Adversarial Caching Training: Unsupervised Inductive Network Representation Learning on Large-scale Graphs

Junyang Chen, Zhiguo Gong*, *Senior Member, IEEE*, Wei Wang, Cong Wang*, Zhenghua Xu, Jianming Lv, Xueliang Li, Kaishun Wu, Weiwen Liu

***Abstract*—Network representation learning (NRL) has far-reaching effects on data mining research, showing its importance in many real-world applications. NRL, also known as network embedding, aims at preserving graph structures in a low-dimension space. These learned representations can be used for subsequent machine learning tasks, such as vertex classification, link prediction, and data visualization. Recently, graph convolutional network (GCN) based models, e.g., GraphSAGE, have drawn a lot of attention for their success in inductive NRL. When conducting unsupervised learning on large-scale graphs, some of these models employ negative sampling (NS) for optimization, which encourages a target vertex to be close to its neighbors while being far from its negative samples. However, NS draws negative vertices through a random pattern or based on the degrees of vertices. Thus, the generated samples could be either highly relevant or completely unrelated to the target vertex. Moreover, as the training goes, the gradient of NS objective calculated with the inner product of the unrelated negative samples and the target vertex may become zero, which will lead to learning inferior representations. To address these problems, we propose an adversarial training method tailored for unsupervised inductive NRL on large networks. For efficiently keeping track of high-quality negative samples, we design a caching scheme with sampling and updating strategies that has a wide exploration of vertex proximity while considering training costs. Besides, the proposed method is adaptive to various existing GCN-based models without significantly complicating their optimization process. Extensive experiments show that our proposed method can achieve better performance compared with the state-of-the-art models.**

***Index Terms*—Graph Neural Network, Adversarial Learning, Network Embedding, Inductive Learning, Negative Sampling**

This work was supported in part by the National Key Development and Research (D&R) Program of China under Grant 2019YFB1600704, in part by The Science and Technology Development Fund, Macau SAR, under Grant FDCT/0068/2020/AGJ and FDCT/0045/2019/A1, in part by Key-Area Research and Development Program of Guangdong Province under Grant 2019B111103001, in part by National Natural Science Foundation of China under Grant (U2001207, 61872248, 61902249, 61876065), in part by the Guangdong "Pearl River Talent Recruitment Program" under Grant 2019ZT08X603. (Corresponding authors: Zhiguo Gong; Cong Wang.)

J. Chen, X. Li, and K. Wu are with the College of Computer Science and Software Engineering, Shenzhen University, China. Email: junyangchen@szu.edu.cn, lixueliangszu.edu.cn, wu@szu.edu.cn

Z. Gong is with State Key Laboratory of Internet of Things for Smart City, Department of Computer Information Science, University of Macau, China, fstzgg@um.edu.mo

W. Wang is with the School of Intelligent Systems Engineering, Sun Yat-sen University, China. Email: wangw328@mail.sysu.edu.cn

C. Wang is with the Department of Computing, The Hong Kong Polytechnic University, China. Email: supercong94@gmail.com

Z. Xu is with State Key Laboratory of Reliability and Intelligence of Electrical Equipment, Hebei University of Technology, China. Email: zhenghua.xu@hebut.edu.cn

J. Lv is the School of Computer Science and Engineering, South China University of Technology, China. Email: jmlv@scut.edu.cn

W. Liu is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, China. Email: wwliu@cse.cuhk.edu.hk

Manuscript received 05/06/2020

I. Introduction

Graph structures, e.g., citation networks and social networks, are ubiquitous and fast-growing in the real world. Network representation learning (NRL) can map the semantic similarity of graph vertices into a low-dimensional vector space where the similar vertices are assigned to the nearby areas [1]. The learned representations are useful for the subsequent applications, such as vertex classification [2], link prediction [3], and data visualization [4]. As demonstrated in the above applications, more discriminative representations of vertices would benefit for the better performance of the downstream tasks. Thus, the key to the success of the downstream applications is learning discriminative representations of vertices.

In general, current developments in NRL mostly fall into two categories: transductive learning and inductive learning. For example, DeepWalk [2], Line [5], Node2vec [3], and GCN [6] are transductive models which require that all vertices in networks are present during the training process of NRL. Though these models can perform well in the training data, they could not be generalized to unseen vertices. Notice that, among them, GCN obtains a lot of attention for its firstly proposing an efficient variant of convolutional neural networks that can operate directly on graphs.

Therefore, inductive GCN-based learning models such as GraphSAGE [7], GAT [8], and FastGCN [9] are recently proposed to generate vertex embeddings for unseen vertices. However, as stated by the authors, GAT is not suitable for large-scale networks since its intense computation of attention coefficients. Moreover, when applying these approaches to fully unsupervised NRL, they may suffer from a gradient vanishing problem during the optimization. Because in the unsupervised setting, negative sampling (NS) [10] is an important step in NRL, which encourages a target vertex to be close to its neighbors while being far from its negative samples. Nevertheless, NS draws negative vertices through a random mode or based on the degrees of vertices, then, the generated samples could be either highly relevant or completely unrelated to the target vertex (an example of using GraphSAGE with NS is shown in Fig. 1). In addition, as training goes, the gradient evaluated with the sampled unrelated negative vertices

Fig. 1: When using GraphSAGE in unsupervised NRL, we can see that: (a) For the target vertex 1, vertex 2 and 3 may be generated as the negative vertices by the original negative sampling method, whereas vertex pair (1, 2) is highly relevant and pair (1, 3) is totally unrelated; (b) Our proposed adversarial sampling method can generate more appropriate negative sample such as vertex 4 in the above example. Not that the details of the aggregators will be introduced in Section II-C.

may become zero, because these negative samples would be far away from the target vertex in the embedding space and the gradient calculated by activation functions (e.g., sigmoid function) could be a very small number. As a result, the NRL process will be stuck by the gradient vanishing problem which leads to inferior representation learning.

In essence, the major problem of NS is that it models negative samples with a xed scheme which ignores the dynamic changes of embedding features during the training. Recently, generative adversarial networks (GAN) [11] and its variants [12], [13] have shown promising ability to capture complex distributions, which is a potential replacement of the xed scheme in NS. However, the integration of GAN and NS is not seamless since keeping track of the dynamic negative sample distribution for each vertex is inef cient.

