ADGNN: Towards Scalable GNN Training with Aggregation-Difference Aware Sampling

ZHEN SONG, School of Computer Science and Engineering, Northeastern University, China
YU GU*, School of Computer Science and Engineering, Northeastern University, China
TIANYI LI*, Department of Computer Science, Aalborg University, Denmark
QING SUN, School of Computer Science and Engineering, Northeastern University, China
YANFENG ZHANG, School of Computer Science and Engineering, Northeastern University, China
CHRISTIAN S. JENSEN, Department of Computer Science, Aalborg University, Denmark
GE YU, School of Computer Science and Engineering, Northeastern University, China

Distributed computing is promising to enable large-scale graph neural network (GNN) model training. However, care is needed to avoid excessive computational and communication overheads. Sampling is promising in terms of enabling scalability, and sampling techniques have been proposed to reduce training costs. However, online sampling introduces large overheads, and while offline sampling that is done only once can eliminate such overheads, it instead introduces information loss and accuracy degradation. Thus, existing sampling techniques are unable to improve simultaneously both efficiency and accuracy, particularly at low sampling rates. We develop a distributed system, ADGNN, for full-batch based GNN training that adopts a hybrid sampling architecture to enable a trade-off between efficiency and accuracy. Specifically, ADGNN employs sampling result reuse techniques to reduce the cost associated with sampling and thus improve training efficiency. To alleviate accuracy degradation, we introduce a new metric, *Aggregation Difference* (*AD*), that quantifies the gap between sampled and full neighbor set aggregation. We present so-called AD-Sampling that aims to minimize the Aggregation Difference with an adaptive sampling frequency tuner. Finally, ADGNN employs an *AD*-importance-based sampling technique for remote neighbors to further reduce communication costs. Experiments on five real datasets show that ADGNN is able to outperform the state-of-the-art by up to nearly 9 times in terms of efficiency, while achieving comparable accuracy to the non-sampling methods.

CCS Concepts: • **Computing methodologies** → **Distributed computing methodologies**; • **Information systems** → **Data management systems**.

Additional Key Words and Phrases: distributed systems, sampling techniques, communication reduction, aggregation difference, graph neural networks

*Corresponding Authors

Authors' addresses: Zhen Song, School of Computer Science and Engineering, Northeastern University, Shenyang, China, songzhen_neu@163.com; Yu Gu, School of Computer Science and Engineering, Northeastern University, Shenyang, China, guyu@mail.neu.edu.cn; Tianyi Li, Department of Computer Science, Aalborg University, Aalborg, Denmark, tianyi@cs.aau.dk; Qing Sun, School of Computer Science and Engineering, Northeastern University, Shenyang, China, sunqing_neu@163.com; Yanfeng Zhang, School of Computer Science and Engineering, Northeastern University, Shenyang, China, Zhangyf@mail.neu.edu.cn; Christian S. Jensen, Department of Computer Science, Aalborg University, Aalborg, Denmark, csj@cs.aau.dk; Ge Yu, School of Computer Science and Engineering, Northeastern University, Shenyang, China, yuge@mail.neu.edu.cn.

2836-6573/2023/12-ART229 $15.00
https://doi.org/10.1145/3626716

ACM Reference Format:
Zhen Song, Yu Gu, Tianyi Li, Qing Sun, Yanfeng Zhang, Christian S. Jensen, and Ge Yu. 2023. ADGNN: Towards Scalable GNN Training with Aggregation-Difference Aware Sampling. *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 229 (December 2023), 26 pages. https://doi.org/10.1145/3626716

1 INTRODUCTION

Graph Neural Networks (GNNs) have gained increasingly widespread popularity in fields such as recommendation [3, 11, 43], natural language processing [23, 51], image analysis [40, 41], knowledge graphs [12, 17, 21, 39], due to its remarkable performance. To further enhance the accuracy of GNN models and extend their applicability, numerous variants of GNNs have been proposed, including GraphSAGE [14], GAT [33], and GIN [42]. However, as GNNs are being applied in increasingly large-scale settings, model training on a single machine is becoming computationally infeasible.

As a result, different distributed systems for GNN training have been proposed. These systems aim to reduce communication and computational costs during training; however, they focus primarily on improved engineering techniques, such as partitioning [32, 50], system framework design [37], caching [52], compression [29], and serverless parallelism [31]. While such techniques are useful, they do not necessarily reduce the training complexity. Sampling, however, is a promising approach to decrease complexity, and it is used commonly in existing GNN training systems [29, 46, 50, 52].

Sampling in GNN training is to sample the adjacent vertices (i.e., neighbors) of the target vertex, which comes in two forms: online sampling and offline sampling. The former selects neighbors in each iteration (DistDGL [50], AliGraph [52]), while the latter selects neighbors only once for an entire training process (AGL [46], EC-Graph [29]). Online sampling can achieve higher testing accuracy, but at the expense of a substantial sampling overhead. Offline sampling is used rarely due to its tendency to yield low accuracy results. Neither sampling mode effectively balances accuracy and efficiency. Additionally, existing sampling techniques [1, 4–6, 14, 16, 43–45, 53] are designed primarily for single-machine settings and adopt an online sampling mode. These techniques also do not take communication costs and overheads incurred by distributed sampling (e.g., distributed subgraph construction and distributed embedding organization) into consideration, which can result in suboptimal efficiency. While a few approaches [34, 35] exist that target distributed environments, they focus on the reduction of communication costs and come with high computational and distributed subgraph construction overheads. We summarize the challenges faced by existing sampling techniques and sampling-based systems as follows:

Challenge 1: High Sampling Costs. Existing proposals for GNN training are often hampered by high sampling costs [4, 6, 16, 50, 53], caused by the traversal of adjacency lists, masking operations, and the building of distributed sampled graphs in each iteration. The high costs limit the scalability of these solutions and their practical use in large-scale applications. There is a potential for more efficient sampling techniques to improve the efficiency of GNN training.

Challenge 2: Low Accuracy. Existing sampling techniques suffer from low accuracy when using a small fanout [4–6, 14, 16, 34, 53], where *fanout* refers to the number of neighbors that are retained after sampling. The reason is that they only sample neighbors based on graph topologies and do not take into account the unique features of GNN training. A sampling technique designed explicitly for GNN training can avoid or reduce accuracy loss.

Challenge 3: Poor Scalability. Most existing sampling techniques [4, 6, 53] are designed with single-machine GPU training in mind, limiting their applicability to large-scale GNN training. While a few studies [34, 35] explore efficient distributed sampling, they focus mainly on reducing communication costs and are constrained by computational, sampling costs. Thus, an efficient distributed sampling technique that can alleviate bottlenecks related to limited memory, sampling and training costs is desirable.

To advance the state-of-the-art in large-scale GNN training, we develop ADGNN, a distributed system for full-batch GNN training that utilizes a hybrid sampling architecture combining offline and online strategies. This architecture enhances the model convergence process through online sampling and reduces distributed sampling costs by leveraging the benefits of offline sampling (i.e., reusing sampling results). Next, we propose a new metric called Aggregation Difference (*AD*) to quantify the difference between full-neighbor and sampled-neighbor aggregation. To mitigate the errors caused by reusing sampling results, we introduce a sampling technique called AD-Sampling to minimize *AD*. We provide theoretical support for the performance of AD-Sampling. Further, we propose an adaptive sampling frequency tuner to dynamically choose the optimal sampling frequency (i.e., rounds with sampling result reuse). We provide theoretical justification for the effectiveness of the adaptive tuner. Finally, to alleviate the communication bottleneck in distributed scenarios, we introduce a sampling technique based on *AD*-based importance, specifically targeting remote neighbors. The goal is to reduce the proportion of remote neighbors. We provide a proof that the node-wise *AD*-minimum sampling technique is superior to layer-wise and subgraph-wise techniques.

