Distributed Optimization of Graph Convolutional Network Using Subgraph Variance

Taige Zhao, Xiangyu Song, Man Li, Jianxin Li, Senior Member, IEEE, Wei Luo, and Imran Razzak, Senior Member, IEEE

Abstract—In recent years, distributed graph convolutional networks (GCNs) training frameworks have achieved great success in learning the representation of graph-structured data with large sizes. However, existing distributed GCN training frameworks require enormous communication costs since a multitude of dependent graph data need to be transmitted from other processors. To address this issue, we propose a graph augmentation-based distributed GCN framework (GAD). In particular, GAD has two main components: GAD-Partition and GAD-Optimizer. We first propose an augmentation-based graph partition (GAD-Partition) that can divide the input graph into augmented subgraphs to reduce communication by selecting and storing as few significant vertices of other processors as possible. To further speed up distributed GCN training and improve the quality of the training result, we design a subgraph variance-based importance calculation formula and propose a novel weighted global consensus method, collectively referred to as GAD-Optimizer. This optimizer adaptively adjusts the importance of subgraphs to reduce the effect of extra variance introduced by GAD-Partition on distributed GCN training. Extensive experiments on four large-scale real-world datasets demonstrate that our framework significantly reduces the communication overhead (≈50%), improves the convergence speed (≈2×) of distributed GCN training, and obtains a slight gain in accuracy (≈0.45%) based on minimal redundancy compared to the state-of-the-art methods.

Index Terms—Communicate reduction, distribution optimize, graph augmentation, graph convolutional network (GCN), subgraph variance.

I. INTRODUCTION

Graph convolutional networks (GCNs) have been widely used with remarkable success in recent years to learn the representations of graph-structured data [1], [2], [3]. Its power to capture the dependencies of a graph through message passing between vertices enables it to obtain vector representations of vertices, edges, or graphs with structure information, further improving the prediction accuracy in many downstream tasks such as vertex classification [4], [5], [6], link prediction [7], [8], and graph classification [9], [10]. This breakthrough led us to apply GCNs to complex scenarios, which may contain millions of vertices and billions of edges. For example, Microsoft Academic Graph [11] contains approximately 111 million vertices and one billion edges, and the knowledge graph of Freebase [12] contains nearly two billion vertices and almost 30 billion edges. This comes with a huge demand for time and memory space for GCN training.

Recently, many attempts have been made to alleviate memory and run-time in GCN training through distributed implementations. However, it brings several new challenges, such as how to rationally allocate the processor memory, balance the load, and optimize the efficiency of communication between processors [13], [14]. Preliminary research including PinSage [15], AliGraph [16] and deep graph library (DGL) [17] targets these core issues. AliGraph separates the traditional underlying architecture into storage, sampling, and operation layers and then applies a caching mechanism to store intermediate results between these layers. PinSage uses the local convolution separation method to dynamically calculate the final convolution result for the whole graph. DGL integrates the information dissemination mechanism into the message and reduction function to transfer and aggregate information. It enhances the scalability of the framework without adding additional overhead.

However, most of these approaches still have problems with the efficiency of distributed GCN training due to the high communication cost. To mitigate the impact of communication, Jiang and Rumi [18] and Scardapane et al. [19] emphasized the reduction of communication between processors by designing the sampling-based optimization method. Specifically, Jiang and Rumi [18] reduced communication overhead by assigning a lower sampling probability to vertices stored in other processors and a higher sampling probability to the vertices stored in the current processor when training the GCN models. Similarly, Scardapane et al. [19] considered the connection between processors. It used an additional concatenation subgraph to record the shortest communication path between processors. In this case, the vertices with the shortest communication path should be sampled first when training GCN in a distributed environment. Even though these methods showed a significant reduction in communication overhead, the communication overhead can not be negligible when training big graphs with a large number of processors because the communication overhead increases linearly with the size of the graph and the number of processors.
To address the aforementioned challenges, in this work, we propose a novel graph augmentation-based Distributed GCN framework (GAD) with two main components, named GAD-Partition and GAD-Optimizer. In the graph augmentation-based partition (GAD-Partition) phase, we divide the input graph into subgraphs with the addition of augmented information. Specifically, we use Metis [20] to split the input graph into a set of subgraphs. Then we devise a Monte-Carlo-based vertices importance measurement and an importance-first sampling method to augment the significant vertices in each subgraph. In addition, we develop a subgraph variance-based optimizer (GAD-Optimizer) to improve the efficiency and accuracy of distributed GCN training. Since subgraph variance is a good metric to measure the degree of vertices and the difference between vertices within a subgraph, i.e., the lower the variance of a subgraph, the closer the structural feature of its vertices in the subgraph could be. Therefore, it is faster to engage the convergence of GCN training propagation and the higher quality of the training result using a group of vertices with a lower subgraph variance than by using those with a higher subgraph variance [21]. Motivated by this property, we extend the subgraph variance measurement formula from sampling-based subgraphs to subgraphs generated by partition methods. In addition, we provide a weighted global consensus mechanism to unify the GCN parameters in each processor, by considering the effect of subgraph variance on the gradient.

The contributions of this work are summarized as follows.

1) An adaptive graph partition method, namely GAD-partition, is developed to partition an input graph into augmented subgraphs. It can reduce the communication costs of processors by adaptively selecting and storing as few significant vertices as possible.

2) We design a variance-based subgraph importance measurement formula and devise a novel weighted global consensus strategy, collectively referred to as GAD-Optimizer. It can adaptively adjust the weight of each subgraph based on the corresponding importance to accelerate the convergence of the training.

3) A unified framework, called GAD, is proposed to integrate the components of GAD-Partition, data loading, GCN training, and GAD-Optimizer into a distributed framework, which obtains the optimal accuracy performance and training speed in downstream tasks.

The remainder of this article is organized as follows. Section II reviews the related work. Section III describes the detailed procedure of GAD-Partition and GAD-Optimizer. Finally, we report the experimental setting and the evaluation results in Section IV and conclude our work in Section V.

II. RELATED WORK

There is a substantial body of research related to the minimization of overhead in computation and communication across multiple processors in distributed GCN training. Below, we first survey the research of distributed graph representation learning. Then, we discuss the in-depth optimized distributed graph representation learning.

A. Distributed Graph Representation Learning

The aim of graph representation learning is to obtain a low-dimensional continuous representation from discrete graphs and preserve the properties of graphs (e.g., vertex similarity) in embedding space [22], [23]. To accelerate the training process and deal with large-scale graph data, several distributed GNN training frameworks such as PyTorch Geometric [24], DGL [17], and Ant graph machine learning system (AGL) [25] were recently developed. They performed graph training on multiple GPUs or distributed machines. Bo et al. [26] presented a consensus-based distributed neural network to allow all the models in processors converging in a similar way. Based on this work, Scardapani et al. [19] presented a fully distributed GCN. It introduced a global consensus algorithm to unify the GCN model parameter by the average gradient. Yoo et al. [27] optimized the consensus-based distributed neural network and proposed an approximation consensus algorithm. It makes the parameter of GCN model being consistent and can guarantee the consensus errors in the suitable range. However, these methods still incur high frequent communication costs among the processors because boundary vertices need to access and aggregate the information of their neighboring vertices from other processors during the model training process.