To balance ef ciency and effectiveness, in this paper, we design an adversarial caching scheme with sampling and updating strategies that has a wide exploration of vertex proximity while considering training costs. Our proposed method, called AdvCaching, is adaptive to the existing well-established GCN-based models. We implement our idea by building upon GraphSAGE for its popularity. The discriminator in AdvCaching is trained to optimize the objective functions of NRL as in the previous models. And the generator in it can be regarded as an auxiliary which learns high-quality negative samples and pushes the discriminator to its limit in representation learning. Speci cally, in initialization, we randomly select negative samples into a cache. Then, we use the combination of uniform sampling and probabilistic updating strategies to maintain this cache during the training. In general, our well-designed caching scheme can capture the dynamic changes of high-quality negative samples, while exploring as more the potentially negative ones as possible. The main contributions of this paper can be summarized as follows:

- We propose an adversarial training method, AdvCaching, which is tailored for unsupervised inductive NRL on large-scale graphs by building upon GraphSAGE. As a principle, the proposed method also can be applied to other GCN-based models.
- Speci cally, we employ a discriminator which contains the original neural network structure as GraphSAGE. And we leverage a generator to make an effect of structure distillation [14], which has fewer parameters compared with the discriminator for modeling negative sample distributions.
- To improve the tracking ef ciency of the dynamic negative sample distribution for each vertex, we employ the combination of uniform sampling and probabilistic updating strategies to maintain a caching scheme for the sample generation.
- We conduct extensive experiments on the subsequent application tasks to evaluate the quality of the representations learned by AdvCaching. Experimental results show that our proposed method can achieve signi cant and consistent improvements over state-of-the-art models. The code and datasets will be released at the revision stage.

The rest of this paper is organized as follows. In Section II, we rstly give preliminaries. In section III, we introduce the core idea of our proposed model and present the AdvCaching algorithm. We discuss experimental results in Section IV and show the related work in Section V. Section VI concludes our work.

II. Preliminaries

In this section, we rstly give the problem formulation and notations. Then, we will introduce the general idea of GraphSAGE which is selected as a base model for the proposed AdvCaching.

A. Problem formulation and notations

In network representation learning (NRL) tasks, we denote a network as $G = (V, E)$, where V is the set of vertices and E

V V denotes the set of edges. For each vertex $v \in V$, NRL aims to learn a low-dimensional embedding $\mathbf{v} \in \mathbb{R}^d$ which preserves the network proximity. Here d j V j represents the dimension of the representation space.

As mentioned before, classical NRL approaches based on transductive learning cannot generalize to unseen vertices. To perform inductive learning, models need to allow embeddings to be ef ciently generated for the unseen vertices. GraphSAGE is one of the state-of-the-art models for inductive NRL, which is introduced as follows.

B. The Objective of GraphSAGE in Unsupervised Inductive NRL

As stated in GraphSAGE [7], the authors employ negative sampling (NS) [10] to learn vertex representations in an unsupervised setting. The objective function of GraphSAGE is de ned as follows:

$$J(\mathbf{v}_i) = \log(\sigma(\mathbf{v}_p^T \mathbf{v}_i)) - \sum_{j=1}^{K} \mathbb{E}_{v_j \sim P_{NS}} \log(\sigma(-\mathbf{v}_j^T \mathbf{v}_i)), \quad (1)$$

where v_i is a target vertex, v_p is its neighbor vertex (there is an edge between them in datasets), σ is the sigmoid function, i.e., $\sigma(x) = 1/(1+\exp(-x))$, P_{NS} is a uniform negative sampling distribution involving all vertices, v_j is a negative sample drawn from P_{NS}, K is the number of negative samples for the estimation, and the representations, i.e., $\mathbf{v}_p$, $\mathbf{v}_i$, and $\mathbf{v}_j$, are aggregated from the features contained within their local neighbors. The details of the aggregation methods will be introduced in the following section. This objective aims to encourage nearby vertices to have similar embeddings while being distinct to their negative vertices.

C. Aggregation Methods in GraphSAGE

There are three aggregation methods in GraphSAGE including mean aggregator, LSTM aggregator, and pooling aggregator [7]. For an illustration of integrating the proposed AdvCaching method into GraphSAGE, we use the mean aggregator as an example, which is de ned as follows:

$$\mathbf{v}^l \leftarrow \sigma(\mathbf{W}^l \cdot \text{CONCAT}(\mathbf{v}^{l-1}, \text{MEAN}(\{\mathbf{v}_p^{l-1}, \forall v_p \in \mathcal{N}(v)\}))), \quad (2)$$

where MEAN denotes the element-wise mean of the vectors, CONCAT represents vector concatenation, $\mathbf{W}^l$ is the weight matrices of layer l, $\mathbf{v}^l$ is the embedding vector in layer l, σ is the sigmoid function, and $\mathcal{N}(v)$ denotes the neighbors of vertex v. From Eq. (2), we can see that GraphSAGE is able to aggregate the neighbor representations of unseen vertices for inductive learning.

III. Proposed AdvCaching Method

In this section, we will present the core idea of the proposed AdvCaching method by using GraphSAGE as the base model (an overview of the training framework is shown in Fig. 2), followed by detailed descriptions of its components.

A. AdvCaching GraphSAGE

As mentioned in the preliminaries, GraphSAGE adopts negative sampling (NS) in NRL but with xed distributions. Specifically, the original NS is based on a uniform distribution or on vertex degrees, thus the vertices with higher degrees are more likely to be drawn as the negative samples [10]. Therefore, NS cannot consider the dynamic changes of embeddings in the training process and may encounter the gradient vanishing problem. Recently, GAN's technique [11] has shown promising capability in monitoring complex distributions, which is a potential replacement of NS. In this paper, we leverage a generator that can bring high-quality negative samples to the discriminator for NRL. As shown in Figure 2, given the input edges of the network, we rstly estimate their negative vertex distributions with the generator. Then, we could sample high-quality negative vertices with the designed caching scheme. Finally, the discriminator performs representation learning. Here we employ GraphSAGE as the base model in the discriminator for illustration. The followings are the detailed descriptions of AdvCaching components.

