting, and allows the network to learn hierarchical representation of the data. In graph representation learning, Graph Convolutional Network [1] could be regarded as a generalization of the convolution module. Nonetheless, a theoretically sound, differentiable pooling module's generalization on graph hasn't been proposed yet.

On the other hand, any potential serious industrial application of deep learning on graphs needs pooling. Graph-structured data such as social network contains billions of nodes, and as the vanilla Graph Convolutional Network's parameter count grows linearly in the number of nodes, the model's training cost could be high when a large amount of GPU instances are needed. It becomes pertinent to coarsen the graph while preserving information that's deemed important. Moreover, pooling also produces hierarchical representations for graphs, which can better inform professionals and decision makers about the structure of the graphs, bringing in new insight.

B. Challenges

A key difference between our problem setting and the traditional graph analytic setting is that our graph is signal-rich. If there is no signal residing on the graph, graph pooling degenerates into classic computer science problems like graph cut or graph coarsening. This is because on a signal-free graph, the most natural graph pooling approach is to merge connected, structurally similar nodes into a hyper-node.

However, when the graph is equipped with signals or features defined on each node/edge, the problem becomes much more complicated. For example, for a regular 10x10 grid graph, optimal k-class graph cut for any k does not exist, since there is no ”cluster” at all in this grid graph. However, if we color nodes on the left side of the graph as blue, and nodes on the right side of the graph as red, then there is a clear clustering structure – nodes with the same color should be clustered together. Node coloring is a simple type of signal on graph, and this example illustrates the difference in problem complexity between signal-rich and signal-free graph.

The major challenges not only lie in the problem setting, but also in the scarcity of related research. In related traditional algorithms such as min-cut, normalized cut, or kron reduction (what we call ”traditional methods” in the following sections), researchers focus on dealing with the graph structure per se, therefore the graph is signal-free. As a result, when we blindly apply traditional methods to ”pool” signal-rich graphs, these methods fall short of utilizing feature information and achieve poor performance. On the other hand, there have been some learning-based graph pooling method, but they mainly utilize feature information while discarding most of the structural information.

C. Contribution

In this project, our main contribution is a new learning-based graph pooling method that aims at incorporating both structural and feature information of signal-rich graphs. Built on *DiffPool*, a recently proposed learning-based graph pooling method, we tweak the neural network structure and add a refinement module to incorporate structural information into the pooling process. Specifically, we optimize the initial graph pooling result by minimizing a min-cut based auxiliary loss function via proximal gradient descent. Moreover, we also propose two synthetic datasets to better test the performance of different graph pooling methods. With graph classification as the downstream task, our proposed method achieves favorable performance against the baseline on both synthetic datasets and real datasets used by the baseline graph pooling method.

This paper proceeds as follows. In section II, we will examine related literature, along both the traditional line of work and the more recent learning-based line of work. In section III, we will first explain the motivation of our work in devising the refinement mechanism as well as construction of the synthetic datasets. We then introduce readers to mechanisms of basic components of our model, i.e. graph neural network and proximal gradient descent. Then, we will describe our proposed learning-based

graph pooling model, *DiffPoolProx*, as well as our synthetic datasets' designs. Section IV will present our experiment results. In section V, we will summarize the project, and discuss future work. Section VI is acknowledgement.

III. Related Work

Our work is conceptually related to a series of traditional methods, graph neural networks (GNNs), and recent advancements in designing the pooling operator on graph using GNNs.

A. Notation

The following notations will be used consistently throughout the paper to provide a unified ground for all of the following discussions. We focus on un-directed, connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a finite vertex set with cardinality $|\mathcal{V}| = n$, $\mathcal{E}$ is the related edge set, and $\mathcal{A} \in \mathcal{R}^{n \times n}$ is the induced adjacency matrix that encodes connectivity information in $\mathcal{G}$. A graph signal $X : \mathcal{V} \to \mathcal{R}^m$ is a function defined on nodes that maps each node to an embedding vector of dimension m. A graph's laplacian L is defined as $L = D - A$, where D is the diagonal degree matrix $D := \mathcal{A}1$, and $\mathcal{A}$ is the adjacency matrix. For the ease of exposition, we define the task of "graph pooling" in its broadest sense, i.e., to find a mapping $f : G \to G^{'}$, where $G^{'} = (\mathcal{V}^{'}, \mathcal{E}^{'})$ with an induced adjacency matrix $\mathcal{A}^{'}$, and $(\mathcal{V}^{'}, \mathcal{E}^{'})$ where their sizes are smaller than the original $\mathcal{V}, \mathcal{E}, \mathcal{A}$, respectively.

B. Traditional methods in graph pooling

On the traditional method line of work, there exists a wide spectrum of algorithms that relate to graph pooling with various problem settings. For example, as we argue in Section I, the *graph cut* problem is one of the simplified forms of graph pooling when we only care about constructing $\mathcal{V}^{'}$ from $\mathcal{V}$. Well-known algorithms like normalized cut [2] work by clustering structurally similar nodes together. *Graph Sparsification* seeks to pool the graph by only reducing the number of edges [3] [4] [5]. Nonetheless, graph sparsification algorithms only reduce the edge set $\mathcal{E}$ but not the vertex set $\mathcal{V}$, which only makes the graph "sparser" instead of "smaller". Thus, graph sparsification methods more serve as approaches to reduce the computational burden for other graph signal processing algorithms [6].

Some other traditional methods are categorized as *Graph Reduction*, which generally follow a "partition – contract – lifting" pipeline. In *Graph Reduction*, one seeks to first find a collection of candidate contraction sets $\mathcal{P}$ of $\mathcal{V}$. Then, the algorithm forms a new graph $G^{'}$ through re-combination of vertices

in each contraction set $\mathcal{C}$ in $\mathcal{P}$, followed by edges' re-wiring or edges' filtering and node embedding's lifting.

Comparing to Graph Sparsification, these methods reduce both the vertex set $\mathcal{V}$ and the edge set $\mathcal{E}$. Therefore, they could be performed iteratively, enabling the construction of hierarchical graph representations [7] [8].