The main contributions can be summarized as follows.

- First, we develop a distributed system, ADGNN, for GNN training that utilizes novel hybrid sampling techniques to reduce the computational and communication costs of full-batch training. We introduce a new metric, *Aggregation Difference* (*AD*), that quantifies the aggregation gap between sampled and full neighbor sets.
- Second, we present so-called AD-Sampling that reduces difference and online sampling costs. We offer a theoretical upper bound on the *AD* value for the current iteration according to the sampling results from the previous iteration and factors that affect convergence. Based on this, we proposed an adaptive tuner to adjust the sampling frequency.
- Third, we introduce optimizations that enable efficient computation of *AD* by pruning degree sizes and reducing the numbers of calculations required. We also propose an *AD*-importance-based sampling strategy to contend with remote neighbors that utilizes a layer-wise pruning approach to reduce communication costs.
- Fourth, experiments on five real datasets show that ADGNN can improve on the state-of-the-art by a factor of up to nearly 9 in terms of efficiency, while achieving accuracy comparable to that of full-neighbor training.

The organization of this paper is as follows. In Section 2, we provide an overview of related work on GNN training modes, sampling techniques, and optimizations for efficient GNN training. We introduce the preliminaries of a general GNN model in Section 3 and the framework of ADGNN in Section 4. Section 5 presents the AD-Sampling techniques and optimizations that reduce the computational cost of *AD*, and Section 6 further enhances ADGNN by adaptively adjusting the sampling frequency and communication reduction. We present our experimental study in Section 7, followed by our conclusion in Section 8.

2 RELATED WORK

2.1 Training Modes for GNN

Mini-Batch vs. Full-Batch. Full-batch GNN training [18, 24, 29] typically achieves better convergence and accuracy, but is not feasible for large-scale graphs due to large intermediate results that exceed GPU memory. In contrast, mini-batch training [50] is more adaptable to GPU training, but can suffer from slower convergence and subgraph construction.

Sampled-Neighbor vs. Full-Neighbor. Sampled-Neighbor training [4, 6, 14] reduces computational, memory consumption, and communication costs. However, poorly designed sampling can

Table 1. Representative Sampling Techniques and Sampling-Based Distributed Systems for GNN Training

Method	Architecture	Characteristics	Distributed	Training Cost	Statistics	Generalization
GraphSAGE [14]	online	sampling each iteration: high cost, but high accuracy	×	computation, sampling, and subgraph construction	random	poor due to considering only the graph topology
FastGCN [4]	online		×		topology	
ClusterGCN [6]	online		×		random	
FOS [48]	online		×		random	
BNS-GCN [34]	online		✓	computation, communication, distributed sampling, and distributed subgraph construction, distributed embedding organization	random	
DistDGL [50]	online		✓		random	
AliGraph [52]	online		✓		random	
AGL [46]	offline	sample once: low cost, but low accuracy	✓		random	
EC-Graph [29]	offline		✓		random	
ADGNN (ours)	hybrid	periodic sampling	✓		feature	powerful

degrade convergence and accuracy. Full-neighbor training [18, 24, 29] is accurate but computationally and memory intensive.

GPU vs. CPU. GPU training [18, 22, 37, 50] enables parallelism, but offers limited memory and cannot handle large-scale graphs. While mini-batch training can extend the applicability of GPUs, convergence often suffers. Conversely, CPU training [29, 46, 52] is slower but has less stringent memory and cost constraints.

ADGNN Training Modes. ADGNN adopts full-batch and sampling modes for two main reasons. First, full-batch training typically yields better convergence and accuracy than mini-batch training. Second, sampling techniques help alleviate GNN computation bottlenecks. ADGNN supports both CPU and GPU training. The former benefits from ample memory resources, while the latter can accelerate training when the available memory is sufficient.

2.2 Sampling-Based GNN Training

Node, Layer, and Subgraph-Wise Sampling. Sampling for GNNs can be classified into *node-wise*, *layer-wise*, and *subgraph-wise* sampling. Node-wise sampling [5, 7, 8, 14, 27, 43] enables fine-grained sampling without requiring additional masking operations but also incurs sampling costs. Layer-wise sampling [4, 16, 53] samples vertices by layers, while subgraph-wise sampling [1, 6, 28, 44, 45] divides an initial graph into subgraphs and samples a subset of subgraphs in each iteration. Layer-wise and subgraph-wise sampling incur lower sampling costs but require extra masking operations that are time-consuming on CPU clusters. All the techniques referenced above are designed primarily for a single machine and disregard communication. Additionally, they disregard the impact of features, a unique characteristic of GNNs, as they focus on sampling according to the graph topology.

Sampling Techniques for Distributed Environments. Two studies of sampling methods for distributed GNN training exist: BNS-GCN [34] and DGS [35]. However, the proposed techniques only sample remote neighbors and exhibit sub-optimal performance when the computational costs (such as tensor operation on CPUs, distributed subgraph construction, and distributed embedding organization) are substantial. In addition, DGS requires training an explanation model iteratively on CPUs, which can be time-consuming, particularly for full-batch training modes. In contrast, our method avoids iterative training and proposes leveraging a heuristic algorithm to reduce the cost of sampling. DGS can also suffer from stale inference due to parallel processing between training GNN models and explanation models.

Online Sampling and Offline Sampling. There are two main types of sampling architectures: online and offline, as shown in Table 1. Online sampling involves performing sampling in each iteration, which can be computationally expensive. Techniques such as FOS [48], GraphSAGE [14], ClusterGCN [6], FastGCN [4], and BNS-GCN [34] fall into this category, along with systems like DistDGL [50] and AliGraph [52]. Conversely, offline sampling is performed once, reusing results

but potentially reducing accuracy. Representative systems in this category include AGL [46] and EC-Graph [29]. Overall, the existing techniques all have shortcomings. Developing a sampling technique that can achieve satisfactory accuracy at a low cost in our setting remains a challenge.

Difference from Variance-Reduction Methods. ADGNN differs fundamentally from traditional sampling techniques that aim to minimize variance. Traditional techniques focus on identifying the optimal transfer probability of Monte Carlo methods to achieve the smallest variance. In contrast, our proposal employs a heuristic technique to select an optimal neighbor combination every m iterations. Furthermore, existing variance-reduction sampling techniques [2, 4–6, 53] only focus on topology information of graphs, such as degrees. MVS-GCN [7] proposes a minimal variance sampling technique by retaining historical embeddings, but still does not use feature information for sampling.

2.3 Optimizations for Efficient GNN Training

Many optimizations have been proposed for distributed GNN training. DistDGL [50] improves the METIS [19] partitioning method to adapt GNN training by reducing the communication between machines and balancing the workload. EC-Graph [29] leverages aggressive, lossy compression to reduce the communication of embeddings and embedding gradients and improves convergence and accuracy using compensation methods. AliGraph [52] reduces the communication between computing workers and graph servers by caching frequently accessed vertices. Various techniques have been proposed to target different aspects of GNN training, including communication compression [29], vertex caching [20, 37, 49, 52], disk-based GNN optimization [25, 46], embedding optimizations for decentralized gnn training [26], and pipeline parallelism [36]. Notably, ADGNN optimizes the efficiency and scalability issues of distributed GNN training through distributed data sampling. These optimizations are orthogonal, and can work in tandem with, our proposals. The paper's experimental study includes an end-to-end comparison with two representative distributed systems for GNN training, AliGraph [52] and DistDGL [50]. FOS [48] is a random sampling technique for feature matrices without statistical rules. Specifically, FOS randomly selects a starting feature and performs consecutive sampling from the feature matrix to reduce random reads. It is an online sampling technique, and it is designed for a single-machine environment. Thus, it does not explicitly account for communication and additional costs associated with sampling, e.g., constructing distributed subgraphs. Furthermore, there are additional distributed optimization endeavors pertaining to machine learning [30] and graph iterative tasks [13, 38] that have yet to be integrated into distributed GNN training. These works are orthogonal to ours.