B. Generator in AdvCaching GraphSAGE

To leverage embedding features for obtaining high-quality negative samples which can bring high gradient loss to the discriminator, we exploit a generator G with softmax function to model the negative candidate distribution, which is de ned as follows:

$$G(v_j | v_i; \theta_G) = \frac{\exp(\mathbf{v}_i \mathbf{v}_j^T)}{\sum_{v_j \in V} \exp(\mathbf{v}_i \mathbf{v}_j^T)}, \quad (3)$$

where v_i is the target vertex, v_j is the negative candidate, V denotes the whole set of vertices, and θ_G represents the union of all vertex embeddings in the generator.

Aggregation of Negative Vertex Embedding. After computing the above generator G, we can sample a negative vertex $v_j \sim G(v_i; \theta_G)$ as a substitute for $v_j \sim P_{NS}$ in Eq. (1), where P_{NS} denotes the original xed distribution of negative samples in NS and we use the generator G to replace P_{NS} for monitoring the dynamic changes of the negative sample distribution. Then, the aggregate embedding of each negative vertex v_j in GraphSAGE can be further represented as follows:

$$\mathbf{v}_j^l \leftarrow \text{Mean-Aggregate}^l(\mathbf{v}_j), \mathbf{v}_j \sim G(v_i; \theta_G), \quad (4)$$

where Mean-Aggregate$^l(\cdot)$ denotes the right part of Eq. (2). To sum up, from Eq. (4) and Eq. (3), we can see that the generator G can well consider the dynamic change of embedding features and provide a concise connectivity distribution for each target vertex. The softmax calculation is intuitive, however, the summation term inside Eq. (3) is computationally inef cient because it involves all vertices for each target vertex, especially for real-world large-scale graphs that may contain millions of vertices.

Generator Optimization. To address the aforementioned problems, we employ a cache scheme for the softmax function. Then, the calculation space of the summation term in

Fig. 2: An overview of the AdvCaching GraphSAGE framework. The generator aims to provide high-quality negative vertices by a well-designed cache scheme and the reward from the discriminator. And the discriminator learns the vertex representations based on the edge and the provided negative vertices.

Eq. (3) can be simplified to the caching space. Specifically, the generator optimization is formulated as follows:

$$G(v_j|v_i;\theta_G) = \frac{\exp(v_i \cdot v_j^T)}{\sum_{v_j \in C}\exp(v_i \cdot v_j^T)}, \quad (5)$$

where C is the cache of vertices, generally $|C| \ll |V|$ (The details of the cache design will be introduced in the Caching Scheme section), and v_j is the negative vertex in C. Thus, the summation term of Eq. (5) only takes the slight expense of computation costs, reducing from $|V|$ to $|C|$. Next, the loss function of the generator can be defined as follows:

$$L_G = \sum_{v_i \in B} E_{v_j \sim G(\cdot|v_i;\theta_G)} D(v_i, v_j; \theta_D), \quad (6)$$

where B denotes a batch in the training process, v_i represents the negative vertex, θ_D is the union of all vertex embeddings in the discriminator, and $D(\cdot)$ indicates the discriminator function (we define it as the sigmoid function, i.e., $D(v_i, v_j; \theta_D) = \sigma(v_i \cdot v_j^T) = 1/(1+\exp(-v_i \cdot v_j^T))$). Note that θ_D and θ_G denote the embedding vectors from the discriminator and generator, respectively, where they do not share the embeddings. In a nutshell, this formulated generator aims to sample high-quality negative vertices, i.e., v_j, from the cache C with the softmax probability distribution, which can prevent from generating totally unrelated vertices when using the uniform sampling. However, the sampled output of the generator is a discrete index of the cache. Therefore, the stochastic gradient descent (SGD) method can not be directly used for optimization. According to [15], [16], we can use a policy gradient-based reinforcement learning method to optimize the generator loss as follows:

$$\begin{aligned}\nabla_{\theta_G} L_G &= \nabla_{\theta_G} \sum_{v_i \in B} E_{v_j \sim G(\cdot|v_i;\theta_G)} D(v_i, v_j; \theta_D) \\ &= \sum_{v_i \in B} E_{v_j \sim G(\cdot|v_i;\theta_G)} D(v_i, v_j; \theta_D) \nabla_{\theta_G} \log G(v_j|v_i;\theta_G),\end{aligned} \quad (7)$$

where the gradient of L_G is an expected summation of $\nabla_{\theta_G} \log G(c_n|v_t;\theta_G)$ weighted by $D(\cdot)$ which is calculated with the discriminator. In the field of reinforcement learning, $D(\cdot)$ in Eq. (7) can be regarded as a reward function and the generator is trained to maximize the expected reward. In order to achieve a higher reward, for each negative pair (v_i, v_j), the policy used by the generator network would punish trivial negative vertices by lowering down their corresponding probability and encourage the discriminator network to distribute high-quality negative vertices, i.e., pair (v_i, v_j) with higher similarity from the discriminator parameterized by θ_D will be encouraged to be generated. Moreover, in practice, the reinforcement-based algorithms may suffer from unstable performance and receive high variance results [17]. According to [18], this problem can be alleviated by adding a baseline function to the reward term in the gradient loss. Then $D(\cdot)$ can be replaced by:

$$D(v_i, v_j; \theta_D) + \frac{\sum_{B \in P}\sum_{v_i \in B} E_{v_j \sim G(\cdot|v_i;\theta_G)} D(v_i, v_j; \theta_D)}{|P|}, \quad (8)$$

where P denotes the whole batches in an epoch, and the baseline function is the average reward of epochs obtained in the training process.