Among existing graph reduction literature, *Heavy Edge Matching* is a heuristic that is widely adopted [9]. All pairs of connected nodes $e_{ij} = (v_i, v_j)$ are first considered as candidate contraction sets for graph pooling. The algorithm then assigns contraction scores of $w_{ij}/max(d_i, d_j)$ to candidate contraction sets, i.e. , edges, where w_{ij} is the edge weight and d_i, d_j are the degrees of node i, j, respectively. Edges are then sorted by their contraction scores in descending order, among which edges with higher scores are contracted first. Intuitively, this approach prioritizes contracting nodes with higher connection weights, adjusted by their overall connectivity. It relies on a natural assumption that the edge weight is a measure of node similarity, and similar nodes should be contracted together. *Kron Reduction* reduces a graph by finding the Schur complement of the original graph laplacian matrix L with respect to the laplacian $L_{\mathcal{V}^c,\mathcal{V}^c}$[1] [10]. The Schur complement is then taken as the graph laplacian of the reduced graph. Comparing to heuristics like Heavy Edge Matching, *Kron Reduction* provides theoretical guarantees on the spectral similarity between the two graphs in sacrificing the computation complexity ($\mathcal{O}(|\mathcal{V}|^3)$. Moreover, Kron Reduction also produces significantly denser graphs. Thus, it is usually used with graph sparsification when performing graph pooling [9]. One of the latest research effort in this direction, *Local Variation* proposed by Loukas & Vandergheynst, coarsens the graph by first computing a local variation of each given contracting set. Local Variation then greedily contracts current contraction sets to form the coarsening matrix C to lift the original graph's Laplacian L and the nodes feature matrix X [11]. Comparing to Kron Reduction, Local Variation provides an even stronger graph spectral similarity guarantee with nearly-linear computational complexity, and does not sacrifice graph sparsity. From Heavy Edge Matching to Local Variation, traditional methods display a clear trend in reduced computation complexity and stronger spectral properties guarantees. Nonetheless, to our best knowledge, existing deterministic graph pooling algorithms only incorporate structural information and do not adapt to data, i.e., node features/signals. Therefore, they only work on signal-free graphs, and will likely fail when pooling signal-rich graphs when

[1] $L_{\mathcal{V}^c,\mathcal{V}^c}$ is the laplacian of the sub-graph formed by left-out vertices $\mathcal{V}^c$

signals/data plays an influential role in determining the actual clusters.

C. Graph neural network

There has always been an interest in bringing deep learning success to non-euclidean domain, particularly in the domain of graph. Like the convolutional neural network in computer vision aims to compute high-dimensional embedding for natural images, a neural network defined on graph aims to compute high-dimensional embedding for nodes/edges in the graph, or the whole graph in general. Two major approaches, i.e., spectral construction and spatial construction, are adopted in defining neural network module on graph. Spectral construction approach explores extending the convolutional neural network to graphs by directly defining the convolution operator on graph domain. Bruna et.al. first proposed a definition of convolution on graph by the convolution theorem and graph Fourier transform [12]. Deferrard et.al. simplified the original graph convolution method by approximating the non-parametric spectral filter function with kth order Chebyshev polynomials [7]. Kipf & Welling proposed Graph Convolutional Network (GCN) to further simplify graph convolution by only taking the first order approximation of the filter function [1]. To enable the model to learn non-localized information, arbitrary GCN layers are stacked together. The non-linearity added to each layer further makes the model more expressive. In the spatial construction line of work, similar graph neural network designs can be dated back to early 2000s. Instead of defining a series of operators to filter signals on the graph level, [13] and [14] defined embedding aggregation and embedding transformation function on the vertex level. Each node's embedding is then iteratively refined by exchanging embedding with neighbouring nodes until convergence. Notable following work includes [15] [16]. Comparing to spectral approach models, spatial approach models do not involve expensive any eigendecomposition and define operations on the vertex level. This makes them easier to be trained and more expressive. Nevertheless, GNNs to this stage is still limited with respect to its input. It's worth-noting that all of the graph neural network models above only perform transductive learning, i.e., learning on a fixed graph, which is a significant drawback if the problem setting is inherently inductive (dynamic), e.g., soical network. Moreover, graphs are also assumed to be homogeneous, and all neighbours' information is aggregated with equal importance across every vertex. GNNs to this stage also suffer from scalability issue, as the number of aggregation/transformation needed grows exponentially in $\mathcal{O}(|\mathcal{V}|)$.

Several recent works seek to address the above issues. R-GCN incorporates edge type information, thus enabling learning on heterogeneous graphs [17]. Adaptive sampling and control variate sampling schemes

were proposed to sample neighbours when performing embedding aggregation [18] [19]. *GraphSage* proposed in [20] makes further abstraction of previous GNNs models and develops a more general and expressive architecture with various choices of aggregator functions. Variants of *GraphSage* also enable inductive learning and training by neighbour sampling, achieving state-of-the-art results on graph classification/node classification across several datasets. As GNN overcomes its limitations on data, the next natural question is how to generalize beyond convolving embedding vectors to summarize node level information.

D. Recent efforts in extending GNNs to pooling operator

Besides convolution, another basic building block in deep neural network is the pooling module. As being point out in [8], pooling is the key to CNN's scale-invariant and rotation-invariant learning ability. A natural question then is how to generalize the pooling operator to the graph domain. Some existing Graph Neural Network pipelines use off-the-shelf deterministic heuristics/algorithms like Kron reduction to perform graph reduction [7]. Others follow a more rudimentary approach, i.e., simply pooling all nodes' embedding globally into a single embedding vector, thus lossing all structural information [20]. Both designs make the pooling operation fixed and not adapted to the data.

Most recently, there are several attempts in introducing a differentiable graph pooling operator into the Graph Neural Network framework. *SortPooling* first learns node embedding by applying stacked GCN layers on graphs, then sorts nodes according to the last layers' embedding. It then flattens any graph to a 1-D array of node representations, and feed the resulting array to a downstream traditional CNN model [21]. This idea equates the pooling operation as an operation to find a consistent node level order across different graphs, thus transforming a pooling problem into a node sorting one. Although this design only adds negligible computational burden to the existing graph neural network pipeline, it fails in producing a hierarchical representation of any given graph. As being pointed out in [22], finding a canonical ordering among nodes is itself a very difficult task, and equivalents to solving the graph isomorphism problem.