3 PRELIMINARIES

We provide the preliminaries of Graph Neural Networks (GNNs) in this section. Specifically, we provide the definitions of attributed graphs and then outline the process of forward and backward propagations for GNN training.

Definition 3.1. An **attribute graph** is denoted as $\mathcal{G} = (\mathcal{V}, \mathcal{E}, X_{\mathcal{V}}, X_{\mathcal{E}})$, where $\mathcal{V}$ is a set of vertices v, $\mathcal{E}$ is a set of edges $e\ (v, u)$, and $X_{\mathcal{V}}$ and $X_{\mathcal{E}}$ are features of vertices and edges, respectively. $\mathcal{N}(v)$ denotes the neighbor set of v, meaning that $\forall u \in \mathcal{N}(v)((v, u) \in \mathcal{E} \vee v = u)$.

GNN algorithms typically employ both forward propagation (FP) and backward propagation (BP) to train on an attribute graph $\mathcal{G}$, each with L layers. We present the process within a single layer to illustrate a typical GNN algorithm. During FP, each vertex v collects embeddings of $\mathcal{N}(v)$ and aggregates them using the function *agg*. The aggregation of neighbor embeddings and the current vertex embedding $\boldsymbol{H}_v^{\ell-1}$ are combined using function *comb* to generate an intermediate

result Z_v^ℓ, as shown in Formula 1.

$$Z_v^\ell = comb\left(H_v^{\ell-1}, agg_{\mathcal{G}u\in\mathcal{N}(v)}(H_u^{\ell-1})\right) \tag{1}$$

$Next$, Z_v^ℓ is transformed using neural network operations with parameter W^ℓ for the ℓ^{th} layer and activation function σ to obtain a final representation of the current layer H_v^ℓ, as shown in Formula 2.

$$H_v^\ell = \sigma\left(Z_v^\ell \odot W^\ell\right) \tag{2}$$

After applying the GNN transformation with L layers, the resulting final embedding vector H_v^L can be utilized for downstream tasks such as node classification [14] and link prediction [47]. The final embedding is typically passed through a softmax function for classification or connected to a regression layer for prediction. The output of the softmax or regression layer is compared to a ground truth label using a suitable loss function $\mathcal{L}$.

The main goal of BP is to minimize $\mathcal{L}$, thereby improving the accuracy of the downstream tasks. First, the derivative of $\mathcal{L}$ with respect to the final embedding H_v^L is calculated, denoted as $G_v^L = \frac{\partial \mathcal{L}}{\partial H_v^L}$. Next, the derivative of $\mathcal{L}$ with respect to ℓ^{th} embedding H_v^ℓ $(1 \le \ell \le L)$, denoted as G_v^ℓ $(1 \le \ell \le L)$, is calculated using the chain rule:

$$G_v^\ell = A^T \odot G^{\ell+1} \odot (W^{\ell+1})^T, \tag{3}$$

where A^T is adjacency matrix A transposed. The formula represents the flow of gradients from the vertices in the $(\ell+1)^{th}$ layer to the vertices in the ℓ^{th} layer based on the reverse direction of FP. Based on Formula 3, we can obtain the gradients of the GNN models in each layer straightforwardly as $\frac{\partial G_v^\ell}{\partial W^{\ell-1}}$. After the backward propagation process, we incorporate the obtained gradients into the optimizer to update the GNN parameters.

4 SYSTEM FRAMEWORK

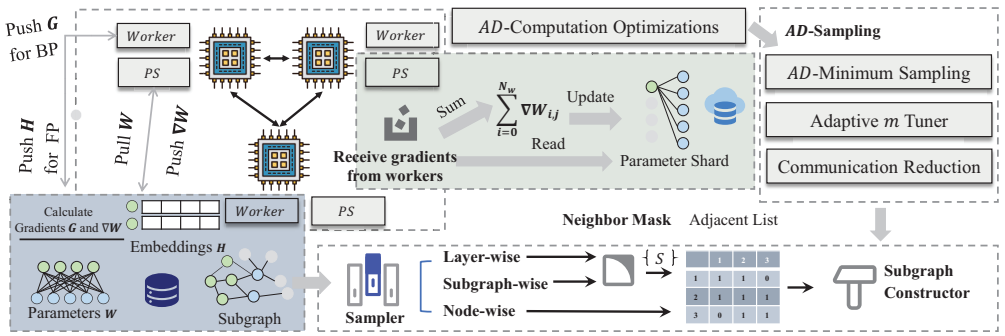


Fig. 1. System Overview

The proposed system ADGNN involves two logical roles, *servers* and *workers*, which are decoupled from physical machines. Fig. 1 shows an example configuration with three servers and three workers. It supports three main functionalities: (i) data and computation distribution, (ii) sampling and subgraph construction, and (iii) forward propagation and backward propagation.

Data and Computation Distribution. We utilize vertex partitioning (edge-cut) to distribute vertices from a large graph among workers, along with their associated features, adjacency tables, and labels. Initially, each worker reads a subgraph from disks that comprises vertices, features, adjacency lists, and labels. Next, the workers partition GNN models into N_w submodels and send

these to the corresponding server. The servers perform model storage and updates, whereas workers perform the main computations.

Sampling and Subgraph Construction. During training, each worker calls *Sampler* to recursively sample target vertices and neighbors for each layer. Users need to implement two functions: *Sample* and *Sample4EachLayer*. The former defines the entire sampling process for L layers, while the latter specifies the sampling rules for a single layer. After each sampling for a single layer, *Subgraph Constructor* generates a trainable subgraph object for a single GNN layer, which includes obtaining target vertices, encoding vertices, building the adjacency matrix, generating the initial feature matrix and constructing a routing table. The routing table specifies which vertex embeddings the current worker needs to send to other workers. We provide the following example to detail the process.

Example 4.1. Fig. 2 depicts the distributed sampling process for a 2-layer GNN model conducted by ADGNN with three workers wk_1, wk_2, and wk_3. Target nodes (the top-level vertices in Fig. 2) from the training set are assigned to workers using a partitioning method. We take the actions of wk_1 as an example. First, wk_1 randomly selects v_1 and v_2 as v_0's neighbors and v_1 and v_3 as v_1's neighbors according to a local adjacency list. Next, wk_1 unions $\{v_1, v_2\}$ and $\{v_1, v_3\}$ to indicate that it samples v_1, v_2, and v_3. These three vertices will be used in the previous layer. Finally, wk_1 notifies wk_2 to include v_3 in the next sampling round. After sampling for the second layer, we obtain the target vertices of the first layer, which are at the middle level. Note that the sampling process is reversed compared to the aggregation process in FP. In the second layer, wk_1 collects the embeddings of vertex v_3 from wk_2 and concatenates it with the local embedding matrix composed of v_1 and v_2.

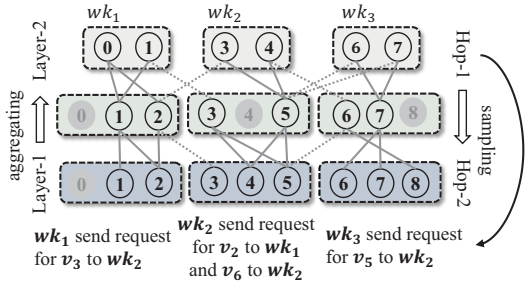


Fig. 2. Distributed Sampling (solid lines represent local edges, while dashed lines represent remote edges.)