C. Discriminator in AdvCaching GraphSAGE

The discriminator of our proposed AdvCaching GraphSAGE aims to perform unsupervised inductive NRL with the high-quality negative samples constructed by the generator. The objective function of D is formulated as follows:

$$\begin{aligned}L_D = \sum_{v_i \in B} [&-\log D(v_i, v_p; \theta_D) \\ &- \sum_{j=1}^{K} E_{v_j \sim G(\cdot|v_i;\theta_G)} \log(D(-v_i, v_j; \theta_D)) \\ &+ (\|v_p\|_F^2 + \|v_i\|_F^2 + \|v_j\|_F^2)],\end{aligned} \quad (9)$$

where B denotes a batch in the training process, v_p denotes the adjacent vertex of v_i, v_j is the negative vertex sampled by using Eq. (5), $\|\cdot\|_F$ is Frobenius norm of vectors, $\{v_p, v_i, v_j\}$

are the embedding vectors in $_D$, and is a harmonic factor for regularization (we set it as 1e-5 in experiments). The discriminator D can be optimized with the gradient descent technique. An important distinction between this objective function and the original one as shown in Eq. (1) is that the negative vertex v_j is sampled from the cache with softmax function, instead of the uniform sampling. We can see that our proposed method is adaptive to the GraphSAGE model without signi cantly complicating its optimization process. Moreover, as a principle, our method also can be extended to other GCN-based models. We leave this discussion in Section III-F.

D. Minimax Form of the Final Loss

Without loss of generality, we provide the minimax form of our method as follows:

$$\min_{\theta} \max_{D} V(G;D) = \sum_{v_i \in B} \log D(v_i, v_p; \ _D) + \sum_{j=1}^{K} E_{v_j \sim \ _\theta(\cdot|v_i; \ _\theta)} \log(1 - D(v_i, v_j; \ _D)) \ ; \quad (10)$$

where G and D are playing a minimax game presented with value function $V(G;D)$. In general, generator G and discriminator D act as two opponents: (1) Generator G would try to generate high-quality negative samples that are similar to the target vertex v_i's real immediate neighbors to deceive discriminator D; (2) On the contrary, discriminator D would try to be far from these generated negative samples.

E. Caching Scheme

As mentioned before, though GAN's technique can monitor a complex generation process, it is inef cient to keep track of the dynamic negative sample distribution for each vertex. As shown in Eq. (5), we are motivated to cache high-quality negative samples with large probabilities. In this way, we can ef ciently track the dynamic changes of embedding features while exploring as more vertices as possible.

The overall training process of our proposed method with the caching scheme is shown in Algorithm 1. To begin with, we give the following notations.

Embedded cache C_1. We employ a cache $C_1 \in R^{|V| \times N_1}$ to store candidate negative samples, where $|V|$ denotes the number of vertices, N_1 is the cache size, and $N_1 \ll |V|$. Note that we represent C_1 as the embedded cache because it has stationary memory costs during the training.

Temporary cache C_2. We use a temporary cache $C_2 \in R^{|B| \times N_2}$ to store the re-sampling vertices, where $|B|$ is the batch size, N_2 represents the re-sample size, and $N_2 \ll |V|$. We can reuse the same temporary cache during the batch processing for space-saving.

In order to achieve the best performance, we need to well de ne sampling and updating strategies (step 9 and step 11 in Algorithm 1), which aims to have a wide exploration of vertex proximity while considering training costs. The details are as follows.

Uniformly Re-sampling Strategy (step 9). This strategy considers ef ciency and aims to explore as more potential negative vertices as possible. As the embedding features are changing in the training process, the negative vertices in the cache C_1 will be outdated and not accurate. Thus, we need to explore more possible vertices to update the cache. This motivates us to adopt a uniform sampler which is a simple and ef cient way to re-sample the candidate negative vertices of the network. The time complexity is O(1). Besides, it is worth mentioning that we also avoid selecting the direct neighbors of vertices when constructing their re-sampling cache C_2. Because it is unlikely that there is an edge between two vertices but they are not related to each other.

Probabilistic Updating Strategy (step 11). This strategy aims to dynamically update the embedded cache C_1 during the iterations. As the training goes, the gradient loss of Eq. (9) may become zero if we x the cache. To avoid the gradient vanishing problem and obtain high-quality negative samples, we draw vertices from the united cache C following the

---

Algorithm 1: Training Process of AdvCaching

Input: Graph $G = (V, E)$, batch size $|B|$, emebding dimension d, embedded cache C_1, temporary cache C_2

Result: Parameters of Discriminator $_D$ and Generator $_\theta$

```
1  begin
2      Initialize _D, _G, and the cache of negative samples C_1 randomly;
3      while not converge do
4          Sample a batch B from the edges of vertices;
5          Initialize the temporary cache C_2;
6          for G-steps do
7              for each edge (v_i, v_p) ∈ B do
8                  Index the cache C_1 to get the candidate negative samples C_1(v_i);
9                  Index the cache C_2 and uniformly re-sampling candidates from the whole set of vertices V into the cache C_2(v_i);
10                 Use _θ to sample quality negative vertex v_j from the cache set C = {C_1(v_i), C_2(v_i)} according to Eq. (5);
11                 Sample a new candidate negative cache from C with probabilities calculated by Eq. (5) to update C_1(v_i);
12             end
13             Update the parameters of _θ via policy gradient in Eq. (7);
14         end
15         for D-steps do
16             For each edge (v_i, v_p) in batch B and the sampled negative vertex v_j, calculate D loss using Eq. (9);
17             Update the parameters of _D via gradient descent;
18         end
19     end
20 end
```

probability calculated by Eq. (5), where $C = \{C_1(v_i), C_2(v_i)\}$ and a higher probability means more relevance of the vertex pair. Then, C_1 can be updated with the sampled vertices.

In summary, we exploit the above strategies to balance the efficiency and the effectiveness of modeling negative sample distributions. A larger size N_2 of the cache C_2 means exploring more potential negative vertices from the network. And the uniform sampler is employed for its efficiency. In addition, a bigger size N_1 of the cache C_1 implies keeping track of more negative vertices. And the cache with the probabilistic update is adopted for its effectiveness. Besides, since our method is an extension for other GCN-based models, the time complexity of our method is on par with them with addition computation $O(|V|(N_1 + N_2)d)$, where d is the embedding dimension.

Till now, we have introduced the training process of AdvCaching and its components as shown in Figure 2. Moreover, as a principle, our method also can be extended to other GCN-based models without significantly complicating their optimization processes. The details will be discussed in the next section.

F. Models Extended by AdvCaching

In this part, we first elaborate on the principle of building AdvCaching on GraphSAGE. Then, we show how to apply AdvCaching to extend other GCN-based models.