Top-k pooling proposes a sparsity-preserved pooling scheme [23]. Largely based on graph U-net's architecture, top-k pooling projects each node's embedding vector against a learnable projection vector, and sorts nodes based on their projection results. The top-k nodes' embedding is then preserved as the "pooled" representation of the original graph. Similar to SortPool, top-k pooling also transforms the pooling into sorting, while changing the scoring function to a projection vector shared across nodes.

TABLE I
SUMMARY OF GRAPH POOLING METHODS

Models	Learning-based	Structural Info	Feature Info	Complexity	Sparsity	hierarchical
Heavy Edge Matching	No	Yes	No	O($\|\mathcal{E}\|$)	Yes	No
Kron Reduction	No	Yes	No	O($\|\mathcal{V}\|^3$	No	Yes
Local Variation	No	Yes	No	O($\|\mathcal{V}\|$)	Yes	Yes
SortPool	Yes	No	Yes	O($\|\mathcal{V}\|$)	Yes	No
Top-k pooling	Yes	No	Yes	O($\|\mathcal{V}\|$)	Yes	Yes
DiffPool	Yes	No	Yes	O($\|\mathcal{V}\|^2$)	No	Yes

While top-k pooling is efficient and can yield sparse pooling assignments, it implicitly replaces the idea of graph pooling with graph sub-sampling. Top-k pooling may learn to pick structurally important nodes in a hierarchical manner. However, it will not yield meaningful *pooling* results, as not-selected nodes are simply *discarded* instead of pooled.

DiffPool proposed in [22] directly produces a soft assignment matrix S using GNNs. Essentially, *DiffPool* considers each node's soft assignment vector as a type of node embedding, then train a separate graph neural network to produce such embedding. Following approaches taken in the traditional graph reduction literature, adjacency matrix A_l and node embedding X_l are then lifted to produce the coarsened graph. *DiffPool* also adds an entropy minimizer and a link-prediction term to encourage the soft assignment to converge to a hard decision. Comparing to previous methods, *DiffPool* produces hierarchical representations and learns a meaningful graph soft clustering on each level. Moreover, it can be easily integrated with different GNN designs by replacing the underlying GNN module.

Nevertheless, there are still several key drawbacks in *DiffPool*. *DiffPool* only incorporates node embedding vectors that are 1-hop localized when producing the assignment matrix, leaving out global feature information and structural information. *DiffPool*'s assignment matrix is also inherently dense, which makes the assignment result hard to interpret. Furthermore, we argue that the datasets used in *DiffPool*, including ENZYMES and COLLAB, lack clearly defined hierarchical structure. It's not fully clear why a neural network model with hierarchy detection capacity/pooling capacity should outperform a baseline model, for example, *GraphSage*. Therefore, to better test a network's graph pooling performance, we deem it necessary to construct synthetic datasets dedicated to this purpose.

IV. SOLUTION

A. Motivation & Goal

In light of the above, our project aims to:

1) Improve current learning-based graph pooling methods by incorporating structure and global feature information.

2) Explore possible designs of synthetic datasets that can better test a model's graph pooling capacity.

To achieve these goals, we propose an improved version of the*Diffpool* model, *DiffpoolProx*. *Diffpool-Prox* aims at incorporating both feature and structural information into the model. It iteratively refines the pooling result by minimizing a min-cut based criterion via proximal gradient descent. The graph neural network module adopted in *DiffpoolProx*, same as *Diffpool*, is *GraphSage*. In the following section, we first briefly review *GraphSage* and the original *Diffpool* model. We then move on to a short introduction to min-cut and proximal gradient descent. Finally, we describe our proposed *DiffpoolProx* model.

B. GraphSage

Same as *Diffpool* [22], *DiffpoolProx* also considers node assignment as a type of node embedding. Therefore, it also uses graph neural network as the base node embedding extractor to produce the node assignment matrix. A node embedding extractor takes (graph, data) pair $(\mathcal{G}(\mathcal{V}, \mathcal{E}), \mathbf{X})$ as input, and computes a new embedding h_i for each node $v_i \in \mathcal{V}$. Write it succinctly, we have $H = f(\mathcal{G}, \mathbf{X}, W)$, where W is the model's parameter and $H \in \mathcal{R}^{|\mathcal{V}| \wedge k}$ is the matrix representation for all nodes' embedding. Specifically, we use *GraphSage* [20], a variant of the original GCN. The core idea of *GraphSage*'s node embedding generation process is "aggregation of the neighbor nodes' information". In another word, for any given node in the graph, its embedding will be a weighted combination of transformed neighboring nodes' embedding. Given a pre-specified depth K and an aggregator function $AGGREGATE$, *GraphSage* performs K rounds of embedding aggregations using $AGGREGATE$. Therefore, neighboring nodes' feature information within K-hop will be incorporated to produce the final embedding. Algorithm 1 gives a formal definition of *GraphSage*.

The aggregator function $AGGREGATE$ could be any function that's permutation-invariant with respect to the input feature vector set $\{\mathbf{h}_u^k, \forall u \in \mathcal{N}(v)\}$. In both *DiffPool* and *DiffPoolProx*, global mean pooling (taking the column-wise mean of the embedding matrix H) is used.