B. Optimized Distributed Graph Representation Learning

To reduce communication between processors in a distributed environment, recent research focuses on the development of optimally distributed representation learning. In general, there are three classes of methods to optimize distributed graph representation learning.

The first class of methods uses the caching mechanism to accelerate communication between processors. Ali-Graph [16] used a storage layer to cache the vertices and their intermediate result to reduce communication cost. Bai et al. [28] presented an efficient data loader to store frequently accessed vertices in the cache using a novel indexing algorithm that speeds up the acquisition of information between processors to reduce communication time. However, the communication cost and storage overhead are enormous when the scale of the intermediate result is large.

The second class of methods designs the sampling strategy to communicate as little vertex and edge information as possible between processors. Jiang and Rumi [18] provided different sampling probabilities for vertices stored on the current processor and other processors. By assigning a higher sampling probability to the vertices stored on the current processor, it can reduce the visiting frequency of the vertices stored in other processors and achieve the purpose of reducing communication costs between processors. Zhao et al. [29] designed an efficient scheduling transmission network to find the shortest communication route of the processor to reduce communication time. However, these methods need to transfer all the dependent vertices information stored on other processors in the worst case.

The third class of methods stores the communicated vertices in each processor. For example, DistDGL [30] used the sampling strategy in GCN [31], which randomly replicates the
communicated vertices and edges through a given probability from other processors. Angerd et al. [32] used the sampling strategy in GraphSAGE [33], which augments each subgraph with a small number of edges and vertices from other subgraphs uniformly and randomly. Additionally, some sampling strategies provided in single-machine GCN training can also be extended to replicate vertices for distributed optimization. ClusterGCN [4] considered the relationships of subgraph vertices. It divided similar vertices into the same cluster (subgraph) and replicated similar vertices into the cluster (subgraph) according to the degree of vertices. GraphSAINT [34] proposes regularization and variance reduction formulas to improve the quality of replicated vertices. Although these distributed optimization methods can avoid vertices and edge communication between processors, the number of replicating vertices in the dense graph is too large to be acceptable due to the unexpected large memory cost. Furthermore, these methods did not consider the difference in variance between graph data stored in each processor. In this work, we will solve these research challenges to achieve higher accuracy and faster training speed while replicating as few vertices as possible.

III. GAD FRAMEWORK

In this section, we describe the training procedure for GAD. We first introduce the overall framework of GAD in Section III-A, followed by GAD-Partition in Section III-B, and finally present the distributed GCN training and its optimization in Sections III-C and III-D, respectively.

A. Framework Overview

As illustrated in Fig. 1, the GAD framework comprises two key components: GAD-Partition and GAD-Optimizer. Compared with other graph partition methods, GAD-Partition generates augmented partitions with important peripheral vertices, which can greatly help downstream tasks. GAD-Optimizer further improves efficiency by incorporating subgraph variance. GAD-Partition and GAD-Optimizer each consist of three steps.

1) GAD-Partition:
   a) Use the multilevel graph partition method to divide the data graph \( G \) into subgraphs \( g_i \). 
   b) Form the augmented graph \( g'_i \) by replicating the important vertices from subgraph \( G \) to subgraph \( g_i \), which avoids frequency communication in distributed GCN training.
   c) Assign the augmented subgraphs to different processors \( p_i \) with the graph variance-based importance score \( \zeta \).

2) GAD-Optimizer:
   a) Calculate the graph variance-based importance score \( \zeta \), which is based on variations in accessing probability for vertices in \( g' \).
   b) Train a GCN model at each processor \( p_i \) using the augmented subgraphs loaded in \( p_i \).
   c) Aggregate the gradient of each GCN model in \( p_i \) with the graph variance-based importance score \( \zeta \).

B. GAD-Partition

GAD-Partition aims to divide the graph into several roughly equal partitions and extend the partitions by replicating the vertices with significant communication. A straightforward solution is to replicate the vertices that need to be communicated from other processors evenly. However, it requires manually adjusting the number of replication vertices to achieve low redundancy and high accuracy because a higher number of replication vertices does not affect the accuracy but increase training time.

To overcome this issue and achieve high accuracy under minimum redundancy, we first employ the multilevel graph partition method that minimizes the candidate replication vertices. Then, we propose a Monte-Carlo-based vertex importance measurement and importance-first sampling method to replicate high-quality vertices with structural regularities from the candidate replication vertices. Finally, we assign the subgraphs to different processors. In the following, we first describe the procedure of graph partition in detail.

1) Graph Partition: Graph partitioning divides a graph into several disjoint partitions. It can be defined as follows.

   **Definition 1 (Graph Partition):** Given an undirected graph \( G = (V, E) \), graph partition divides \( G \) into \( k \) disjoint subgraphs \( g_1, \ldots, g_k \) with vertices sets \( V_1, \ldots, V_k \) such that \( \bigcup_{i=1}^{k} V_i = V \).

   **Definition 2 (Boundary Vertices of Subgraph):** Given an undirected graph \( G = (V, E) \) and a subgraph \( g_i = (V_i, E_i) \) of \( G \), the boundary-vertex set \( B(g_i) \) of \( g_i \) is the set \( \{ v \mid v \in V_i \land \exists u \in V \setminus V_i : e_{u,v} \in E \} \).

   **Definition 3 (Candidate Replication Vertex Set of Subgraph):** Given a GCN layer number \( m \), an undirected graph \( G = (V, E) \), and a subgraph \( g_i = (V_i, E_i) \) of \( G \), the candidate-replication vertex set \( V_i \) of \( g_i \), denoted as \( C(g_i) \), is the intersection of all \( m \)-hop neighbors of the vertices in \( B(g_i) \) and the vertices in \( V \setminus V_i \).
Fig. 2. Example of a candidate-replication vertex set. Assume the GCN layer number \( m = 2 \). The boundary vertex set \( B(g_1) \) of subgraph \( g_1 \) contains only \( v_1 \) and the candidate replication vertex set contains \( b_1, b_2, b_3 \).

Fig. 2 shows one example of a candidate-replication vertex set. To minimize the number of candidate replication vertex in each subgraph and keep the balance of computational load in the distributed environment, our target of graph partition is to divide a graph satisfying

\[
\min \left( |E| - \sum_{i=1}^{k} |E_i| \right) \tag{1}
\]

subject to

\[
|V_i| \leq (1 + \epsilon) \left\lceil \frac{|V|}{k} \right\rceil \tag{2}
\]

where \( k \) is the number of partitions, \( \epsilon \) is the imbalance constant, indicating the tolerable difference in the number of vertices on the partitions [20]. Intuitively, (1) minimizes edge cuts between subgraphs, and (2) ensures the approximate balance for the number of vertices on each subgraph.