As mentioned before, all of the GCN-based models including GraphSAGE utilize negative sampling (NS) [10] for the optimization in NRL. The major idea of Advcaching is to replace NS with a defined cache scheme. Specifically, the proposed generator can embed this cache seamlessly while the discriminator completes NRL tasks with the base models. Moreover, this generator has fewer parameters compared with the discriminator. For example, GraphSAGE contains multiple layers and has complicated forward propagation algorithms, i.e., mean aggregator, LSTM aggregator, and pooling aggregator [7]. We can generalize GraphSAGE to a discriminator which contains the original complex neural network structure. In the meantime, we use a generator to model negative vertex distributions with only one layer for structure distillation [14].

Note that the main difference between GraphSAGE and other GCN-based models is the aggregator definition. For instance, GCN [6] adopts a localized spectral convolutional aggregator which is defined as follows:

$$v^l \leftarrow \sigma(W^l \cdot \text{MEAN}(\{v^{l-1}\} \cup \{v_p^{l-1}, \forall v_p \in \mathcal{N}(v)\})), \quad (11)$$

where the variable meanings are the same as Eq. (2). We denote the right part of the above equation as GCN-Aggregate$^l(\cdot)$. Then, the aggregate embedding of each negative candidate vertex v_j in GCN can be further represented as follows:

$$v_j^l \leftarrow \text{GCN-Aggregate}^l(v_j), v_j \in \mathcal{G}_b(v_i; \theta_b), \quad (12)$$

where $v_j \in \mathcal{G}_b(v_i; \theta_b)$ represents that we use the proposed generator $\mathcal{G}_b$ to sample negative vertices instead of the uniform sampling adopted in the original GCN. Besides, follow the same principle, more base models such as GAT [8] also can be easily extended with the above generator.

TABLE I: Statistics of datasets

Datasets	# of vertices $\lvert V \rvert$	# of edges $\lvert E \rvert$	# of labels $\lvert L \rvert$
Cora	2708	5278	7
Citeseer	3264	4551	6
Wiki	2363	11596	17
DBLP_C4	17725	52914	4

IV. Experiments

In the experiments, we evaluate the performance of our proposed method in terms of vertex classification tasks on four real-life networks. Moreover, we also investigate the parameter influence on these tasks.

A. Datasets

We conduct experiments on four widely used network datasets with the statistics listed in Table I, where L denotes the label set.

Cora[1] is a research citation network constructed by [19]. It contains 2708 machine learning papers with 7 labels.

Citeseer[2] is another extensively adopted research paper set which contains 3264 publications and 6 labels.

Wiki[3] is a language network that contains 2363 web pages and 17 labels after preprocessing by deleting self-loops and nodes with zero degrees.

DBLP_C4[4] consists of bibliography data in computer science constructed by [20]. In the experiments, we select a list of conference papers from 4 research fields: database, data mining, AI, and CV.

To comprehensively evaluate the performance of models, we conduct experiments on these datasets with or without their nodes attributes. Specifically, on the Cora and Citeseer datasets, we perform representation learning on the graph adjacency with node features. For the Wiki and DBLP datasets, we are only focused on connectivity patterns in networks. One reason is that, in real-life scenarios, graphs without node attributes are more common and easy to acquire. We also want to evaluate the performance of GCN-based models on pure graph adjacency.

B. Baseline Models

We employ several state-of-the-art GCN-based models as the baselines, including GCN, GraphSAGE, and GAT. There are many other NRL methods but we do not consider them here, because either their performances are inferior to these baselines as shown in corresponding papers or they are transductive models that are inappropriate for inductive NRL. The descriptions of the baseline models are as follows:

GCN [6] first introduces an effective variant of convolutional neural networks that can operate directly on graphs. For inductive learning on large-scale networks, an improved version of the GCN approach [7] is derived. We employ this variant for comparison.

[1] https://people.cs.umass.edu/~mccallum/data.html
[2] https://github.com/wonniu/AdvT4NE_WWW2019
[3] https://github.com/albertyang33/TADW
[4] http://arnetminer.org/citation (V4 version is used)

GraphSAGE [7] proposes an inductive manner for computing vertex representations, which has yielded impressive performance on several large-scale benchmarks. For each vertex, GraphSAGE firstly samples fixed-size neighbors and then performs an aggregate function over them to obtain the vertex representations. There are four types of aggregate functions including GraphSAGE-mean, GraphSAGE-LSTM, GraphSAGE-maxpool, and GraphSAGE-meanpool (a variant of the maxpool aggregator, where the element-wise mean-pooling replaces the element-wise max-pooling). One main difference between "meanpool" and "mean" is that "meanpool" needs each neighbor's vector to be independently fed trough a fully-connected neural network.

GAT [8] leverages self-attentional layers to learn the importance weights of centered nodes' neighbors.

As mentioned before, we implement the proposed AdvCaching method by using GraphSAGE as the base model. We denote the improved variants according to the aggregate functions as follows: AdvCaching-GCN, AdvCaching-mean, AdvCaching-LSTM, AdvCaching-maxpool, AdvCaching-meanpool, and AdvCaching-GAT.

C. Parameter Settings and Evaluation Metrics

We follow the experiment setup in [7] to demonstrate the effectiveness of our proposed method. Specifically, for the discriminator of AdvCaching, we follow GraphSAGE [7] by setting the number of network layers $l_s = 2$, the hidden dimension $d = 128$, and the neighborhood sample sizes of layers $S_1 = 25$ and $S_2 = 10$, respectively. For the generator of AdvCaching, we set $l = 1$, $d = 128$, and use Adam optimizer [21] with the initial learning rate 1e-3. For the cache size, we uniformly set $N_1 = N_2 = 10$. Besides, we adopt vertex classification as the benchmark task for evaluating the learned representations: using Liblinear package [22] with default settings to build the classifier and employing classification accuracy [4] as the metrics.

D. Vertex Classification

In this section, we conduct downstream multi-class classification tasks on four benchmark datasets, including Cora, Citeseer, Wiki and DBLP_C4, with the training ratios ranging from 10% to 90%. We build the classifier using Liblinear package [22] with its default setting. From Table II, III, IV, and V, we have the following observations.