Algorithm 1 GraphSage

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; depth $\mathbf{K}$; input features $\{\mathbf{x}_v \in R^t, \forall v \in \mathcal{V}\}$, or equivalently $\mathbf{X} \in R^{|\mathcal{V}| \times t}$; weight matrix $\mathbf{W} \in R^{t \times k}$; non-linearity σ; differentiable aggregator functions $AGGREGATE$; neighborhood function $\mathcal{N} : v_i \rightarrow \{v_j, \forall v_j \in \mathcal{V} s.t. e_{ij} \in \mathcal{E}\}$

Output: Vector representations $\mathbf{z}_v \in R^k$ for all $v \in \mathcal{V}$, or equivalently the feature matrix $\mathbf{Z} \in R^{|\mathcal{V}| \times k}$

procedure GRAPHSAGE
  $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$
  for k = 1...K **do**
    for $v \in \mathcal{V}$ **do**
      $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow AGGREGATE(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / ||\mathbf{h}_v^k||_2, \forall v \in \mathcal{V}$
  $\mathbf{z} \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

C. *DiffPool*

In *DiffPool*, a soft assignment matrix S is learned by $S = GNN_{pool}(\mathcal{G}, H, W_{pool})$, where H is the current node embedding matrix for $\mathcal{G}$, and GNN_{pool} is a graph neural network. A pooling layer than coarsens the input graph, generating a new coarsened adjacency matrix A_{pool} as well as the coarsened graph's node embedding matrix by:

$$A_{pool} = S^T A S \tag{1}$$

$$H_{pool} = S^T H \tag{2}$$

$$\tag{3}$$

Notice that H is computed by another GNN dedicated to produce the node embedding matrix on the **unpooled** graph by $H = GNN_{embedding}(\mathcal{G}, X, W_{pool})$, where X is the input node feature matrix.

D. *Min-cut & Proximal Gradient Descent*

Our goal is to devise a pooling refinement process that incorporates both structural and node feature information beyond 1-hop. To incorporate structural information, a feasible approach is to refine the assignment matrix produced by the neural network by updating it with its gradient from a chosen structure-based loss function. A good candidate is the *min-cut* criterion. In graph theory, a *cut* is a partition of a vertex set into disjoint subsets. When fixing a cut, we naturally obtain a clustering of a graph's nodes. Take an arbitrary graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ as an example. If we define s_i as the ith node's clustering assignment, *cut score* is defined as the sum of all edge weights (edge counts if there is no edge weight) across clusters.

Mathematically, $Cut := \sum_{i,j \in \mathcal{V}} e_{ij} 1_{s_i \neq s_j}$. If we extend the assignment s_i into an one-hot vector, or even further into a soft assignment vector, then a vectorized representation of cut score is $Cut = trace(S^T LS)$, where the ith row in S is the assignment vector of node n_i, and L is the graph laplacian of graph $\mathcal{G}$. *Min-cut* refers to the target function of minimizing the cut score of a graph $min_S trace(S^T LS)$. Naturally, if more structurally similar nodes are clustered together, the cut will cut across fewer edges, and the cut score will be smaller.

In our case, the structure-based loss function contains both differentiable and non-differentiable terms. This create a problem as we cannot compute the gradient and update the assignment matrix. A method called *Proximal Gradient Descent* could help us solve this problem.

For a decomposable function $f(x) = g(x) + h(x)$, where $g(x)$ is differentiable and $h(x)$ is non-differentiable, we can use quadratic approximation on $g(x)$ to find the correct updating step for $f(x)$:

$$\begin{aligned} x^+ &= argmin_z g(x) + \nabla g(x)^T (z - x) + \frac{1}{2t} ||z - x||_2^2 + h(z) \\ &= \frac{1}{2t} ||z - (x - t\nabla g(x))||_2^2 + h(z) \end{aligned}$$

where we replace $\nabla^2 g(x)$ by $\frac{1}{t} I$. Intuitively, this could be understood as making the update stay close to the gradient update of $h(x)$, while minimizing the non-differentiable function h.

Define the *proximal mapping* as a function of h and t such that:

$$prox_{h,t}(x) = argim_z \frac{1}{2t} ||z - x||_2^2 + h(z) \tag{4}$$

Then, the gradient update on f could be written as:

$$x^+ = prox_{h,t}(x - t\nabla g(x)) \tag{5}$$

Therefore, proximal gradient descent could be defined as follows:

$$x^{(k)} = prox_{h,t_k}(x^{(k-1)} - t_k \nabla g(x^{(k-1)})) \tag{6}$$

where we can choose the initial $x^{(0)}$. Notice that proximal operator $prox_{h,t}$ is agnostic with the choice of g. Therefore, once we find a good h which has a closed form proximal operator, we can compute the proximal gradient for any differentiable function g regularized by h.

E. *DiffPoolProx*

In *DiffPoolProx*, we choose the min-cut criterion as the refinement objective function. However, to make sure that the assignment matrix after the refinement step still contains feature information, we want to ensure that it does not deviate too much from the initial assignment matrix. Meanwhile, ideally we also want to encourage sparsity on the assignment matrix. Motivated by these two requirements, we add two regularization terms on min-cut and derive the following objective function:

$$r(S) = trace(S^T LS) + \frac{1}{2}||S - S_0||_2^2 + \lambda||S||_1 \tag{7}$$

Where S_0 is the initial assignment matrix produced by the graph neural network. The frobenius norm of the difference between the current assignment matrix and the original refinement matrix controls their distance. The 1-norm on S encourages sparsity. To apply proximal gradient descent, we consider the first two terms as the differentiable function $g(S)$, and the last term as the non-differentiable function $h(S)$. We then discover the proximal operator of r(S) is exactly the soft-thresholding operator as the following:

$$[prox_{\lambda||.||_1,t}(S)]_{ij} = [T_{t\lambda}(S)]_{ij} = \begin{cases} S_{ij} - t\lambda & \text{if } S_{ij} > t\lambda \\ S_{ij} & \text{if } -t\lambda \leq S_{ij} \leq t\lambda \\ S_{ij} + t\lambda & \text{if } S_{ij} < -t\lambda \end{cases}$$

For $g(S) = trace(S^T LS) + \frac{1}{2}||S - S_0||_2^2$, the gradient $\nabla g(S) = LS + L^T S + 2(S - S_0)$. Therefore, the whole proximal gradient update step for our objective function $r(S)$ is:

$$T_{t\lambda}(S - t(LS + L^T S + 2(S - S_0))) \tag{8}$$

In another word, we compute S_{k+1} with S_k:

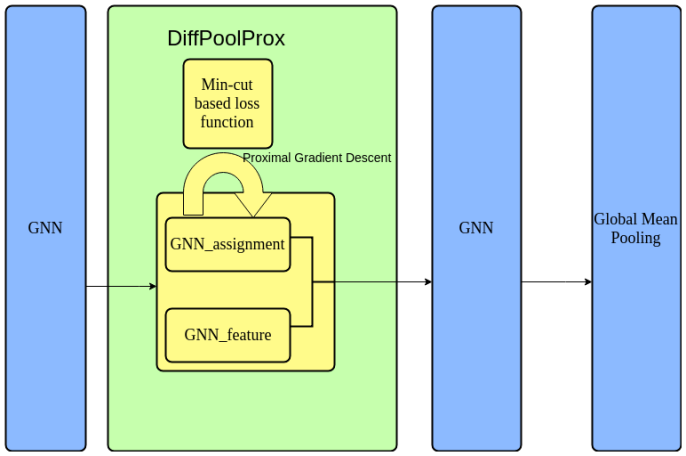
$$S_{k+1} = T_{t\lambda}(S_k - t(LS_k + L^T S_k + 2(S_k - S_0))) \tag{9}$$

, and we refine k steps for each pooling layer. Besides the above refinement mechanism, we also tweak the neural network architecture to incorporate more global feature information. There are two variants of our improvement scheme. For one, we replace the one-layer *GraphSage* that generates the assignment matrix with a two-layer *GraphSage* to incorporate 2-hop feature information. For another, we add a global

mean-pooling between the two *GraphSage* layers. We then concatenate the global readout to the output embedding of the first layer, and feed the concatenation result to the second layer. In this way, we propogate feature information on a global level. In both variants, we use the embedding matrix computed by the second *GraphSage* layer as the initial assignment matrix. These two variants are called *DiffPoolProx-2hop* and *DiffPoolProx-global*, respectively.

We give a pseudo-code version for *DiffPoolProx-global*. For *DiffPoolProx-2hop*, we simply skip line 3. Figure 1 illustrates our model design. The *DiffPoolProx-global* model is then defined as the following:

Fig. 1. *DiffPoolProx* model design



Algorithm 2 DiffPoolProx-global

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with adjacency matrix A; input features $\{\mathbf{x}_v \in R^t, \forall v \in \mathcal{V}\}$; or equivalently $\mathbf{X} \in R^{|\mathcal{V}| \times t}$; graph convolution module $GraphSage$; pool ratio ρ; refinement step k; refinement parameter λ

Output: A coarsened weighted complete graph $\mathcal{G}_{pool}(\mathcal{V}_{pool}, \mathcal{E}_{pool})$ with an induced adjacency matrix A_{pool} and nodes' embedding matrix $X_{pool} \in R^{|\mathcal{V}| \times t}$

1: **procedure** DIFFPOOLPROX
2: $S_{intermediate} \leftarrow$GraphSage$_{intermediate}(\mathcal{G}, X, W_{intermediate})$
3: $S_{globalmean} \leftarrow$GlobalMeanPooling(S$_{intermediate}$)
4: $S_{preRefined} \leftarrow$GraphSage$_{pool}(\mathcal{G}, Concatenate(S_{intermediate}, S_{globalmean}), W_{pool})$
5: **for** i = 1...k **do**
6: $S_{iter} \leftarrow$S$_{preRefined}$
7: $S_{iter} \leftarrow$ ***ProxRefine***$(\mathcal{G}(\mathcal{V}, \mathcal{E}, A), S_{iter}, S_{preRefined}, \lambda)$
8: $S \leftarrow S_{iter}$
9: $A_{pool} \leftarrow$S$^T AS$
10: $Z \leftarrow$GraphSage$_{embedding}(\mathcal{G}, X, W_{embedding})$
11: $Z_{pool} \leftarrow$S$^T Z$
12: Construct $\mathcal{G}_{pool}$ from (A_{pool}, Z_{pool})

Algorithm 3 ProxRefine

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with adjacency matrix A; current graph assignment matrix S; initial graph assignment matrix S_0; refinement parameter λ
Output: Refined graph assignment matrix $S^{'}$

1: **procedure** PROXREFINE
2: $D \leftarrow A\mathbf{1}$
3: $L \leftarrow D - A$
4: $S^{'} \leftarrow T_{t\lambda}(S - t(LS + L^T S + 2(S - S_0)))$

F. Dataset

As we argue in previous sections, current datasets used by *DiffPool* lack clearly defined hierarchical substructures. *DiffPool* outperforms models like *GraphSage*. Nonetheless, it's not clear whether the increase in performance comes from the pooling mechanism or simply from an increase in model complexity. An ideal synthetic dataset for graph pooling should satisfy the following design principles:

1) Graphs in the dataset have clearly defined hierarchical substructures.

2) Detection of such hierarchical substructures has a direct positive influence on the downstream task.

Base on these two principles, we propose two synthetic datasets, *ENZYMES-H* and *ENZYMES-E* with graph classification as the downstream task. *ENZYMES-H* is designed to test graph pooling method's capacity in handling graphs with hierarchical substructures. *ENZYMES-E* is designed to test the model's capacity in handling scale-variant graphs. In *ENZYMES-H*, we randomly select component graphs from different classes in the ENZYMES dataset, and sparsely connect them together into composite graphs. The connection probability between a pair of component graphs ($\mathcal{G}_A$, $\mathcal{G}_B$) is determined by the product of the pre-specified link probabilities for both classes, l_{pi}, l_{pj}. The label is then set according to the ratio of component graph classes in the composite graph. For example, assuming we have three classes of component graphs, $A, B,$ and C. Then label 1 will be assigned to composite graphs with component graphs ratio $A : B : C = 6 : 3 : 1$. In *ENZYMES-E*, for each original graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ in *ENZYMES*, we randomly select a scaling factor $e \in N, e > 1$, and expand every node $v \in \mathcal{V}$ into a clique of size e.

The intuition behind such designs is the following. In *ENZYMES-H*, labels are assigned according to component graph ratios, therefore a component graph's class is determined by both its feature and its structural information. Therefore, in order to learn the component graph ratios, the classifier needs to first learn both the feature and structural characteristics of each component graph to determine its class. In *ENZYMES-E*, class labels for all expanded graphs are tied to the original, pre-expanded graphs.