To balance the number of vertices across the partitioned subgraphs [see (2)] and minimize the number of candidate replication vertices [see (1)], inspired by Metis [35], we use the multilevel graph partitioning method. Intuitively, we first coarsen the input graph to a much smaller graph, and then iterative bisect this much smaller graph. Finally, we project the much smaller graph back to the input graph. Fig. 3 shows the overview of the multilevel graph partition method. In the following, we describe the three major steps in detail.

1) **Coarsening Phase:** To reduce the graph size while maintaining its important properties, we iteratively contract the graph \( \mathcal{G} = (V, E) \) to the coarsen graph \( \mathcal{G}_c = (V'_c, E'_c) \) so that \( \mathcal{G}_c \subseteq \mathcal{G} \). For graph \( \mathcal{G}_c \), we initialize every vertex’s and edge’s weight to 1. To generate the coarsened graph \( \mathcal{G}_c \), we randomly select the number of vertices \( \{\hat{v}_1, \ldots, \hat{v}_m\} \) from \( \mathcal{G}_{c-1} \). For each \( \hat{v}_i \) in \( \{\hat{v}_1, \ldots, \hat{v}_m\} \), we find its adjacent edge with the maximum weight, e.g., the edge is denoted as \( (\hat{v}_i, \hat{v}'_i) \). If multiple adjacent edges have the same maximum weight, we randomly select one of them. Then we merge \( \hat{v}_i \) and \( \hat{v}'_i \) as a coarsened vertex where the weight of the coarsened vertex is the aggregation of the weight of two merging vertices, and its connected edge’s weight is the sum of the edges to be merged. We repeat the above steps to continuously coarsen the graph until the number of vertices in \( \mathcal{G}_c \) is reduced to a certain value, e.g., 20\% number of vertices in the input graph.

2) **Partition Phase:** For a given coarsened graph \( \mathcal{G}_c \), we divide \( \mathcal{G}_c \) into \( k \) coarsened partitions \( \hat{g}_1, \ldots, \hat{g}_k \) based on the constraints described in (1) and (2). To do so, first we randomly choose \( k \) vertices from the current level \( \mathcal{G}_i \) as the seeds of \( k \) partitions \( \hat{g}_1, \ldots, \hat{g}_k \). Then, for each partition \( \hat{g}_i \in \{\hat{g}_1, \ldots, \hat{g}_k\} \), we expand the partition \( \hat{g}_i \) by adding the vertices that satisfy: a) these vertices are not contained in \( \hat{g}_i \) and have the edges linked to a vertex of \( \hat{g}_i \) and b) the edges between the added vertices and the vertices of \( \hat{g}_i \) have the maximum weight. The above procedure is repeated until the sum of vertex weights in each partition satisfies the constraint in (2). After that, there may be some vertices that do not belong to any partitions. To solve this problem, we pick up each vertex and add it to the nearest partition. To achieve the goal in (1), we run the above procedure many times and obtain multiple partition candidate results. Finally, we select the approximate optimal result by taking the result with the minimum \( |E| - \sum_{i=1}^{k} |E_i| \), denoted as the coarsened partitions \( \hat{g}_1, \ldots, \hat{g}_k \).

3) **Uncoarsening Phase:** For each coarsened subgraph \( \hat{g}_i \in \{\hat{g}_1, \ldots, \hat{g}_k\} \), we obtain its corresponding partition \( g_i \) by uncoarsening the coarsened vertices in \( \hat{g}_i \). In other words, we can obtain the graph partitions \( \{g_1, \ldots, g_k\} \) of the input graph.

4) **Local Subgraph Augmentation:** After obtaining subgraphs, we propose a strategy to add high-quality replication vertices with structural regularities for each subgraph according to the important measurement of each candidate replication vertex and the importance-first selection of candidate replication vertices.

a) **Importance measurement:** To assign an importance weight to each candidate replication vertex, the common practice considers the structure around the vertex, i.e., use the degree of the vertex as the importance weight. However, it does not work in our case, as the candidate replication vertices are also associated with different parts of the subgraph. We use random walk (RW) to capture the structure information and dependencies between candidate replication vertices and subgraphs to overcome the challenge mentioned above. In addition, we use the Monte Carlo approach to
approximate the frequency of occurrence of each candidate replication vertex in RW. In particular, for each subgraph \( g_i = (v_i, e_i) \), we randomly select a vertex from the boundary vertex set \( B(g_i) \) of \( g_i \) as the beginning of RW and form an RW sequence \( RW_j \). We repeat this operation \( n \) times and traverse RW sequences to calculate the important weight \( I(v) \) for each \( v \in C(g_i) \), which can be measured as follows:

\[
I(v) = \frac{\sum_{j=1}^{n} RW_j(v)}{n}, \quad v \notin v_i
\]

(3)

where \( RW_j(v) = 1 \) if \( v \) in the \( j \)th RW sequence, otherwise \( RW_j(v) = 0 \).

To identify the length \( l \) of the RW sequence, we present a property as below.

**Property 1:** Given a subgraph \( g_i = (v_i, e_i) \), all candidate replication vertices of \( g_i \) are covered by RW sequences without irrelevant vertices when the length of the RW sequence \( l \) is equal to the number of GCN layers.

**Proof:** If we set RW with length \( l \), it can reach all the \( l \) hop neighbors. Based on Definition 3, if \( l \) is equal to the number of GCN layers and starts from the vertices in \( B(g_i) \), RW can reach all the candidate replication vertices. Thus, \( l \) is equal to the number of GCN layers.

To converge \( l(v) \) to an acceptable range in minimal time, we need to minimize \( n \) in (3). In essence, (3) is a kind of Monte Carlo method, which simulates the probability distribution of sampling through multiple enumerations. Thus, we can estimate the enumeration times based on the Monte Carlo error [36]. Formally, the Monte Carlo error \( E \) can be measured as follows:

\[
E = \frac{z \cdot \sigma}{\sqrt{n}}
\]

(4)

where \( z \) is a statistic number representing a certain confidence interval. We set the 95% confidence level and \( E = 0.05 \) to ensure that the estimation is close to the true value. It can be described as, for a 95% confidence interval, the estimation will not differ by more than 5% from the truth. When we have a 95% confidence level, we can get the value of \( z \) from the Z test that \( z = 1.96. \) \( \tilde{x} \) and \( \sigma \) are the sample mean and the standard deviation, respectively.

The detailed procedure is presented in Algorithm 1 (Lines 1–17). We iteratively choose a subgraph \( g_i \) from the partitions (Line 1). For each \( g_i \), the RW sequence collection \( R \) is initialized as empty (Line 2). Then we add a small number of RW sequences to \( R \) (Lines 3–8), and estimate the mean \( \tilde{x} \) and variance \( \sigma^2 \) of \( I(v) \) for each \( v \) in \( R \) (Lines 9 and 10). Next, we estimate the required repetitions \( n \) using (4) with \( \tilde{x} \) and \( \sigma \) (Line 11). Finally, we add \( n \) RW sequences to \( R \) (Lines 12–16) and calculate \( I(v) \) for each \( v \) in \( R \) (Line 17).