The proposed AdvCaching method, built upon GraphSAGE with adversarial learning components, consistently outperforms both GCN and the variants of GraphSAGE on four datasets across all training ratios, only with some exceptions in Wiki when the ratios are 10% and 20%. More specifically, the variants of AdvCaching achieve 5.9%, 9.6%, 7.2%, and 6.5% performance gains over the variants of GraphSAGE on average of ratios in Cora, Citeseer, Wiki and DBLP_C4 respectively, while the accuracy improvements of AdvCaching-GCN over GCN are 3.2%, 4.2%, 7.4%, and 5.2% respectively. In general, the experimental results validate that AdvCaching can obtain benefits from the high-quality negative samples generated by the proposed adversarial training framework. And this also confirms that our designed caching scheme can keep track well of these negative sample generations. Combined with Section III-F, we can summarize that the proposed AdvCaching method can produce positive results on GCN-based models in two aspects: making better performance contributions and less complicating the optimization process of the original models.

E. Convergence Analysis

In this part, we perform the convergence analysis of models. Fig. 3 shows the influence of epochs on the loss of algorithms during the pre-training. We report the performance of GAT, GCN, GraphSAGE-mean, and our proposed models on Cora, Citeseer, Wiki, and DBLP_C4 respectively. From Fig. 3, we can see that most of the models obtain dismissing loss after 25 epochs and all models achieve convergence at or before the 40th epoch. Note that we use the converged node embeddings in the above evaluation of vertex classification. Besides, the values of the pre-training loss would not be directly related to the downstream classification performance, because the embeddings of models may be over-smooth and thus can obtain smaller pre-training loss.

F. Execution Time and Classification Performance Analysis

To perform an ablation study, we make a comparison of methods in terms of execution time and classification accuracy on Cora, Citeseer, Wiki, and DBLP, respectively. Specifically, we compare GAT, GCN, and GraphSAGE-mean with our proposed methods with and without caching scheme, as shown in Fig. 4, 5, 6, and 7. Generally, the average execution time of our methods is 2.18, 2.37, 2.82, and 2.79 times comparing with the original methods on four datasets, respectively. In contrast, the proposed methods w/o caching scheme are 101, 131, 143, and 150 times over the original ones, which demonstrates the effectiveness of the caching design that can obtain competitive execution time compared with the original networks. Moreover, we can observe the accuracy performance of our methods can achieve 4.81%, 2.76%, 6.42%, and 4.16% improvements over the original methods, respectively, while the methods without caching scheme drop by 2.81%, 3.07%, 2.74%, and 6.12% of accuracy comparing with the original ones. We conjecture that the reason of performance degradation for methods without caching is that computing all vertices for generating negative samples at each step is time-consuming and more likely to fall into the local optimum. In general, our method makes improvements over previous methods coming at an acceptable computational cost.

G. Parameter Sensitivity

Fig. 8 shows the influence of the cache size on the accuracy and runtime performances of our proposed method. Here we adopt AdvCaching-mean on the Citeseer dataset for the evaluation of the parameter influence. First, as shown in Fig. 8a, we let the embedded cache size N_1 equal to the temporary cache size N_2. From this figure, we can observe that the runtime of the proposed method is approximately

TABLE II: Accuracy (%) of vertex classification on Cora

% Label Nodes	10%	20%	30%	40%	50%	60%	70%	80%	90%
GAT	63.00	68.57	70.31	70.03	71.42	71.49	71.46	71.40	73.43
AdvCaching-GAT	67.43	73.60	76.42	75.69	77.18	78.14	79.34	78.69	78.97
GCN	72.44	76.37	77.00	77.54	78.29	79.24	80.07	78.41	77.86
AdvCaching-GCN	75.64	77.99	78.01	78.09	78.73	79.15	80.44	78.78	79.34
GraphSAGE-mean	42.53	48.78	52.11	52.12	55.02	56.18	56.95	57.56	54.98
AdvCaching-mean	45.82	50.76	53.11	55.32	57.53	59.41	60.76	60.15	56.83
GraphSAGE-LSTM	68.17	73.56	74.95	74.52	75.63	76.94	78.97	78.41	76.75
AdvCaching-LSTM	71.21	76.42	77.43	77.05	78.06	78.69	79.95	79.15	79.70
GraphSAGE-maxpool	65.30	69.87	71.52	71.32	71.71	73.43	73.68	72.69	69.37
AdvCaching-maxpool	67.39	71.34	73.31	72.62	75.04	74.82	75.40	75.65	69.74
GraphSAGE-meanpool	68.46	71.34	73.05	72.68	74.52	75.83	77.49	77.31	74.17
AdvCaching-meanpool	75.80	78.50	79.32	79.08	80.28	81.37	82.66	82.29	81.92

TABLE III: Accuracy (%) of vertex classification on Citeseer

% Label Nodes	10%	20%	30%	40%	50%	60%	70%	80%	90%
GAT	61.82	63.02	65.11	65.85	66.49	67.47	66.60	64.71	63.86
AdvCaching-GAT	62.16	63.81	64.90	66.70	67.27	66.87	66.90	66.82	66.87
GCN	63.50	65.25	66.32	67.30	67.21	67.77	68.71	69.23	68.07
AdvCaching-GCN	65.65	67.51	68.69	69.52	70.23	70.79	71.53	70.44	68.67
GraphSAGE-mean	42.17	47.47	51.70	53.52	55.13	54.87	56.54	54.75	53.31
AdvCaching-mean	45.05	49.13	51.92	54.83	55.80	55.85	58.45	58.97	58.13
GraphSAGE-LSTM	64.84	65.43	66.71	67.35	67.93	67.70	68.71	68.48	68.37
AdvCaching-LSTM	65.95	67.06	68.35	69.16	69.63	69.43	70.42	69.68	69.88
GraphSAGE-maxpool	62.86	64.49	65.20	65.69	66.91	67.09	67.91	67.57	67.17
AdvCaching-maxpool	64.51	66.26	66.93	67.35	68.96	69.21	68.91	68.17	69.78
GraphSAGE-meanpool	62.70	65.43	66.93	66.75	66.67	66.79	67.51	67.57	67.77
AdvCaching-meanpool	63.17	66.79	67.53	67.96	68.18	68.98	70.22	69.83	68.37