Algorithm 4 ENZYMES-H Generator

Input : Hyper-graph classes **K**; hyper-graph per class **H**; Set of component graph generator $\hat{\mathcal{G}}$; hyper-graph size **S**; hyper-graph divide ratio $\boldsymbol{R}$: $(\boldsymbol{K}, \boldsymbol{Y}) \rightarrow [\mathbf{0}, \mathbf{1}]$; component graph classes **Y**
Output : Set of generated hyper-graph $\boldsymbol{\mathcal{G}_{hyper}}$

procedure ENZYMES-H
  $\boldsymbol{\mathcal{G}_{hyper}} \leftarrow []$
  for k = 1...K **do**
    for h = 1...H **do** $\boldsymbol{\mathcal{G}_{component}} = []$
      for y = 1...Y **do**
        $\boldsymbol{s_y} = \boldsymbol{Round(R(k, y))}$
        for i=1...$\boldsymbol{s_y}$ **do**
          $\boldsymbol{G} \sim \hat{\boldsymbol{\mathcal{G}}}_{\boldsymbol{y}}$
          $\boldsymbol{\mathcal{G}_{component}.insert(G)}$
      $\boldsymbol{G} \leftarrow \boldsymbol{Combine(\mathcal{G}_{component})}$
      $\boldsymbol{\mathcal{G}_{hyper}.insert(G)}$

Algorithm 5 Combining Component Graph

Input:List of component graph $\mathcal{G} = [G_1(V_1, E_1, X_1, y_1), G_2(V_2, E_2, X_2, y_2), ...G_n(V_n, E_n, X_n, y_n)]$; List of link probability $L_p = [l_{p1}, l_{p2}, ..., l_{py}]$
Output : Hyper graph G_{hyper}

procedure COMBINE
  $G_{hyper} \leftarrow \{G_1, G_2, ..., G_n\}$
  for i=1...n **do**
    for j=i...n **do**
      $l_{linkpred} \leftarrow l_{pi} \cdot l_{pj}$
      if $Bernoulli(l_{linkpred})$ **then**
        $AddEdge(G_i, G_j)$

Algorithm 6 ENZYMES-E Generator

Input : Original graphs $\{\mathcal{G}\}_m$; node embedding matrices $\{X\}_m$
Output : Expanded graphs $\{\mathcal{G}_e\}_m$; node embedding matrix $\{X_e\}_m$

procedure ENZYMES-E
  $\{\mathcal{G}_e\}_m \leftarrow []$
  $\{X_e\}_m \leftarrow []$
  for i = 1...m **do**
    $e \sim N$ ▷ Sample a random expand size
    $\mathcal{G}_e^{(i)} \leftarrow Expand(\mathcal{G}^{(i)}, e)$ ▷ Expand each node in $\mathcal{G}^{(i)}$ into a size e clique
    $\{\mathcal{G}_e\}_m.append(\mathcal{G}_e^{(i)})$
    $X_e^{(i)} \leftarrow []$
    for $j = 1...|\mathcal{V}_{\mathcal{G}^{(i)}}|$ **do** ▷ ith graph's vertex set
      for k = 1...e **do**
        $p \sim \{0, 1\}^k$ ▷ sample a k-dimension random noise uniform on [-1,1]
        $x_j^k \leftarrow x_j^{(i)} + p$
        $X_e^{(i)} \leftarrow Concatenate(X_e^{(i)}, x_j^k)$
    $\{X_e\}_m.append(X_e^{(i)})$

Therefore, in order to learn the class labels for expanded graphs, the classifier needs to learn substructures of expanded graphs, and merge each clique back to a node.

V. Discussion

A. Dataset

ENZYMES ENZYMES is a dataset of protein tertiary structures obtained from [24]consisting of 600 enzymes from the BRENDA enzyme database [25]. The graph classification task here is to correctly assign each enzyme to one of the 6 EC top-level classes.

COLLAB COLLAB is a scientific-collaboration dataset, derived from 3 public collaboration datasets in the fields of High Energy Physics, Condensed Matter Physics and Astro Physics. In [26], the dataset is consructed by first generating ego-networks of different researchers from each field, and labeling each graph as the field of the researcher. The task is to classify the ego-collaboration graph of a researcher to the fields of High Energy, Condensed Matter or Astro Physics field.

PROTEINS PROTEINS is a dataset obtained from [24]. In PROTEINS, nodes represent secondary structure elements (SSEs). There is an edge between two nodes if they are neighbors in the amino-acid sequence or in 3D space. Graphs in this dataset have 3 possible labels, representing helix, sheet or turn.

ENZYMES-H Our *ENZYMES-H* dataset contains 840 graphs, each with 10 components graphs. We randomly chose 3 classes from the original *ENZYMES* dataset as component graphs, and connected component graphs into composite graphs in *ENZYMES-H*. There are 3 classes in *ENZYMES-H*, corresponding to component graph ratio $\{0.6, 0.2, 0.2\}, \{0.2, 0.6, 0.2\}, \{0.2, 0.2, 0.6\}$, respectively.

ENZYMES-E Our *ENZYMES-E* dataset contains 600 graphs. For each graph in the original *ENZYMES* dataset, we randomly chose an expand size e within $[1, 10]$, and expanded each node in the original graph into a clique of size e. the new, expanded graphs' labels are the same as the original graphs' labels.

We benchmarked the baseline *DiffPool* and two variants of our model *DiffPoolProx-2hop*, *DiffPoolProx-global* with the graph classification task. The tested dataset were *ENZYMES-E* together with *ENZYMES*, *COLLAB*, and *PROTEIN* used in the original *DiffPool* paper. It's expected that the our improved *DiffPoolProx* model should have comparable or higher performance on all datasets comparing to *DiffPool*. Moreover, we expect a performance drop between *ENZYMES-E* and ENZYMES for *DiffPool* since we randomly expanded training graphs in *ENZYMES-E*. However, we predict *DiffPoolProx* will suffer less performance decrease comparing to *DiffPool*.