As such, the time complexity of this procedure is composed of the following two steps. First, we iteratively enumerate the RW sequences from each subgraph in \( O(kz^2) \), where \( k \) is the number of subgraphs and \( z^2 \) is the Z test value of the confidence level we set. Second, we iteratively sort the frequency of each vertex that appeared in RW sequences for each subgraph in \( O(k|V| \log |V|) \) using a binary sort algorithm, where \( |V| \) is the number of vertices appearing in RW sequences. Therefore, the total time complexity of this processor is \( O(k(z^2 + |V| \log |V|)) \).

**b) Importance based augmentation:** To overcome the aforementioned challenge of vertex replication (balance high accuracy and low redundancy), we introduce graph density \( d(g_i) \) to compute the number of replication vertices \( n(g_i) \) for subgraph \( g_i \).

**Definition 4 (Graph Density):** For a given subgraph \( g_i \), with the vertices number \( |v_i| \) and edge number \( |e_i| \), the density of \( g_i \) can be computed as

\[
d(g_i) = \frac{2|e_i|}{|v_i|(|v_i| - 1)}.
\]

(5)

**Example 1:** As shown in Fig. 4, the high-density subgraph [Fig. 4(b) with three edges cut] has higher information loss than the low-density subgraph [Fig. 4(a) with one edge cut] under the same partition. Therefore, more vertices need to be replicated for density graphs to maintain accuracy.

Note that the graph density \( d(g_i) \) ranges from 0 to 1. In general, it can be treated as a complement of the replicated vertices. Here, we determine the number of replicate vertices \( n(g_i) \) according to \( d(g_i) \), which can be measured as below

\[
n(g_i) = \alpha(1 + d(g_i))|v|
\]

(6)

where \( \alpha \) is a hyperparameter to control the proportion of replicated vertices. Generally, our method can work well even though we set \( \alpha \) to the default value, i.e., \( \alpha = 0.01 \).

We need to consider how to select important vertices based on \( I(v) \). The naive solution is to copy the most important replication vertices. But it may have dangling vertices, which means that the selected replication vertices have no path to \( g_i \). To solve this problem, we propose an importance-first selection strategy. The detailed procedure is presented in Algorithm 1 (Lines 18–26). First, we calculate \( n(g_i) \) for \( g_i \) (Line 18), then compute the important score \( I(RW_i) \) for each RW, through sum \( I(v) \) for \( v \) in \( RW_i \), and iteratively select RW with maximum \( I(RW) \) (Line 20). Next, we add the vertices \( v \) in RW to the replication vertex set \( v' \) that never appear in \( v' \) until the number of replication vertex is equal to \( n(g_i) \) (Lines 21–25). Finally, we add \( v' \) back to subgraph \( g' \) with the connected edges \( e_i' \) (Line 26).

**3) Subgraph Loading:** To allocate augmented subgraphs to multiple processors and achieve workload balance in the distributed GCN training, we allocate a similar number of vertices to each processor. For the given augmented subgraph sequence \( g'_1, \ldots, g'_k \) and processors \( p_1, \ldots, p_n \), we iteratively
be measured as follows:

\[ \nabla W = \frac{\partial L}{\partial \hat{y}} = \left( -\frac{y_i}{\hat{y}} + \frac{y_j}{1 - \sum_{i=1, j \neq i}^k \hat{y}} \right) \]  

(10)

For the gradient, it can be denoted as

where \( \partial L / \partial \hat{y} \) stands for the gradient \( \hat{y} \) under \( L \). After obtaining the gradient in each processor, the distributed GCN training will aggregate all the gradients and send the average gradient back to each processor for parameter update. This process is called global consensus.

Definition 5 (Global Consensus [19]): Given a gradient sequence \( \{ \nabla W_1, \nabla W_2, \ldots, \nabla W_n \} \) calculated by GCN models in processors \( \{p_1, p_2, \ldots, p_n\} \), the global consensus can be measured as follows:

\[ \nabla W = \frac{1}{n} \left( \nabla W_1 + \nabla W_2 + \ldots + \nabla W_n \right). \]  

(11)

Finally, we use \( \nabla W \) and learning rate \( \eta \) to update the parameters of GCN models in each \( p_i \) synchronously, which can be denoted as

\[ W_i = W_i - \eta \frac{\partial L}{\partial \nabla W}. \]  

(12)

After updating the parameters, the distributed GCN training framework synchronously selects the next subgraph as a minibatch to train GCN models in each processor until all the subgraphs are traversed. We repeat this process until the loss function converges.

D. GAD-Optimizer

For any GCN training method, we should consider the problem of slow convergence and poor generalization caused by the variance of subgraphs. A general approach is to reduce subgraph variance by optimizing the sampling vertex selection strategy [5], [6], [34], but it cannot be applied to GAD-Partition because graph partition has different ways to form subgraphs. To solve this problem, we propose the GAD-Optimizer, which reduces the impact of high-variance subgraphs by assigning a low importance weight to the trained gradient of the subgraph. Here we separate GAD-Optimizer into two parts: variance-based subgraphs importance assignment and weighted global consensus.

1) Variance-Based Subgraph Importance: To measure the subgraph variance \( \text{var}(g) \), we follow the approach described in [34] for the sampling-based subgraph \( g \), which can be measured as follows:

\[ \text{var}(g) = \sum_{i,j \in V} \frac{H_i^{(l)}}{p_i \alpha_{ij}} d(i,j) \]  

(13)

where \( l \) denotes the layer number in a GCN model, \( p_i \) is the probability of selecting the vertex \( v_i \), \( \alpha_{uv} \) is the normalization constant and calculated as \( \alpha_{uv} = (C_{uv} / C_v) \). \( C \) is the vertex sampling times, and \( d(i,j) \) is the Euclidean distance between vertices \( v_i \) and \( v_j \).

Intuitively, the variance of the sampling-based subgraph is calculated by accumulating the variance between two vertices in the sampling layers. Since the subgraphs in our work are generated by partition instead of sampling, we can obtain

select and allocate \( g_i \) to \( p_i \) if \( p_i \) has the least number of vertices.