TABLE IV: Accuracy (%) of vertex classification on Wiki

% Label Nodes	10%	20%	30%	40%	50%	60%	70%	80%	90%
GAT	49.46	52.35	53.23	54.65	54.74	55.92	56.14	53.91	51.90
AdvCaching-GAT	50.54	53.73	55.23	56.21	56.68	57.61	58.25	56.03	56.12
GCN	35.06	40.23	41.92	41.44	42.56	43.35	44.18	42.62	37.34
AdvCaching-GCN	35.47	41.84	43.17	43.80	45.64	46.88	46.26	46.57	46.47
GraphSAGE-mean	27.86	31.39	32.30	32.64	34.33	34.20	34.35	34.10	28.22
AdvCaching-mean	28.36	33.21	35.33	35.62	36.24	36.49	38.37	37.01	36.51
GraphSAGE-LSTM	42.08	46.05	47.03	46.99	49.38	49.58	48.48	45.95	44.39
AdvCaching-LSTM	39.68	46.31	48.28	48.99	51.70	52.81	53.60	53.85	50.62
GraphSAGE-maxpool	25.81	29.09	31.12	31.45	31.90	31.92	33.15	31.08	32.49
AdvCaching-maxpool	27.32	28.93	31.48	32.37	32.66	35.73	35.12	31.92	36.29
GraphSAGE-meanpool	42.17	44.43	45.37	45.88	47.05	46.78	47.78	45.74	43.98
AdvCaching-meanpool	42.03	48.49	49.29	49.27	49.88	51.66	52.91	51.35	47.30

linear to the cache size. Meanwhile, the accuracy performance fluctuates along with the increase of the cache size and reaches the bottom when $N_1 = N_2 = 30$. Then, we vary the cache size N_1 or N_2 given $N_2 = 10$ or $N_1 = 10$ to further evaluate the accuracy performance of cache size, as shown in Fig. 8b. The performance lines are normally stable. In general, our model is generally robust to the cache size settings. Moreover, we try to evaluate how continuously changing N1 or N2 will influence the final accuracy. As shown in Fig. 8c, we take both N_1 and N_2 as the axes by varying their numbers in $\{10, 15, 20, 25, 30\}$. We can observe that the proposed method achieves the best classification performance when $N_1 = 20$

TABLE V: Accuracy (%) of vertex classification on DBLP_C4

% Label Nodes	10%	20%	30%	40%	50%	60%	70%	80%	90%
GAT	72.37	73.18	73.79	73.72	73.81	73.54	73.69	73.29	73.15
AdvCaching-GAT	72.65	73.42	73.94	74.29	74.38	74.51	74.76	74.19	74.56
GCN	64.04	65.07	65.80	65.94	66.25	66.32	66.90	66.40	65.09
AdvCaching-GCN	67.81	68.60	69.12	69.39	69.49	69.15	69.50	69.59	69.66
GraphSAGE-mean	59.56	60.28	60.71	61.04	61.16	60.82	61.72	61.75	62.44
AdvCaching-mean	64.48	64.62	65.10	65.34	65.35	65.37	65.64	65.84	65.99
GraphSAGE-LSTM	63.62	64.55	65.13	65.49	65.47	65.20	65.55	64.82	64.02
AdvCaching-LSTM	66.24	66.97	67.89	68.23	68.35	68.18	68.47	68.58	68.98
GraphSAGE-maxpool	64.16	64.24	64.69	65.00	64.99	64.72	65.25	65.08	64.69
AdvCaching-maxpool	66.41	66.61	67.02	67.55	67.48	67.56	67.83	67.87	68.13
GraphSAGE-meanpool	62.38	62.79	63.14	63.68	63.79	63.61	63.86	63.92	63.23
AdvCaching-meanpool	68.17	69.13	69.59	69.92	70.19	70.23	70.61	70.13	70.16

(a) Cora (b) Citeseer (c) Wiki (d) DBLP_C4

Fig. 3: Convergence analysis. We report the performance of GAT, GCN, GraphSAGE-mean, and our proposed models on Cora, Citeseer, Wiki, and DBLP_C4, respectively. In general, all models achieve convergence before 40 epochs.

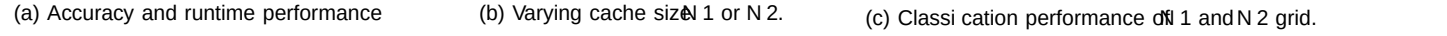
Fig. 4: Ablation study of methods in terms of execution time and classification accuracy on Cora dataset.

Fig. 5: Ablation study of methods in terms of execution time and classification accuracy on Citeseer dataset.

and $N_2 = 15$.

Fig. 6: Ablation study of methods in terms of execution time and classi cation accuracy on Wiki dataset.

Fig. 7: Ablation study of methods in terms of execution time and classi cation accuracy on DBLP dataset.



(a) Accuracy and runtime performance (b) Varying cache size N_1 or N_2. (c) Classi cation performance of N_1 and N_2 grid.

Fig. 8: Here we use AdvCaching-mean on the Citeseer dataset for the evaluation of parameter in uence. (a) Accuracy and runtime performance with respect to the cache size. Here we set $N_1 = N_2$ and report its average classi cation performance with training ratios from 10% to 90%; (b) We vary N_1 or N_2 given $N_2 = 10$ or $N_1 = 10$ to evaluate the classi cation accuracy in terms of cache sizes; (3) We take both N_1 and N_2 as axes by continuously varying their numbers from $\{5, 10, 15, 20, 25, 30\}$ to evaluate how they in uence the nal classi cation accuracy.

V. Related Work

Since we propose an adversarial caching training for incorporating GCN-based models into network representation learning, our work is related to the following three aspects:

Network Representation Learning (NRL), i.e., network embedding, learns to represent graph vertices in low-dimensional vectors. Following the success of word embedding [10], DeepWalk [2] is proposed to perform representation learning by applying SkipGram model [10] on the generated random walks. Subsequent improved algorithms such as LINE [5], Node2vec [3], and PolyDeepwalk [23] also achieve breakthroughs. However, these methods suffer from computational inef ciency because no parameters are shared between nodes in the encoder [24]. Besides, they only focus on learning representation from the local connectivity of vertices (i.e., neighbors) but ignore the vertex information propagation guided by the graph structure.