FP and BP. During FP, ADGNN fetches trainable subgraphs of each layer sequentially, gathers the remote neighboring embeddings (i.e., embeddings residing on different machines from the target vertices), and concatenates them with local embeddings for aggregation. A computation graph is constructed during this process. During BP, ADGNN computes the embedding gradients for each layer. The embedding gradients are sent to neighboring vertices along the in-edge directions for the computation of embedding gradients of the previous layer. ADGNN calculates the gradients of model parameters, denoted as $\nabla \boldsymbol{W}$, for each layer of BP. These gradients are then sent to servers for updating the model.

5 AGGREGATION-DIFFERENCE AWARE SAMPLING

5.1 Aggregation Difference

Most existing sampling techniques used in GNN training either sample in each iteration, such as GraphSAGE [14] implemented in AliGraph [52] and DistDGL [50], or sample once during data-preparation, like the techniques used in AGL [46]. Neither of these techniques strikes a good balance

between accuracy and average epoch time. Thus, we propose to sample once each m iterations to amortize the sampling cost. The choice of m is covered later. However, simply applying it to random sampling, such as in GraphSAGE [14], can result in poor performance. This is because random sampling follows a discrete uniform distribution, i.e., $\hat{\boldsymbol{H}} \sim DiscreteUniform(\{\boldsymbol{H}_u | u \in \mathcal{N}(v)\})$, while we cannot guarantee that the variance remains low. Since the neighbor set that is sampled is used in the next $m-1$ iterations, a large deviation from the expected value can result in a long-term negative impact on the direction of the gradient descent. Therefore, although random sampling at intervals of m iterations can reduce the sampling cost in each iteration, it may lead to uncontrollable degradation of convergence due to significant deviations from expected values.

Therefore, it is crucial to measure and minimize the error induced by sampling. Existing techniques [4, 5, 53] aim to minimize the variance to approximate full aggregation while maintaining randomness in each sampling iteration only based on graph topologies. However, these techniques lack design considerations that consider the unique characteristic of GNNs, i.e., features. Motivated by this, we introduce a novel metric, *Aggregation Difference* (AD), that measures the sampling error and minimizes the error from the perspective of system optimizations, where we select the optimal neighbor combination to generate the aggregation.

Definition 5.1. Given a sampling technique $\mathcal{S}$, a vertex v, and a fanout k, $\mathcal{S}(v;k)$ denotes the **sampled neighbor set** using $\mathcal{S}$ with k.

We define *AD* as the squared Euclidean distance between the average value of full-neighbor aggregation and that of sampled aggregation. The average sampled aggregation approximates average full-neighbor aggregation, and the squared Euclidean distance can quantify the difference between two vectors. In the case of Example 5.3, the average aggregation of the full-neighbor set $\{v_2, v_3, v_4, v_5\}$ of v_1 is [-0.2, 0.4, 0.3], while the average aggregation of the sampled neighbor set $\{v_2, v_3\}$ is [-0.15, 0.45, 0.45]. The error of approximating the average aggregation result of $\{v_2, v_3, v_4, v_5\}$ with the average aggregation result of $\{v_2, v_3\}$ can be calculated using the squared Euclidean distance, which is 0.0275.

Definition 5.2. Given a sampled neighbor set $\mathcal{S}(v;k)$, $AD^{t,\ell}_{\mathcal{S}(v;k)}$ represents the ℓ^{th} layer's **Aggregation Difference** of vertex v in the t^{th} iteration with $\mathcal{S}(v;k)$, calculated as follows:

$$AD^{t,\ell}_{\mathcal{S}(v;k)} = || \frac{1}{|\mathcal{N}(v)|} \sum_{u \in \mathcal{N}(v)} \boldsymbol{H}^{t,\ell}_u - \frac{1}{k} \sum_{u \in \mathcal{S}(v;k)} \boldsymbol{H}^{t,\ell}_u ||^2, \tag{4}$$

where $\mathcal{N}(v)$ is the neighbor set of v and $\boldsymbol{H}^{t,\ell}_u$ denotes the ℓ^{th} layer embedding of vertex u in the t^{th} iteration.

Example 5.3. As shown in Figure 3a, the average aggregation result of v_1 is $[-0.2, 0.4, 0.3]$, while that of the sampled neighbor set $\{v_2, v_3\}$ is $[-0.15, 0.45, 0.45]$. Therefore, the aggregation difference is calculated as $(-0.2 + 0.15)^2 + (0.4 - 0.45)^2 + (0.3 - 0.45)^2 = 0.0275$.

We simplify $AD^{t,\ell}_{\mathcal{S}(v;k)}$ to $AD_{\mathcal{S}(v;k)}$ when the layer number and iteration count are unspecified. Similarly, we simplify $AD^{t,\ell}_{\mathcal{S}(v;k)}$ to $AD^{t,\ell}$ when the sampled neighbor set is unspecified.

Definition 5.4. Given a fanout k, the ***k*-optimal neighbor set** of a vertex v, denoted as $\mathcal{V}_{opt}(k;v)$, is the neighbor combination that has the minimum *AD* w.r.t full-neighbor aggregation, i.e., $\mathcal{V}_{opt}(k;v) = \arg\min_{s_v} AD_{s_v}$, where $s_v \subset \mathcal{N}(v) \wedge |s_v| \leq k$. $\mathcal{V}_{opt}(k;v)$ in the ℓ^{th} layer $(1 \leq \ell \leq L)$ of the t^{th} iteration is denoted as $\mathcal{V}^{t,\ell}_{opt}(k;v)$.

We aim to obtain the *k-optimal neighbor set* for each vertex in each layer. It is challenging to ensure that each mini-batch has a distribution that is similar to that of the entire training set. We

thus focus on the full-batch scenario, which typically represents a simpler and more direct way to achieve better convergence and accuracy. However, this necessitates a reduction of the time and memory costs of the sampling. We propose *Aggregation-Difference Aware Sampling* (AD-Sampling) to address the issue. It selects a subset of neighbors for each vertex by minimizing AD, thereby ensuring the selection of an optimal neighbor subset while adhering to a given sampling fanout. We use Example 5.5 to detail the process of AD-Sampling.

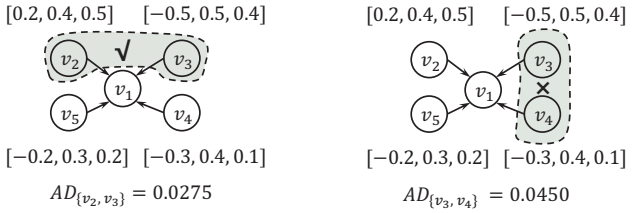


Fig. 3. Example of AD-Sampling

Example 5.5. Fig. 3 shows an example of sampling for vertex v_1, where $k = 2$. The embeddings of v_1's neighbors v_2, v_3, v_4, and v_5 are $[0.2, 0.4, 0.5]$, $[-0.5, 0.5, 0.4]$, $[-0.3, 0.4, 0.1]$, and $[-0.2, 0.3, 0.2]$, respectively. The full-neighbor aggregation value of v_1, i.e., $agg_{u \in \mathcal{N}(v)} \boldsymbol{H}_u$, is $[-0.2, 0.4, 0.3]$. The sampling aggregation values of v_2, v_3 and v_3, v_4 are $[-0.15, 0.45, 0.45]$ and $[-0.4, 0.45, 0.25]$, respectively. We obtain that the ADs of these two neighbor combinations are 0.0275 and 0.0450, respectively, by using Formula 4. Since a smaller AD indicates a better approximation, v_2, v_3 is the *k-optimal neighbor set.*

5.2 Heuristic AD-Sampling

Given a fanout k, our objective is to determine $\mathcal{V}_{opt}(k; v)$, which requires us to find the minimum of AD. However, the time complexity of minimizing AD for a vertex with $\mathcal{N}(v)$ neighbors is $\sum_{i=1}^{k} C^i_{\mathcal{N}(v)}$, which usually is unacceptable. This motivates us to develop a heuristic method that can compute AD efficiently.