C. Distributed GCN Training

In this section, we describe the standard distributed GCN training process. Following [19], we first perform forward propagation in each processor, which is measured as follows:

\[ H_p^{(l+1)} = F(\tilde{A} \cdot H_p^{(l)} \cdot W_p^{(l)}) \]  

(7)

where \( l \) denotes the GCN layer number, \( F \) is an activation function, such as rectified linear unit (ReLU) or LeakyReLU, \( p \) is the \( p \)th processor, and \( W_p^{(l)} \) is the trainable weight matrix of \( l \). Prediction \( \hat{y} \) can be calculated by forward propagation stacking. Specifically, a two-layer stacking to calculate \( \hat{y} \) can be denoted below

\[ \hat{y} = \text{softmax}(\tilde{A} \cdot F(\tilde{A} \cdot H^{(0)} \cdot W^{(1)}) \cdot W^{(2)}). \]  

(8)

Second, we execute the backward propagation to calculate the loss and gradient at each processor. The first loss layer can be measured as follows:

\[ \mathcal{L} = - \sum_{i=1}^N y_i^{(i)} \log \hat{y}_i^{(i)} + (1 - y_i^{(i)}) \log (1 - \hat{y}_i^{(i)}). \]  

(9)
var(g) directly by accumulating the variance between two vertices in g. Thus, to compute \((H^{(t)}_g)/(p_{α,j,i})\) in (13), we only need to consider the difference in vertex degree without the dependence of vertices during sampling. We provide the following property to measure the difference of vertex degree for subgraphs generated by the partition method.

Property 2: Given a subgraph \(g_i\) and its vertex selection probability \(p(v)\), \(\sum_{i,j} p(v_i)p(v_j)\) is large if the difference of vertex degree in \(g_i\) is small.

Proof: Consider that we have \(n\) vertices such that \(V = \{v_1, \ldots, v_n\}\). Assume that we have the probability \(p(v_i) = ((1+t)/n), p(v_2) = ((1-t)/n), \ldots, p(v_n) = ((1-t)/n)\) for each vertex, so that \(\sum_{v_i \in V} p(v_i) = 1\) in a special case. Maximizing \(\sum_{i,j} p(v_i)p(v_j)\) requires maximizing \(((n^2/2) + ((n^2/2) - 2) + \cdots + 1)\) \(\left\{(1+t)/n\right\} \cdot \left\{(1+t)/n\right\} \cdot \left\{(1-t)/n\right\} \cdot \left\{(1-t)/n\right\}\), which can be simplified as \((1/n^2 - 1)\) \(= \left((n^2 - 1)\right)/n^2\). Since \(n\) is constants, thus, to maximize \(\sum_{i,j} p(v_i)p(v_j)\), it is required to minimize \((n^2)/n^2\), which can be achieved when \(t = 0\). Therefore, \(\sum_{i,j} p(v_i)p(v_j)\) increases when we have a small difference of \(p(v_i)\).

According to Property 2, we replace \([H^{(t)}_g]/(p_{α,j,i})\) with \(\sum_{i,j} p(v_i)p(v_j)\) in (13). Additionally, since the difference in vertex degrees is inversely proportional to the variance-based important weight \(\xi\), we change it from the denominator to the numerator. To this end, we can write \(\xi\) as

\[
\xi(g') = \frac{\sum_{i,j} p(v_i)p(v_j)}{d(i,j) + \beta}
\]

where \(\beta\) is the constant to prevent the denominator from being zero. Normally \(\beta = 1\). \(\xi(g')\) can be calculated in the distributed environment during the subgraph loading process (described in Section III-B3). Furthermore, \(v_1, v_2\) in (14) can be extended to multiple dimensions by calculating each dimension separately. We illustrate the measure \(\xi(g')\) of subgraph \(g'\) with an example.

Example 2: Fig. 5 demonstrates the variance-based important weight of three subgraphs with different degree distributions. Notice that the distance of the vertices in each subgraph is zero, i.e., \(d(i,j) = 0\) for any two vertices \(v_i, v_j\) in a subgraph. We can observe that Fig. 4(b) has the highest variance with degree sequence \(2, 2, 2, 1\). The subgraph in Fig. 4(a) has the lowest variance with sequence \(2, 2, 1, 1\). According to (14), \(\xi(g')\) is 3.75, 3.59, 3.61 for Fig. 4(a)–(c), respectively.

2) Weighted Global Consensus: To reduce the impact of variance for the subgraphs generated by GAD-Partition, we extend the global consensus (see Definition 5) by providing less weight to subgraphs with higher variance. Specifically, (11) can be rewritten as

\[
\nabla \hat{W} = \frac{1}{\sum_{j=1}^{n} \xi_j} \sum_{j=1}^{n} \xi_j \nabla W_1 + \xi_2 \nabla W_2 + \ldots + \xi_n \nabla W_n.
\]

To update the parameters of GCN, we can rewrite (12) as

\[
W_t = W_{t-1} - \eta \frac{\partial L}{\partial \nabla \hat{W}}.
\]

We provide the distributed GCN training and GAD-Optimizer process in Algorithm 2. Lines 1–4 indicate the process of subgraph allocation and initialization. Lines 5–9 describe distributed GCN training. Finally, Lines 10–12 describe the weighted global consensus and the parameter update. Compared with single-machine GCN training, distributed GCN training has global consensus at the end of each iteration, i.e., the weighted average of the gradient. As such, the time complexity of Algorithm 2 is the same as the time complexity of the implemented GCN model.

IV. EXPERIMENTS

This section first presents experimental setups such as datasets, baseline methods, and parameter configuration, followed by the analysis of the effectiveness, precision, and scalability of the GAD framework and the effect of graph augmentation and weighted global consensus. To eliminate the impact of different parallelization architectures on training time, we implemented our framework and all baseline models under single thread, single process, and multiple GPU processors. Experiments are implemented using Pytorch 1.5.0 with Python 3.7.7 and run on four NVIDIA GeForce GTX 1080 Ti 11-GB Memory with no NVLink connections.
TABLE I
DATASET STATISTICS

| Dataset  | Nodes | Edges  | Labels | Features | Data splits of Training/Validation/Test (%) |
|----------|-------|--------|--------|----------|-------------------------------------------|
| Cora     | 2,708 | 5,429  | 7      | 1433     | 45/18/37 (%)                             |
| Pubmed   | 19,717| 44,324 | 3      | 500      | 92/03/05 (%)                             |
| Flicker  | 89,250| 899,756| 7      | 500      | 50/25/25 (%)                             |
| Reddit   | 231,443| 11,606,919| 41  | 602      | 70/20/10 (%)                             |
| Amazon   | 1,569,960| 132,169,734| 107  | 200      | 85/05/10 (%)                             |

A. Experiment Setup

1) Datasets: We perform extensive experiments on four real-world benchmark datasets to demonstrate the effectiveness of our distributed GCN optimization framework. All benchmark datasets follow “fixed-partition” splits, which have been discussed in [37]. Table I shows the statistics for all the datasets as follows.

1) Cora is a citation network with the scientific papers as the vertices and their citation relationships as the edges. Here, each node is represented by a 1433-D word vector.

2) Pubmed is a citation network. The vertices are composed of scientific publications on diabetes, represented by the tf-idf word vector in a dictionary of unique words. The edges represent the citation relationships of the publications.

3) Flicker is an image network. The vertices are composed of online images, represented by a 500-D description vector. The edges represent the relationship between images.