B. Training Details & Model Configurations

Both *DiffPoolProx* and *DiffPool* use *GraphSage* architecture as the building block GNN module. Both use the mean-pooling variant $AGGREGATE$ module in *GraphSage*. For *ENZYMES* and *ENZYMES-E*, we stacked two *GraphSage* layers and a *DiffPool*/*DiffPoolProx* layer, followed by two *GraphSage* layers to further refine nodes embedding. For *COLLAB* and *PROTEINS*, we used two *DiffPool*/*DiffPoolProx* layers, each is followed by a *GraphSage* layer. In all DiffPool/*DiffPoolProx* layers, the pooling ratio was set as 50% of the maximum number of nodes. We also applied batch normalization [27] after each *DiffPool*/*DiffPoolProx* layers. A linked prediction auxiliary function $Lp = ||A - S^T S||_F$ was also added after each *DiffPool*/*DiffPoolProx* layer. *ENZYMES* was repeated 6 times and we took the average validation accuracies as the final reporting result. Due to time constraint, *ENZYMES-E*, PROTEINS were repeated 2 times, and *ENZYMES-H*, *PROTEINS* were run for one time.[2]. All datasets were randomly splitted into training sets and valiation sets. For *PROTEINS*, we used early stopping when validation accuracy started decreasing, as the graphs in *PROTEINS* are relatively simple. For synthetic dataset *ENZYMES-H* and *ENZYMES-E*, we ensured that the no graphs with the same origin, i.e. same component graphs in *ENZYMES-H* or same original graphs in *ENZYMES-E* appeared in both dataset. In another word, we ensured models that were trained on the training datasets should have no prior knowledge on validation datasets.

C. Results

1) Metrics: We used the validation accuracy on graph classification task as the metrics. Table II illustrates our final result. We compared the classification accuracy results achieved by two variants of our proposed model, *DiffPoolProx-2hop* and DiffPoolProx-global as well as the baseline *DiffPool* model across all datasets. The best result on each dataset is highlighted in bold.

2) Visualization: To better understand and compare different models' learning processes, we visualized their learned node assignment results when doing inference on the validation set. Since the output assignment matrix is soft, we obtain an hard assignment result for each node by taking the index corresponding to the maximum assignment weight across each row. Figure 2 to 4 present the hard

[2]We will update the experiment results in the future

TABLE II
GRAPH CLASSIFICATION ACCURACY

Models	*ENZYMES*	*ENZYMES-E*	*COLLAB*	*PROTEINS*	*ENZYMES-H*
DiffPool	52.47%	45.28%	68.99%	73.37%	**48.88%**
DiffPoolProx-global	50.49%	46.39%	65.35%	73.62%	39.54%
DiffPoolProx-2hop	**53.09%**	**53.61%**	**70.90%**	**74.41%**	47.12%

assignments' visualizations on a random graph from *ENZYMES*. FIgure 5-7 show the hard assignments' visualizations on a random graph from *ENZYMES-E*. We chose these two specific datasets because graphs of these two datasets are small enough and can be easily visualized, yet at the same time not too simple. All visualizations were rendered using Matplotlib and TensorBoardX.

3) Analysis: Both variants of our proposed *DiffPoolProx* achieved comparable/favorable performance comparing to the baseline *DiffPool*, with *DiffPoolProx-2hop* performing the best across almost all datasets. It's quite remarkable that *DiffPoolProx-2hop*'s classification accuracy on *ENZYMES-E* doesn't drop comparing to *ENZYMES*. This lends support to our claim that *DiffPoolProx-2hop* is more resilient to training graphs' size variance. From the hard assignment visualizations, we observe that *DiffPoolProx-2hop*'s assignment results are more consistent comparing to the baseline model *DiffPool*. To wit, neighboring nodes are more likely to be assigned to the same cluster.

4) Challenges: **Refinement Mechanism Design** The first challenge we encountered was the design of our refinement mechanism. In essence, a graph neural network works by performing message passing on transformed node features/embedding between nodes. This process only implicitly involves the graph structure, i.e., the graph topology, because features are passed along existing edges. It was a challenging task to inject structural information signals more explicitly to the network. We did extensive research and consulted experts including Professor Zheng Zhang and Dr. Xingjian Shi, and finally decided to use a min-cut based auxiliary loss to update/refine the assignment matrix. The next immediate question was how to compute the "gradient" of a loss function that has a non-differentiable term. After some research, we decided to use the proximal gradient descent algorithm.

Fig. 2. *DiffPool* hard assignment on *ENZYMES*

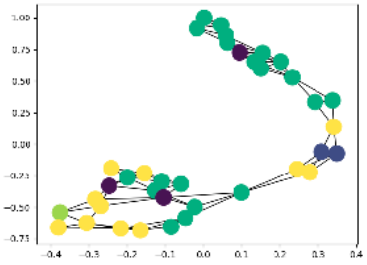


Fig. 3. *DiffPoolProx-2hop* hard assignment on *ENZYMES*

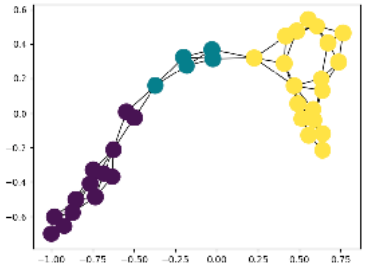
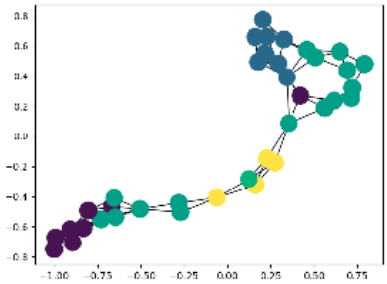


Fig. 4. *DiffPoolProx-global* hard assignment on *ENZYMES*



Synthetic Dataset Design As we argue in the previous sections, datasets used in [22] do not contain clearly defined substructures. Therefore, to better test the model's pooling capacity, it's necessary to design a synthetic dataset on which models that can extract graph substructures should in theory outperform models that cannot. To tackle this, we first came up with several synthetic dataset designs. Then we used some simple types of graphs such as ring graph or star graph to build synthetic graphs, and visualized them to see whether there is any clearly defined substructure. We then tweaked the synthetic dataset design, and finally reached the current design.