We propose a heuristic AD-Sampling. Instead of selecting a neighbor combination with the smallest AD directly, we choose a neighbor with the maximum reduction in AD at each step and add it to the candidate k-optimal neighbor set. Algorithm 1 details the process. All embeddings $\boldsymbol{H}$ generated in the previous iteration are retrieved from the computation graph and utilized as input when computing AD in the current iteration. We perform k iterations for each target vertex (lines 4–10). This ensures that the size of the k-optimal neighbor set does not exceed the fanout k. We continue inserting neighboring vertices until the size of the candidate k-optimal neighbor set reaches k (line 4). In each iteration, we compute AD between $\mathcal{V}_{opt} \cup \{u\}$ and the full-neighbor aggregation for each target vertex v (line 6) and then we insert it into the map of AD, denoted as M (line 7). We compare this new aggregation with the full-neighbor aggregation AD_{last}. If $\boldsymbol{M}[v_s] \leq AD_{last}$, we add the vertex that results in the minimum AD to the candidate set; otherwise, we stop searching as no further AD reduction can be achieved (lines 9–11).

We propose an amortization strategy that samples once every m iterations to avoid a high computational cost of AD-Sampling. However, it is impossible to select an optimal $\mathcal{V}^{\ell}_{opt}$ for the next $m - 1$ iterations at the beginning, as AD values change in each iteration with the updates of GNN models. Thus, we need to analyze how this strategy impacts the selection of the k-optimal neighbor set as follows. We exclude the effect of activation functions to simplify the investigation.

Algorithm 1: Heuristic AD-Sampling

Input: Adjacency List $\boldsymbol{A}$, Embeddings $\boldsymbol{H}$, fanout k
Output: New Adjacency List $\hat{\boldsymbol{A}}$

for each $v \in \mathcal{V}$ **do**
  $\mathcal{V}_{opt} = \emptyset$, $\boldsymbol{M}$ = HashMap()
  $AD_{last} = \infty$, $\boldsymbol{A}_v = \text{copy}(\mathcal{N}(v))$
  `// Select` k `neighbors based on` AD
  while $|\mathcal{V}_{opt}| < k$ **do**
    for $u \in \boldsymbol{A}_v$ **do**
      $AD_{tmp} = \| \frac{1}{|\mathcal{N}(v)|} \sum_{i \in \mathcal{N}(v)} \boldsymbol{H}_i - \frac{1}{|\mathcal{V}_{opt}|+1} \sum_{i \in \mathcal{V}_{opt} \cup \{u\}} \boldsymbol{H}_i \|^2$
      $\boldsymbol{M}$.insert(u, AD_{tmp})
    `// Select` v `with the minimum` AD
    $v_{opt} = \arg\min_v(\boldsymbol{M})$
    if $\boldsymbol{M}[v_{opt}] > AD_{last}$ **then** break
    $\mathcal{V}_{opt}$.add(v_{opt}), $\boldsymbol{A}_v$.remove(v_{opt}), $AD_{last} = \boldsymbol{M}[v_{opt}]$
  $\hat{\boldsymbol{A}}$.append$(\mathcal{V}_{opt})$
return $\hat{\boldsymbol{A}}$

Lemma 5.6. *Assuming that the change of weight $\boldsymbol{W}$ of successive iterations satisfies $||\boldsymbol{W}^{t+1,\ell} - \boldsymbol{W}^{t,\ell}||^2 \leq \alpha \cdot ||\boldsymbol{W}^{t,\ell}||^2$, the difference of AD between the $(t+1)^{st}$ iteration and the t^{th} iteration satisfies $\mathbb{E}||\Delta AD_v^{t+1,\ell}||^2 \leq \alpha^{2\ell} \cdot \mathbb{E}||AD_v^{t,\ell}||^2$ (where $\mathbb{E}X$ represents the expectation of X), when using the sampling results of the $(t+1)^{st}$ iteration.*

Proof. We derive:

$$\begin{aligned}
&\mathbb{E}||\Delta AD_v^{t+1,\ell}||^2 = \mathbb{E}||AD_v^{t+1,\ell} - AD_v^{t,\ell}||^2 \\
&= \mathbb{E}||\Delta \boldsymbol{A}^\ell \prod_{i=1}^{\ell-1} \boldsymbol{A}^{t,i} \boldsymbol{H}^0 \prod_{j=1}^{\ell} \boldsymbol{W}^{t,j} - \Delta \boldsymbol{A}^\ell \prod_{i=1}^{\ell-1} \boldsymbol{A}^{t,i} \boldsymbol{H}^0 \prod_{j=1}^{\ell} (\boldsymbol{W}^{t,j} + \Delta \boldsymbol{W}^j)||^2 \\
&= \mathbb{E}||\Delta A^\ell \prod_{i=1}^{\ell-1} \boldsymbol{A}^{t,i} \boldsymbol{H}^0 \prod_{j=1}^{\ell} (\Delta \boldsymbol{W}^j)||^2 \leq \alpha^{2\ell} \cdot \mathbb{E}||AD_v^{t,\ell}||^2
\end{aligned}$$

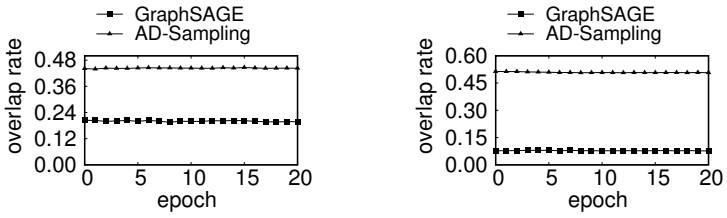
We conduct experiments on two real-world datasets, i.e., Pubmed and OGBN-Arxiv (see Section 7), to examine the dynamics of the k-optimal neighbor set over m successive iterations. In Fig. 4, the y-axis indicates the overlap rate between the current iteration and the first iteration. We compare AD-Sampling with GraphSAGE. Note that GraphSAGE employs random sampling. On the datasets, GraphSAGE only retains 20% and 8% of the vertices of the first sampling set, while the heuristic AD-Sampling can retain 45% and 51% of the vertices. Therefore, AD-Sampling using reuse technique performs significantly better than random sampling using reuse technique.

Next, we show that applying node-wise sampling to AD-Sampling always results in a smaller AD compared to applying layer-wise or subgraph-wise sampling.

Lemma 5.7. *Given an attributed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, X_\mathcal{V}, X_\mathcal{E})$, and node, layer, subgraph-wise sampling techniques, denoted by $\mathcal{S}_{nw}$, $\mathcal{S}_{lw}$, and $\mathcal{S}_{sw}$, respectively, we then have:*

$$\sum_{v \in \mathcal{V}} AD_{\mathcal{S}_{nw}(v;k)} \leq min\{\sum_{v \in \mathcal{V}} AD_{\mathcal{S}_{lw}(v;k)}, \sum_{v \in \mathcal{V}} AD_{\mathcal{S}_{sw}(v;k)}\} \tag{5}$$

Proof. Assume that both layer-wise and subgraph-wise sampling select a vertex u for a particular layer and subgraph, respectively. Then, v that samples u must satisfy $u \in \mathcal{N}(v)$. If u has



(a) Pubmed (b) OGBN-Arxiv

Fig. 4. Changes of the k-Optimal Neighbor Set

the minimal AD required to construct v then the node-wise sampling technique must also sample u for v, i.e., $AD_{\mathcal{S}_{nw}(v;k)} = min\{AD_{\mathcal{S}_{lw}(v;k)}, AD_{\mathcal{S}_{sw}(v;k)}\}$. Otherwise, the node-wise sampling technique selects a u with the minimal AD, i.e., $AD_{\mathcal{S}_{nw}(v;k)} < min\{AD_{\mathcal{S}_{lw}(v;k)}, AD_{\mathcal{S}_{sw}(v;k)}\}$. As a result, we obtain $AD_{\mathcal{S}_{nw}(v;k)} \leq min\{AD_{\mathcal{S}_{lw}(v;k)}, AD_{\mathcal{S}_{sw}(v;k)}\}$, which implies $\sum_{v\in\mathcal{V}} AD_{\mathcal{S}_{nw}(v;k)} \leq min\{\sum_{v\in\mathcal{V}} AD_{\mathcal{S}_{lw}(v;k)}, \sum_{v\in\mathcal{V}} AD_{\mathcal{S}_{sw}(v;k)}\}$ straightforwardly.