4) Reddit is a user-interaction network. The vertices are posts on the Reddit forum, represented by the time and title vector of the user comment. The edges indicate that the same person comments on two posts.

5) Amazon is a product review network. The nodes are products on the Amazon website, represented by the vector of the reviewer comment. The edges indicate that the two products are bought by the same user.

2) Baseline Methods: We compare our framework with the following baseline methods implemented in a distributed environment.

1) Distributed GCN [31] is a classical graph network model. It uses convolution computation to gather information from neighboring nodes within a graph.

2) Distributed GraphSAGE [33] is a framework for inductive representation learning on large graphs. It gathers information by sampling a fixed number of neighbor nodes uniformly. It provides a strategy for learning the different aggregation functions on a different number of hops.

3) Distributed Cluster-GCN [4] is a training method for scalable training of a graph network model using stochastic gradient descent. It samples a dense subgraph identified by a graph clustering algorithm and restricts the neighborhood search within this subgraph.

4) Distributed GraphSAINT [34] is a graph sampling-based inductive learning method. It constructs mini-batches by sampling the training graph. Here, we compare its three sampling strategies and indicate them as GraphSAINT-Node, GraphSAINT-Edge, and GraphSAINT-RW. Specifically, GraphSAINT-Node is the layer-based random node sampling, GraphSAINT-Edge is the layer-based random edge sampling, and GraphSAINT-RW is the RW sampling.

3) Parameter Configuration: We have considered four parameters in our experiments: learning rate $\eta$, GCN layer number $l$, batch size $b$, and number of hidden neurons $h$.

To compare performance with baselines, we set the default learning rate $\eta = 0.0001$ for the Cora, Flicker, Reddit, Amazon datasets, and $\eta = 0.001$ for the Pubmed dataset. Furthermore, we set the default batch size $b = 300$ for the Cora, Flicker, Reddit, Amazon datasets, and $b = 1500$ for the Pubmed dataset. To determine the number of GCN layers $l$ and the hidden number of neurons $h$, we use the optimal parameter settings for each experiment by varying $l$ and $h$ so that $l \in \{2, 3, 4\}$ and $h \in \{128, 256, 512\}$.

B. Evaluation of Effectiveness and Accuracy

Fig. 6 and Table II show the comparison of accuracy and convergence. Clearly, our framework achieves significantly higher accuracy and faster convergence speed on all datasets. Specifically, the accuracy of GAD can achieve higher accuracy for Cora ($\uparrow 0.67\%$), Pubmed ($\uparrow 0.47\%$), Flicker ($\uparrow 0.21\%$), Reddit ($\uparrow 0.41\%$), and Amazon ($\uparrow 0.73\%$) in comparison to best performer.

In addition, the comparison of training time with the benchmark methods is shown in Fig. 7. We average the training time for Cora, Pubmed, Flicker, and Reddit under the same sampling method. The results show that GAD improves the training speed by $2 \times 1.7 \times, 2.3 \times, 3.1 \times, 2.1 \times$, and $1.8 \times$ compared to distributed GCN, GraphSAGE, ClusterGCN, GraphSAINT-Node, GraphSAINT-Edge, and GraphSAINT-RW, respectively.

Notice that our method uses the same graph partitioning method and GCN training model as distributed ClusterGCN [4]. Therefore, we can conclude that the improvement of accuracy and training speed is improved by the graph augmentation and weighted global consensus strategies (discussed in Sections III-B and III-D). Although it has less improvement in accuracy (0.21%–0.73% improvement) over distributed ClusterGCN, the total running speed improves significantly (1.7×–3.1× improvement). Furthermore, the distributed GraphSAINT-Edge sampler has a higher computational complexity. Thus, it does not support large-scale datasets. In our experiment, we did not report the comparison with the distributed GraphSAINT-Edge on the Flicker and Reddit datasets.
Fig. 6. Accuracy curves of the test data on (a) Cora, (b) Flicker, (c) Pubmed, (d) Reddit, and (e) Amazon.

We further compare the graph partition time and total time (including training time and partition time) of GAD and Cluster-GCN when the number of partitions is 50. As shown in Table III, the GAD has a slightly longer partition time for Cora, Pubmed, Flicker, and Reddit than the distributed Cluster-GCN method. This is because the augmentation process takes extra running time in the GAD-partition method. In addition, the total running time of GAD is much smaller than that of the cluster-GCN. This is because the GAD speeds up the training time, and the extra running time of the augmentation process is much smaller than the training time.

We also evaluate the scalability of the framework with the number of GPUs. As shown in Table IV, the GAD framework maintains consistent accuracy and reduces training costs as the number of GPUs increases. For example, for the Pubmed dataset, the accuracy decreases from 0.8160 to 0.7558 when the number of GPUs increases from 1 to 8. This indicates that the framework is scalable and can be used in multi-GPU training.

C. Scalability of GPU and Layers

To demonstrate the scalability of deep layers in multi-GPU training, we evaluate our framework’s training accuracy and training speed with the different number of layers and number of GPU. Generally, the GCN model with more number of layers is enormous complex in training [5], [31], [33]. In contrast, adding more layers to the network is effective and can improve expressive power, which can result in improved performance [38]. Thus, in our framework, we aim to maintain consistent accuracy and reduce training costs as the number of GPUs increases. Table IV shows the GCN training accuracy with different values of \( l \) and \( h \) such that \( l \in \{2, 3, 4\} \) and \( h \in \{128, 256, 512\} \), respectively, and different number of GPUs as 1–8 for the Pubmed dataset. We can observe that there is no significant loss in accuracy for a single GPU.
TABLE IV  
SCALABILITY OF GAD ON TEST ACCURACY WHEN GPU AND LAYER NUMBER VARY

| GPU Number | 2 Layers | 3 Layers | 4 Layers |
|------------|----------|----------|----------|
| 1 GPU      | 0.7443   | 0.7398   | 0.7540   |
| 2 GPUs     | 0.7378   | 0.7330   | 0.7518   |
| 3 GPUs     | 0.7428   | 0.7304   | 0.7490   |
| 4 GPUs     | 0.7369   | 0.7345   | 0.7503   |

Fig. 8. Scalability of GAD in training time when GPU and layer number vary.

Compared with the training accuracy by using the single machine GCN training (one GPU) with multiple distributed GPUs under a different number of layers, there is no significant loss in accuracy, i.e., the fluctuation of accuracy is less than 0.01 in the distributed GCN training environment.

In addition, Fig. 8 describes the average training time for each epoch with a variable number of GPUs (1–8) and \( l = 2, 3, 4 \). We can see that the training time decreases with an increasing GPU number. Besides, we can observe that the time cost of GCN training decreases nonlinearly. It tends to converge into a constant with an increasing GPU number. This is because communication and blocking take extra time during distributed training.