Fig. 5. *DiffPool* hard assignment on *ENZYMES-E*

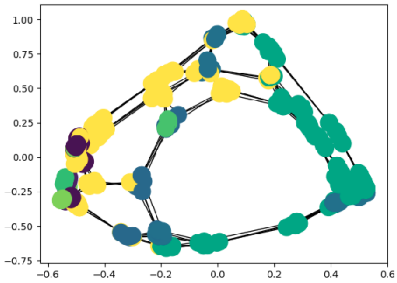


Fig. 6. *DiffPoolProx-2hop* hard assignment on *ENZYMES-E*

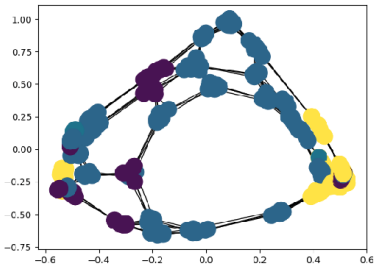
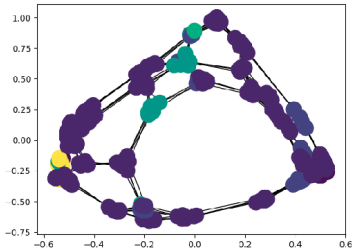


Fig. 7. *DiffPoolProx-global* hard assignment on *ENZYMES-E*



VI. Conclusion

We proposed a new learning-based graph pooling method that iteratively refines node assignments after each pooling GNN layer by optimizing a min-cut based auxiliary loss function with proximal gradient descent. In this way, we successfully incorporated both structural and feature information into the graph pooling process. To better test a model's graph pooling capacity, we also developed two synthetic datasets that contain clearly defined substructures in each graph example. Our model, *DiffPoolProx* achieved higher performance comparing to the baseline *DiffPool* model on the graph classification task, and can produce more consistent node assignments during the inference time.

Looking forward, there are several directions for future work:

- Improve the refinement auxillary loss function $r(S)$. We could use normalized cut to replace min-cut.

This could help to produce a more balanced node assignment result for a given graph. We could also replace the sparsity constraint $||S||_1$ with the $l_{2,1}$ norm on S to encourage structured sparsity on S instead of an overall sparsity constraint.

- Sparsify the network architecture. As both *DiffPoolProx* and *DiffPool*'s assignment matrices are still inherently dense, they require a $\mathcal{O}(V^2)$ storage complexity. This makes pooling large graphs a quite expensive task. If we can sparsify the assignment matrix during the training time, we can reduce the model's complexity and make it more deployable to various computing platforms.
- Develop a theoretical framework for graph pooling. Both our work and [22] are empirical, exploratory work on this emerging topic of graph pooling. To have a deeper understanding of graph pooling, it's necessary to develop a rigorous theoretical framework including a mathematical definition of graph pooling, as well as expected, clearly defined mathematical properties that a "good" graph pooling method should have.

VII. Acknowledgement

We thank Prof. Zheng Zhang, Prof. Xianbing Gu, Prof. Oliver Marin, Xingjian Shi, Jinjing Zhou, Ziyue Huang, and many other friends for their invaluable advice on this project. Code is available at https://github.com/HQ01/CS_capstone_2019.

References

[1] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[2] J. Shi and J. Malik, "Normalized cuts and image segmentation," *Departmental Papers (CIS)*, p. 107, 2000.

[3] D. Peleg and A. A. Schäffer, "Graph spanners," *Journal of graph theory*, vol. 13, no. 1, pp. 99–116, 1989.

[4] D. R. Karger, "Random sampling in cut, flow, and network design problems," *Mathematics of Operations Research*, vol. 24, no. 2, pp. 383–413, 1999.

[5] D. A. Spielman and S.-H. Teng, "Spectral sparsification of graphs," *SIAM Journal on Computing*, vol. 40, no. 4, pp. 981–1025, 2011.

[6] J. Batson, D. A. Spielman, N. Srivastava, and S.-H. Teng, "Spectral sparsification of graphs: theory and algorithms," *Communications of the ACM*, vol. 56, no. 8, pp. 87–94, 2013.

[7] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Advances in neural information processing systems*, 2016, pp. 3844–3852.

[8] 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.

[9] D. I. Shuman, M. J. Faraji, and P. Vandergheynst, "A multiscale pyramid transform for graph signals," *IEEE Transactions on Signal Processing*, vol. 64, no. 8, pp. 2119–2134, 2016.

[10] F. Dorfler and F. Bullo, "Kron reduction of graphs with applications to electrical networks," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 60, no. 1, pp. 150–163, 2013.

[11] A. Loukas, "Graph reduction by local variation," Available at https://arxiv.org/pdf/1808.10650.pdf.

[12] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," *arXiv preprint arXiv:1312.6203*, 2013.

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

[14] 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.

[15] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 1263–1272.

[16] 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*, 2015, pp. 2224–2232.

[17] 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*. Springer, 2018, pp. 593–607.

[18] W. Huang, T. Zhang, Y. Rong, and J. Huang, "Adaptive sampling towards fast graph representation learning," in *Advances in Neural Information Processing Systems*, 2018, pp. 4563–4572.

[19] J. Chen, J. Zhu, and L. Song, "Stochastic training of graph convolutional networks with variance reduction," *arXiv preprint arXiv:1710.10568*, 2017.

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

[21] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[22] 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*, 2018, pp. 4805–4815.

[23] C. Cangea, P. Veličković, N. Jovanović, T. Kipf, and P. Liò, "Towards sparse hierarchical graph classifiers," *arXiv preprint arXiv:1811.01287*, 2018.

[24] 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.

[25] I. Schomburg, A. Chang, C. Ebeling, M. Gremse, C. Heldt, G. Huhn, and D. Schomburg, "Brenda, the enzyme database: updates and major new developments," *Nucleic acids research*, vol. 32, no. suppl_1, pp. D431–D433, 2004.

[26] P. Yanardag and S. Vishwanathan, "Deep graph kernels," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 1365–1374.

[27] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.