5.3 Optimizations for the Computation of *AD*

We propose several optimizations that improve the performance of computing AD. According to Algorithm 1, the time complexity of AD is $O(N \cdot k \cdot \bar{g} \cdot d)$, where N is the number of target vertices, k is the fanout, $\bar{g}$ is the average degree, and d is the dimension size of features. Although all four parameters can be optimized, we focus on reducing k and $\bar{g}$. Reducing k can decrease the number of neighbors that need to be considered, while decreasing $\bar{g}$ can limit the number of edges to traverse. We exclude optimization for N since it pertains to the mini-batch training mode, and we exclude optimization for d since it may introduce a significant amount of error due to one-hot encoding and activation functions.

Reducing k. We reduce the number of AD computations, which is equivalent to reducing k. To achieve this, we propose two strategies: Similarity-Priority and Reduction-Priority. The Similarity-Priority method selects the first k_{sp} vertices with the minimum ADs to form a set s_v that satisfies $s_v = \{u \in \mathcal{N}(v) \mid \text{rank}(\boldsymbol{M}[u]) \leq k_{sp}\}$, where $\text{rank}(\boldsymbol{M}[u])$ denotes the rank of element $\boldsymbol{M}[u]$ in map $\boldsymbol{M}$.

The Reduction-Priority method selects the vertex v with the minimum AD value in each step, i.e., $s_v = \{u \in \mathcal{N}(v) \mid u = \arg\min \boldsymbol{M}^i[u],\ 1 \leq i \leq k_{rp}\}$, which is repeated k_{rp} (a predefined hyperparameter for the Reduction-Priority method) times.

Similarity-Priority computes AD only once and tends to provide more stable results. However, it may not necessarily compute an AD that is closer to the optimal result. Next, Reduction-Priority is more likely to generate an AD that is closer to the optimal result, but it requires k AD computations. To balance efficiency and accuracy, we apply both strategies jointly. Specifically, we set a constant value k_c as the total number of AD computations, and select $\frac{k}{k_c}$ neighbors in each step. This allows us to effectively reduce the computational cost to a constant value that we control.

Reducing $\bar{g}$. In high-degree datasets, it is often the case that not all neighbors contribute equally to the aggregation of target vertices. To reduce the computational cost of AD in such cases, we propose to sample a subset of neighbors of size $k_g \cdot k$ ($1 \leq k_g \leq g_v$, where g_v is the degree of vertex v) from the adjacency list and calculate AD only using this subset. This reduces the cost of computing AD to $1/k_g$ of the original cost. We propose an additional optimization for $\bar{g}$ to reduce the computational cost of AD-Sampling. Specifically, when the fanout k of a vertex exceeds $\alpha \cdot g_v$ ($0 \leq \alpha \leq 1$), we retain the full-neighbor of v. This reduces the amount of AD sampling without significantly increasing the computational cost.

5.4 Analysis on *AD* Expectation and Time Complexity

We analyze the expectation of *AD* and the time complexity of AD-Sampling and state-of-the-art sampling techniques [4, 6, 14, 34, 46].

***AD* Expectation.** Given a vertex v, a fanout k, and a sampling technique $\mathcal{S}$, $AD_{\mathcal{S}(v,k)}$ measures the accuracy of sampling over a specific neighbor set $\mathcal{N}(v)$ (see Definiton 5.2). Thus, the expectation of *AD* over all possible sampled neighbor sets of all vertices provides insight into the effectiveness of the employed sampling technique. However, the expectation of *AD* is hard to identify, as it depends on the embeddings of each vertex $v \in \mathcal{V}$. Hence, we propose to determine the overlap between the neighbor set sampled by AD-Sampling and those obtained through a particular sampling technique, which can be used to approximately estimate the expectation of *AD*.

Definition 5.8. We randomly sample n vertices from the finite set of $|\mathcal{N}(v)|$ neighbors, where k vertices are in both $\mathcal{V}_{opt}$ and $\mathcal{N}(v)$. The number of vertices successfully sampled from $\mathcal{V}_{opt}$ without replacement is defined as ***AD* Expectation**, denoted as variable X.

AD Expectation X follows the hypergeometric distribution, i.e., $X \sim H(|\mathcal{N}(v)|, n, k)$. We express the expectation as a function of n to allow for the analysis of the expected values of different sampling techniques in Formula 6.

$$f_v(n) = \sum_{i=1}^{k} \frac{C_k^i \cdot C_{|\mathcal{N}(v)|-k}^{n-i}}{C_{|\mathcal{N}(v)|}^n} \cdot i = \frac{n \cdot k}{|\mathcal{N}(v)|} \tag{6}$$

Given a sampling fanout k, sampling techniques sample up to k neighbors for each vertex. For simplicity, we assume all vertices sample k neighbors. Different sampling techniques differ in their selection of neighbors. Node-wise sampling samples k neighbors for each vertex, so the expectation over all vertices is $\frac{1}{|\mathcal{V}|}\sum_{v\in\mathcal{V}} f_v(k) = \mathbb{E}_{v\in\mathcal{V}}\frac{k^2}{|\mathcal{N}(v)|}$. FastGCN [4] performs layer-wise sampling, and the number of neighbors for each vertex is $\frac{1}{|\mathcal{V}|}\sum_{v\in\mathcal{V}} f_v(\frac{k_l\cdot\bar{d}_l}{|\mathcal{E}|}\cdot|\mathcal{N}(v)|) = \frac{k_l\cdot\bar{d}_l\cdot k}{|\mathcal{E}|}$, where k_l is the sampling fanout for a layer and $\bar{d}_l$ is the average degree of the vertices sampled by FastGCN [4]. ClusterGCN [6] performs subgraph-wise sampling and samples k_s subgraphs from S subgraphs. Each worker can get $\frac{k_s}{N_{wk}}$ subgraphs, meaning that each vertex gets $\frac{k_s}{S\cdot N_{wk}}\cdot|\mathcal{N}(v)|$ neighbors. BNS-GCN [34] keeps all local neighbors, and the ratio of sampled vertices is $\frac{N_l+k_r}{|\mathcal{V}|}$, where N_l denotes the number of local neighbors and k_r denotes the sampling fanout (only sampling remote nodes). AGL-sampling [46] has the same expectation as random sampling, while it never changes at training epochs.

Time complexity. Node-wise sampling mainly contains two time-consuming stages: sampling and subgraph construction. Given the total number of epochs T, GraphSAGE [14] needs T sampling and subgraph-constructing stages, and each epoch involves sampling at each of ℓ layers. GraphSAGE [14] selects k neighbors for N vertices in each layer. Thus, constructing subgraphs for T epochs costs $O(T\cdot\mathcal{B})$, and the sampling costs $O(T\cdot N\cdot k\cdot L)$.