D. Evaluation of Graph Augmentation

To validate the effectiveness of graph augmentation in distributed GCN training (described in Section III-B), we evaluate the performance in terms of accuracy, allocated memory, and communication size with and without graph augmentation in single-machine training (one GPU) and distributed training (multi-GPU). Table V describes the comparative analysis of the proposed framework on the Cora and Pubmed datasets. We can notice that compared with single machine training (one GPU) without graph augmentation, the distributed training (four GPUs) without graph augmentation has comparatively low accuracy for Cora (\( \downarrow 2.84\% \)) and Pubmed (\( \downarrow 3.94\% \)). This is because there is some information loss during the communication of node information in the distributed training. When the graph augmentation is applied, the accuracy is increased significantly to achieve the rate being close to the single machine training accuracy in both datasets.

In addition, we also test the memory and communication size with/without graph augmentation when the number of GPUs is 4 for both datasets. To compare these performances under single-machine training, we tested when the number of GPUs is 1. We can observe a significant decrease (54.55\% and 52.35\% for Cora and Pubmed, respectively) in communication overhead. This happens because we store the important nodes of other processors when these nodes are frequently used and communicated to the current processors. Furthermore, the average allocated memory increases slightly by 5.37\%. This happens because we need to replicate some communication nodes and add them to each subgraph. Furthermore, we also find that using more GPUs in a distributed environment can effectively reduce the allocated memory per GPU. It is because the training data stored in multi-GPUs is less than a single GPU. For example, when the number of GPU is 4, the total allocated memory for Cora is 16.32 MB (4.08 \( \times 4 \)), which is slightly larger than the memory training in the single-machine environment (13.39 MB). This is because distributed training stores a small number of intermediate results on each GPU.

Furthermore, Fig. 9 shows the effect of graph augmentation on the number of graph partitions when the number of partitions is set to 10, 50, 100 and \( l = 4, h = 512 \) in the Pubmed dataset. Specifically, Fig. 9(a) and (b) present the results of the loss convergence with/without subgraph augmentation, respectively. From the results, we can see that GCN training without graph augmentation has different losses under different partition numbers in Fig. 9(b). This happens because partitioning the input graph causes a loss of structure information when training the boundary vertices. Thus, we need to balance the loss and the number of partitions when training GCN without augmentation. When graph augmentation is used, the information loss caused by partitioning can be compensated for. Thus, the convergence of loss for different partition numbers remains consistent in Fig. 9(a). In addition, Fig. 10 shows the effect of graph augmentation when applying GAD-Optimizer under 50 and 100 partition numbers in the Pubmed dataset. From the result, we can see that the GAD-Partition, i.e., graph partition with augmentation, has a lower loss. This can also support that when graph augmentation is used, the information loss caused by partitioning can be compensated.

E. Evaluation of GAD-Optimizer

To validate the effectiveness of the GAD-Optimizer, we compared the impact of the GAD-Optimizer on the convergence speed and loss. Fig. 11(a) and (b) show results with the Flicker dataset when the number of partitions is set to 50 and 100, respectively. In this evaluation, we set \( h = 128 \) and \( l = 4 \). The experimental results show a lower convergence loss and a
TABLE V
IMPACT OF GRAPH AUGMENTATION WHEN GPU NUMBER VARIES

| Dataset | Number of GPU | Augmentation | Accuracy | Allocated Memory per GPU (MB) | Communication Size (MB) |
|---------|---------------|--------------|----------|-------------------------------|------------------------|
| Cora    | 1             | No           | 0.8067   | 13.39                         | 0                      |
| Cora    | 4             | No           | 0.7783   | 3.86                          | 0.11                   |
| Cora    | 4             | Yes          | 0.8037   | 4.08                          | 0.05                   |
| Pubmed  | 1             | No           | 0.7532   | 38.91                         | 0                      |
| Pubmed  | 4             | No           | 0.7358   | 10.62                         | 3.61                   |
| Pubmed  | 4             | Yes          | 0.7423   | 11.14                         | 1.72                   |

Fig. 10. Impact of graph partition method. (a) Partition number = 50. (b) Partition number = 100.

Fig. 11. Impact of weighted global consensus. (a) Number of partition = 50. (b) Number of partition = 100.

V. CONCLUSION
In this article, we presented a novel GAD to reduce processor communication costs and accelerate GCN training efficiency. At the same time, GAD achieves a high accuracy for distributed GCN training. In particular, we designed GAD-Partition to generate subgraphs with important communication information through Monte Carlo-based vertex importance measurement and the depth-first sampling strategy. In addition, we developed a GAD-Optimizer to reduce the impact of high graph variance by assigning a low importance weight to the gradient of the subgraph when the GCN parameters are updated. To show the performance of our proposed framework, we conducted an extensive experiment and compared six baseline methods on four real-world datasets. The experimental results demonstrated high accuracy, good stability, and low communication costs of our GAD in different configuration settings, e.g., the accuracy of GAD can achieve about 0.816, 0.756, 0.49, 0.931, and 0.763 on the test dataset of Cora, Pubmed, Flicker, Reddit, and Amazon. Besides, there is a 1.7–3.1 times improvement in training time than the baseline methods. Thus, we can conclude that the proposed GAD can be deployed to learn large-scale graphs in real applications.

REFERENCES
[1] T. Zhao, N. Cui, Y. Chen, and M. Li, “Efficient strategy mining for football social network,” Complexity, vol. 2020, Sep. 2020, Art. no. 8823189.
[2] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, “The graph neural network model,” IEEE Trans. Neural Netw., vol. 20, no. 1, pp. 61–80, Jan. 2009.
[3] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, “A comprehensive survey on graph neural networks,” IEEE Trans. Neural Netw. Learn. Syst., vol. 32, no. 1, pp. 4–24, Jan. 2020.
[4] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, “Cluster-GCN: An efficient algorithm for training deep and large graph convolutional networks,” in Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, Jul. 2019, pp. 257–266.
[5] J. Chen, T. Ma, and C. Xiao, “FastGCN: Fast learning with graph convolutional networks via importance sampling,” in Proc. ICLR, 2018, pp. 1–15.
[6] W. Huang, T. Zhang, Y. Rong, and J. Huang, “Adaptive sampling towards fast graph representation learning,” in Proc. NeurIPS, 2018, pp. 4563–4572.
[7] V. G. Satorras and J. B. Estrach, “Few-shot learning with graph neural networks,” in Proc. ICLR, 2018, pp. 1–13.
[8] X. Song, J. Li, Y. Tang, T. Zhao, Y. Chen, and Z. Guan, “JKT: A joint graph convolutional network based deep knowledge tracing,” Inf. Sci., vol. 580, pp. 510–523, Nov. 2021.
[9] T. Ma, H. Wang, L. Zhang, Y. Tian, and N. Al-Nabhan, “Graph classification based on structural features of significant nodes and spatial convolutional neural networks,” Neurocomputing, vol. 423, pp. 639–650, Jan. 2021.
[10] X. Song, J. Li, Q. Lei, W. Zhao, Y. Chen, and A. Mian, “Bi-CLKT: Bi-graph contrastive learning based knowledge tracing,” Knowl.-Based Syst., vol. 241, Apr. 2022, Art. no. 108274.
[11] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su, “ArnetMiner: Extraction and mining of academic social networks,” in Proc. 14th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, Aug. 2008, pp. 990–998.
[12] T. P. Tanon, D. Vrandecic, S. Schaffert, T. Steiner, and L. Pintscher, “From freebase to Wikidata: The great migration,” in Proc. 25th Int. Conf. World Wide Web, Apr. 2016, pp. 1419–1428.