Graph Neural Networks (GNNs) motivated by CNNs [25] are proposed to collectively aggregate information from graph structures. In recent years, models such as graph convolutional network (GCN) [6] and its variant graph attention network (GAT) [8], have shown ground-breaking performance on many tasks including NRL. Compared with the previous models [2], [5], [3] mentioned before, GCN can jointly consider the local connectivity and global consistency on graphs. Based on the fundamental theory of GCN, more variants are proposed including GraphSAGE [7], FastGCN [9], DGCN [26], and HAN [27]. Among them, PinSage [28] incorporates Graph-

SAGE and achieve the largest application of deep graph embeddings, which paves the way for a new generation of web-scale recommender systems. Our proposed method can be applied to GraphSAGE and achieve better performances in NRL as shown in the experiments, which demonstrate that our work is worth investigating. Note that there are many other fancy graph-based models such as [29], [30] but we do not consider them here, because these models do not adopt negative sampling for unsupervised representation learning.

Generative Adversarial Network (GAN) [11] recently draws a lot of attention for its promising performances in various applications [31]. For example, KBGAN [12] and IGAN [13] propose to incorporate GAN for negative sampling in knowledge graph learning. Then, NSCaching [32] further proposes an ef cient method to improve its sampling way. Our work is inspired by these models but with notable differences. The essential distinction is that the assumption of NRL, that two connected vertices should be similar and close in embedding space, does not hold in the knowledge graph. In general, to the best of our knowledge, there is no practice of incorporating adversarial training of modeling negative samples into GCN for network representation learning.

VI. Conclusion

In this paper, we propose an adversarial training method called AdvCaching, for unsupervised inductive NRL on large-scale networks. Though GCN and its variants show effective performance in NRL, they may suffer from a gradient vanishing problem when adopting the negative sampling method for optimization. Instead of generating negative vertices from a uniform sampler, we want to keep track of the dynamic negative sample distributions by leveraging GAN. Speci cally, in AdvCaching, we adopt a discriminator that contains original complex neural networks of the base models, and use a generator that has fewer parameters compared with the discriminator to model negative sample distributions. To balance ef ciency and effectiveness, we further design an adversarial caching scheme with sampling and updating strategies that has a wide exploration of vertex connectivity while considering training costs. Besides, since GraphSAGE achieves great success in practical applications, we present the core idea of our work by using it as the base model. As a principle, AdvCaching also can be applied to extend other GCN-based models. Experiments on real-world datasets demonstrate that AdvCaching can achieve signi cant improvements over the state-of-the-art models, which validates the effectiveness of our proposed method.

References

[1] P. Cui, X. Wang, J. Pei, and W. Zhu, "A survey on network embedding," IEEE Transactions on Knowledge and Data Engineering, vol. 31, no. 5, pp. 833–852, 2018.

[2] 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. ACM, 2014, pp. 701–710.

[3] 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. ACM, 2016, pp. 855–864.

[4] Q. Dai, Q. Li, J. Tang, and D. Wang, "Adversarial network embedding," in Thirty-Second AAAI Conference on Arti cial Intelligence, 2018.

[5] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in Proceedings of the 24th international conference on world wide web. International World Wide Web Conferences Steering Committee, 2015, pp. 1067–1077.

[6] T. N. Kipf and M. Welling, "Semi-supervised classi cation with graph convolutional networks," arXiv preprint arXiv:1609.02907, 2016.

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

[8] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," arXiv preprint arXiv:1710.10903, 2017.

[9] J. Chen, T. Ma, and C. Xiao, "Fastgcn: fast learning with graph convolutional networks via importance sampling," arXiv preprint arXiv:1801.10247, 2018.

[10] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in Advances in neural information processing systems, 2013, pp. 3111–3119.

[11] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in Advances in neural information processing systems, 2014, pp. 2672–2680.

[12] L. Cai and W. Y. Wang, "Kbgan: Adversarial learning for knowledge graph embeddings," in Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers), 2018, pp. 1470–1480.

[13] P. Wang, S. Li, and R. Pan, "Incorporating gan for negative sampling in knowledge representation learning," in Thirty-Second AAAI Conference on Arti cial Intelligence, 2018.

[14] X. Wang, R. Zhang, Y. Sun, and J. Qi, "Kdgan: knowledge distillation with generative adversarial networks," in Advances in Neural Information Processing Systems, 2018, pp. 775–786.

[15] J. Schulman, N. Heess, T. Weber, and P. Abbeel, "Gradient estimation using stochastic computation graphs," in Advances in Neural Information Processing Systems, 2015, pp. 3528–3536.

[16] L. Yu, W. Zhang, J. Wang, and Y. Yu, "Seqgan: Sequence generative adversarial nets with policy gradient," in Thirty-First AAAI Conference on Arti cial Intelligence, 2017.

[17] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," Machine learning, vol. 8, no. 3-4, pp. 229–256, 1992.

[18] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in Advances in neural information processing systems, 2000, pp. 1057–1063.

[19] A. K. McCallum, K. Nigam, J. Rennie, and K. Seymore, "Automating the construction of internet portals with machine learning," Information Retrieval, vol. 3, no. 2, pp. 127–163, 2000.

[20] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su, "Arnetminer: extraction and mining of academic social networks," in Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2008, pp. 990–998.

[21] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv preprint arXiv:1412.6980, 2014.

[22] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "Liblinear: A library for large linear classi cation," Journal of machine learning research, vol. 9, no. Aug, pp. 1871–1874, 2008.

[23] N. Liu, Q. Tan, Y. Li, H. Yang, J. Zhou, and X. Hu, "Is a single vector enough? exploring node polysemy for network embedding," arXiv preprint arXiv:1905.10668, 2019.

[24] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," arXiv preprint arXiv:1812.08434, 2018.

[25] 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.

[26] C. Zhuang and Q. Ma, "Dual graph convolutional networks for graph-based semi-supervised classi cation," in Proceedings of the 2018 World Wide Web Conference, 2018, pp. 499–508.

[27] X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, and P. S. Yu, "Heterogeneous graph attention network," in The World Wide Web Conference, 2019, pp. 2022–2032.