Next, layer-wise and subgraph-wise sampling techniques require another stage to mask adjacency lists. FastGCN [4] needs to traverse edges to get the sampling probability and mask with a layer-wise sampling neighbor set, which are both proportional to the number of edges $\mathcal{E}$. The sampling cost of ClusterGCN [6] is low since it is linear in the number of clusters. The main cost of ClusterGCN [6] consists of masking and graph construction. BNS-GCN [34] only needs to mask the remote edges, and the masking time is $O(T\cdot|\mathcal{E}_b|\cdot l)$, where $|\mathcal{E}_b|$ represents the number of boundary edges. AGL-sampling [46] only needs to sample once without the item T compared with random sampling. AD-Sampling amortizes the sampling and constructing costs into m epochs. The initial sampling complexity of *AD*-Sampling is $O(T\cdot N\cdot L\cdot d\cdot g\cdot k)$. Through optimizations, we make k a constant,

Table 2. *AD* Expectation and Time Complexity

Methods	*AD* Expectation	Time Complexity
GraphSAGE [14]	$\mathbb{E}_{v\in\mathcal{V}}\frac{k^2}{\lvert\mathcal{N}(v)\rvert}$	$O\ (T\cdot N\cdot L\cdot k+T\cdot\mathcal{B})$
FastGCN [4]	$\frac{k_l\cdot d_l\cdot k}{\lvert\mathcal{E}\rvert}$	$O\ (T\cdot L\cdot\lvert\mathcal{E}\rvert+T\cdot\mathcal{B})$
ClusterGCN [6]	$\frac{k_s\cdot k}{S\cdot N_{wk}}$	$O\ (T\cdot\lvert\mathcal{E}\rvert+T\cdot\mathcal{B})$
BNS-GCN [34]	$\frac{(N_l+k_r)\cdot k}{\lvert\mathcal{V}\rvert}$	$O\ (T\cdot L\cdot\lvert\mathcal{E}_b\rvert+T\cdot\mathcal{B})$
AGL-Sampling [46]	$\mathbb{E}_{v\in\mathcal{V}}\frac{k^2}{\lvert\mathcal{N}(v)\rvert}$	$O\ (N\cdot L\cdot k+\mathcal{B})$
AD-Sampling	k	$O\ (\frac{T\cdot N\cdot L\cdot d\cdot k+T\cdot\mathcal{B}}{m})$

reducing the cost to $O(T\cdot N\cdot L\cdot d\cdot g)$. We also relate candidate set size g to k, resulting in a cost of $O(T\cdot N\cdot L\cdot d\cdot k)$. As a result, the time complexity of sampling is reduced to $O(\frac{T\cdot N\cdot L\cdot d\cdot k}{m})$.

In the rest of the paper, AD-Sampling denotes Heuristic AD-Sampling (see Algorithm 1) with the optimizations presented in Section 5.3, unless stated otherwise.

6 OPTIMIZED AD-SAMPLING

6.1 Adaptive Sampling Frequency Tuner

AD-Sampling samples every m iterations (see Section 5). We propose to adaptively tune m based on the current convergence rate. This eliminates the need for manual hyperparameter tuning and increases the applicability of AD-Sampling across datasets and models. Since our primary goal is to perform sampling while minimizing *AD*, we investigate the factors that affect *AD*. We exclude the effect of activation functions to simplify the investigation.

Lemma 6.1. *Given a full-neighbor aggregation $\boldsymbol{H}_v^{t,\ell}=agg_{u\in\mathcal{N}(v)}\ \boldsymbol{H}_u^{t,\ell-1}$ and an approximate aggregation of sampled neighbor set $\hat{\boldsymbol{H}}_v^{t,\ell}=agg_{u\in\mathcal{S}(v;k)}\boldsymbol{H}_v^{t,\ell-1}$, we obtain the aggregation difference $AD^{t,\ell}\le||\Delta\boldsymbol{A}^{\ell}(\prod_{i=1}^{\ell-1}\hat{\boldsymbol{A}}^i)\boldsymbol{H}^0||^2\cdot||\prod_{i=1}^{\ell}\boldsymbol{W}^{t,i}||^2$, where $AD^{t,\ell}$ is the aggregation difference of the ℓ^{th} layer in the t^{th} iteration, $\Delta\boldsymbol{A}^{\ell}$ denotes the adjacency list composed of vertices that have not bSeen sampled in the ℓ^{th} layer, and $\hat{\boldsymbol{A}}$ represents the sampled adjacency list.*

Proof. We have:

$$\begin{aligned}
AD^{t,\ell}&=||\boldsymbol{H}^{t,\ell}-\hat{\boldsymbol{H}}^{t,\ell}||^2\\
&=||\boldsymbol{A}^{\ell}\boldsymbol{H}^{t,\ell-1}\boldsymbol{W}^{t,\ell}-\hat{\boldsymbol{A}}\boldsymbol{H}^{t,\ell-1}\boldsymbol{W}^{t,\ell}||^2\\
&=||\Delta\boldsymbol{A}^{\ell}\boldsymbol{H}^{t,\ell-1}\boldsymbol{W}^{t,\ell}||^2\\
&=||\Delta\boldsymbol{A}^{\ell}\hat{\boldsymbol{A}}^{\ell-1}\dots\hat{\boldsymbol{A}}^{1}\boldsymbol{H}^{0}\boldsymbol{W}^{t,1}\dots\boldsymbol{W}^{t,\ell-1}\boldsymbol{W}^{t,\ell}||^2\\
&\le||\Delta\boldsymbol{A}^{\ell}(\prod_{i=1}^{\ell-1}\hat{\boldsymbol{A}}^i)\boldsymbol{H}^0||^2\cdot||\prod_{i=1}^{\ell}\boldsymbol{W}^{t,i}||^2.
\end{aligned}$$

The factor $||\Delta\boldsymbol{A}^{\ell}(\prod_{i=1}^{\ell-1}\hat{\boldsymbol{A}}^i)\boldsymbol{H}^0||^2$ is fixed once the features and adjacency lists are determined. Next, $AD^{t,\ell}$ is proportional to the latter factor $||\prod_{i=1}^{\ell}\boldsymbol{W}^{t,i}||^2$, denoted as PW. A higher $AD^{t,\ell}$ indicates worse convergence. Thus, we reduce the error between $\boldsymbol{A}^{\ell}$ and $\hat{\boldsymbol{A}}^{\ell}$ to decrease $\Delta\boldsymbol{A}^{\ell}$ to improve the convergence when PW is high. To decrease $\Delta\boldsymbol{A}^{\ell}$, we compute a more accurate k-optimal neighbor set by reducing the re-computation interval m.

In each iteration, we calculate the value of PW and compare it with the previous iteration's value to determine whether we need to increase or decrease m. Algorithm 2 describes the process

Algorithm 2: Adaptive Sample Tuner

Input: weights $\boldsymbol{W}$, current iteration t
Output: sampling frequency m

1 **if** $t == 0$ **then**
    // initialize parameters of adaptive tuner
2 $m = \lfloor 2 \cdot (T_f + T_s + T_{ts})/T_{ts} \rfloor$
3 $m_{lb} = \lfloor 1 + T_s/(T_f - T_{ts}) \rfloor$
4 $PW = || \prod_{i=1}^{L} \boldsymbol{W}^{t,i} ||$
5 $t_{last} = 0$, $PW_{last} = PW$
6 **if** $t == t_{last} + m$ **then**
7 $PW = || \prod_{i=1}^{L} \boldsymbol{W}^{t,i} ||^2$
8 **if** $PW < PW_{last}$ **then**
9 $m = \lfloor m + 0.2m \rfloor$
10 **else**
11 **if** $(\, m - 0.2m \,) > m_{lb}$ **then** $m = \lfloor m - 0.2m \rfloor$ **else** $m = 1$
12 $t_{last} = t$, $PW_{last} = PW$

of tuning parameter m. A counter t keeps track of the number of iterations performed. When $t = 0$, we initialize m based on empirical observations (line 2), where $\lfloor \cdot \rfloor$ denotes the floor function. Here, T_f denotes the training time with all neighbors, T_s denotes the sampling time, and T_{ts} denotes the training time with sampled neighbors. We denote the lower bound of m by m_{lb} (line 3). Only when $m > m_{lb}$, does AD-Sampling outperform the training methods that use the full neighbors in terms of efficiency. The m_{lb} is calculated by $T_f = \frac{T_f + (T_s + T_{ts}) + (m_{lb} - 2) \cdot T_{ts}}{m_{lb}}$, i.e., $m_{lb} = 1 + \frac{T_s}{T_f - T_{ts}}$. After m iterations, we update PW and compare it with PW_{last} (lines 7–11). We increase m by $0.2m$ if $PW < PW_{last}$ (line 9). Otherwise, m is either decreased by $0.2m$ if $m - 0.2m > m_{lb}$ or set to 1 if $m - 0.2m \leq m_{lb}$ (line 11). The latter case implies that we do not perform sampling, as it does not enhance efficiency.