[13] S. Zhao et al., “VPipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel DNN training,” IEEE Trans. Parallel Distrib. Syst., vol. 33, no. 3, pp. 489–506, Mar. 2022.

[14] I.-H. Chang et al., “Parallel deep neural network training for big data on blue Gene/Q,” IEEE Trans. Parallel Distrib. Syst., vol. 28, no. 6, pp. 1703–1714, Jun. 2017.

[15] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and M. Wang et al., “Deep graph library: A graph-centric, highly-performant package for graph neural networks,” 2019, arXiv:1909.01315.

[16] P. Jiang and M. A. Rumi, “Communication-efficient sampling for distributed training of graph convolutional networks,” 2021, arXiv:2101.07706.

[17] S. Scardapane, I. Spinelli, and P. D. Lorenzo, “Distributed training of graph convolutional networks,” IEEE Trans. Signal Inf. Process. over Netw., vol. 7, pp. 87–100, 2021.

[18] G. Karypis and V. Kumar, “Multilevel k-way partitioning scheme for irregular graphs,” J. Parallel Distrib. Comput., vol. 48, no. 1, pp. 96–129, 1998.

[19] J. Chen, J. Zhu, and L. Song, “Stochastic training of graph convolutional networks with variance reduction,” in Proc. ICML, in Proceedings of Machine Learning Research, vol. 80, 2018, pp. 941–949.

[20] J. Chen, M. Zhong, J. Li, D. Wang, T. Qian, and H. Tu, “Effective deep attributed network representation learning with topology adapted smoothing,” IEEE Trans. Cybern., vol. 52, no. 7, pp. 1–12, Mar. 2021, doi: 10.1109/TCYB.2021.3064082.

[21] J. Chen, M. Zhong, J. Li, J. Zhu, C. Zhai, and R. Kong, “Dynamic network embedding survey,” 2021, arXiv:2103.15447.

[22] M. Fey and J. E. Lenssen, “Fast graph representation learning with PyTorch geometric,” 2019, arXiv:1903.02428.

[23] D. Zhang et al., “AGL: A scalable system for industrial-purpose graph machine learning,” Proc. VLDB Endowment, vol. 13, no. 12, pp. 3125–3137, Aug. 2020.

[24] B. Liu, Z. Ding, and C. Ly, “Distributed training for multi-layer neural networks by consensus,” IEEE Trans. Neural Netw. Learn. Syst., vol. 31, no. 5, pp. 1771–1778, May 2020.

[25] S. J. Yoo, “Distributed consensus tracking for multiple uncertain nonlinear strict-feedback systems under a directed graph,” IEEE Trans. Neural Netw. Learn. Syst., vol. 24, no. 4, pp. 666–672, Apr. 2013.

[26] Y. Bai et al., “Efficient data loader for fast sampling-based GNN training on large graphs,” IEEE Trans. Parallel Distrib. Syst., vol. 32, no. 10, pp. 2541–2556, Oct. 2021.

[27] Z. Zhao, G. Verma, C. Rao, A. Swami, and S. Segarra, “Distributed scheduling using graph neural networks,” 2020, arXiv:2011.09430.

[28] D. Zheng et al., “DistDGL: Distributed graph neural network training for billion-scale graphs,” 2020, arXiv:2010.05337.

[29] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” in Proc. ICLR, 2017, pp. 1–14.

[30] A. Anger, K. Balasubramanian, and M. Annamalai, “Distributed training of graph convolutional networks using subgraph approximation,” 2020, arXiv:2012.04930.

[31] W. L. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” in Proc. NIPS, 2017, pp. 1024–1034.

[32] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. K. Prasanna, “GraphSAINT: Graph sampling based inductive learning method,” in Proc. ICLR, 2020, pp. 1–19.

[33] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” SIAM J. Sci. Comput., vol. 20, no. 1, pp. 359–392, Aug. 1998.

[34] M. Driels and Y. Shin, “Determining the number of iterations for Monte Carlo simulations of weapon effectiveness,” Tech. Rep., 2004.

[35] W. Hu et al., “Open graph benchmark: Datasets for machine learning on graphs,” 2020, arXiv:2005.00687.

[36] M. Telgarsky, “Benefits of depth in neural networks,” in Proc. 29th Annul. Conf. Learn. Theory, in JMLR Workshop and Conference Proceedings, vol. 49, 2016, pp. 1517–1539.

Taige Zhao received the M.S. degree from The University of Sydney, Sydney, NSW, Australia, in 2020. He is currently pursuing the Ph.D. degree with the School of Information Technology, Deakin University, Geelong, VIC, Australia.

His research interests include graph represents learning, distributed computing, deep learning, and natural language processing.

Xiangyu Song received the B.S. degree from Beijing Jiaotong University, Beijing, China, in 2013. He is currently pursuing the Ph.D. degree with the School of Information Technology, Deakin University, Geelong, VIC, Australia.

His research interests include educational knowledge data mining, representation learning, and deep learning on graph.

Man Li received the M.S. degree from Northeaster University, Shenyang, China, in 2006. She is currently pursuing the Ph.D. degree with the School of Information Technology, Deakin University, Geelong, VIC, Australia.

Her research interests include similarity computation, data mining, and machine learning on time series data.

Jianxin Li (Senior Member, IEEE) received the master’s degree from Northeastern University, Shenyang, China, in 2005, and the Ph.D. degree in computer science from the Swinburne University of Technology, Melbourne, VIC, Australia, in 2009. He is currently an Associate Professor in data science with the School of Information Technology, Deakin University, Geelong, VIC, Australia. His research interests include graph database query processing and optimization, social network analytics and computing, representation learning, and online learning analytics.

Wei Luo received the Ph.D. degree in computer science from Simon Fraser University, Burnaby, BC, Canada, in 2007, where he received training in statistics, machine learning, computational logic, and modern software development.

He is currently a Senior Lecturer in data science with Deakin University, Geelong, VIC, Australia. His research interests include machine learning under uncertainties.

Imran Razzak (Senior Member, IEEE) received the master’s degree in machine learning from International Islamic University, Islamabad, Pakistan, in 2009, and the Ph.D. degree in data science from the University of Technology Sydney, Ultimo, NSW, Australia, in 2012.

He is currently a Senior Lecturer in computer science with the School of Information Technology, Deakin University, Geelong, VIC, Australia. His research interests include machine learning in general with an emphasis to natural language processing and image analysis to solve real-world problems related to health, finance, social media, and security.