6.2 Communication Reduction

AD-Sampling samples the neighbors of vertices to construct a k-optimal neighbor set without distinguishing between local and remote neighbors (see Section 5). If a vertex samples a large number of remote neighbors, AD-Sampling alone may not be sufficient to improve performance in a distributed environment. To address this issue, we propose to selectively prune unimportant remote neighbors based on their *layer-wise AD-importance*.

Definition 6.2. Given a neighbor u of any target vertex v, the ℓ^{th} layer's **layer-wise *AD*-importance** of u in the t^{th} iteration, denoted as $\mathcal{I}^{t,\ell}(u)$, is the sum of *AD*s between u and its relevant aggregation:

$$\mathcal{I}^{t,\ell}(u) = \sum_{v \in \mathcal{V} \wedge u \in \mathcal{S}(v;k)} (AD^{t,\ell}_{\mathcal{S}(v;k)-\{u\}} - AD^{t,\ell}_{\mathcal{S}(v;k)}) \tag{7}$$

We observe that $\mathcal{I}^{t,\ell}(u)$ represents the negative effect on the entire convergence when u is missed. We aim to reduce the communication while minimizing the degradation of convergence and accuracy by employing layer-wise sampling based on the *AD*-importance metric.

According to Formula 7, no additional computation is required to obtain the *AD*-importance. After the remote neighbor sampling, each worker only requests N_c embeddings from the remote

neighbors that have the highest importance scores, rather than all remote neighbor embeddings, where N_c is the fanout of remote neighbors. This approach can effectively reduce communication costs.

Example 6.3. Fig. 5 provides an example of *AD*-importance based communication reduction, where $\{v_2, v_3, v_5\}$ is local node set and $\{v_1, v_4, v_6, v_7, v_8, v_9\}$ is remote node set. Consider remote neighbor v_4. We find that v_4 is involved in computing two local target vertices as a remote neighbor. We find that v_4 as a remote neighbor participates in the computing of two local target vertices v_2 and v_5. We can thus calculate $\mathcal{I}^{t,\ell}(v_4)$ using Formula 6.2. We keep neighbors with the top-N_c $\mathcal{I}^{t,\ell}$s, and thus the communication cost is reduced to N_c. The newly generated remote neighbor set will be used in training. The effect of neighbor u on vertex v is represented by $E^{t,\ell}_{u,v}$, which is defined as $E^{t,\ell}_{u,v} = AD^{t,\ell}_{\mathcal{S}(v;k)-\{u\}} - AD^{t,\ell}_{\mathcal{S}(v;k)}$.

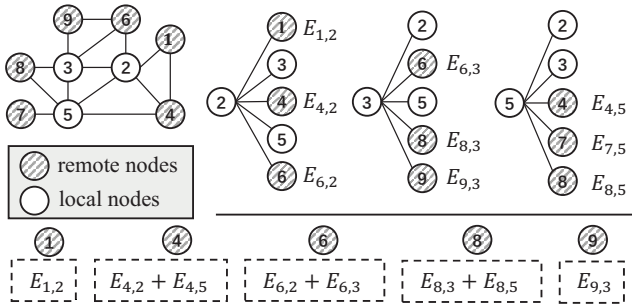


Fig. 5. *AD*-Importance-Based Communication Reduction

7 EXPERIMENTS

7.1 Experimental Setup

GNN Models and Datasets. We employ Graph Convolutional Networks (GCN) [9] and Graph Attention Networks (GAT) [33] in our experiments. However, it is worth noting that ADGNN is also applicable to other GNN models, as it can effectively reduce sampling costs and *AD*s. The default number of layers is set to 2. For GCN, the hidden layer size for Pubmed is set to 16, while it is set to 128 for the other four datasets. For GAT, we employ 8 attention heads with a hidden layer size of 8 for Pubmed and OGBN-Arxiv, and 1 attention head with a hidden layer size of 16 for Reddit-Small and OGBN-Papers100M. We set the dropout to 0.6. We use five real-world datasets from PyTorch Geometric [10] and Open Graph Benchmark [15]: Pubmed, OGBN-Arxiv, Reddit-Small, OGBN-Products, and OGBN-Papers. Pubmed, OGBN-Arxiv, and OGBN-Papers are citation networks that contain information about research papers and their citations. Reddit-Small is a dataset constructed from an online forum, and OGBN-Products is an Amazon product co-purchasing network. For brevity, we refer to OGBN-Arxiv, OGBN-Products, and OGBN-Papers without using the prefix OGBN, while we refer to Reddit-Small as Reddit. Table 3 provides statistics on the datasets, including the feature dimensionality (#Feat), the number of categories (#Class), and the ratio of the splits of the datasets into training/validation/testing sets (Train/Val/Test).
Environments. We conduct experiments on two clusters: **cluster-1:** a cluster consisting of 13 machines, each equipped with 32 GB DRAM and an Intel(R) Xeon(R) 4-core CPU E3-1226 v3 @ 3.30 GHz; **cluster-2:** a cluster consisting of 4 machines, each equipped with 250 GB DRAM, an Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz with 32 cores, and an NVIDIA RTX 2080Ti with 11GB memory. We employ six machines from cluster-1 for experiments, except for the layer-scalability

Table 3. Datasets (M=million and B=billion)

Datasets	$\|\mathcal{V}\|$	$\|\mathcal{E}\|$	#Feat	#Class	Train/Val/Test
Pubmed	0.02M	0.04M	500	3	0.65/0.10/0.25
Arxiv	0.16M	1.17M	128	40	0.54/0.18/0.28
Reddit	0.23M	57.3M	602	41	0.66/0.10/0.24
Products	2.44M	61.9M	100	47	0.08/0.02/0.90
Papers	0.11B	1.62B	200	172	0.78/0.08/0.14

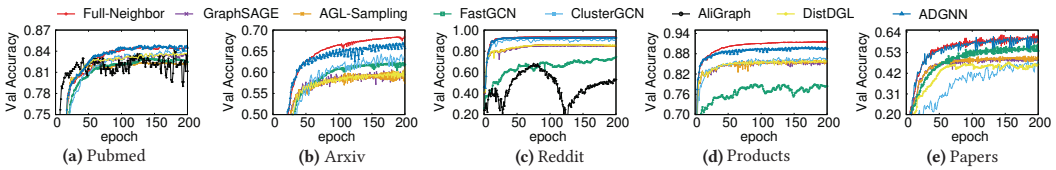


Fig. 6. Convergence Evaluation on Fanout = [1, 1]

experiments on Reddit and Products, which use 12 machines. We evaluate the results on Papers using cluster-2 and conduct tests on GPUs using the same cluster. The clusters employ 1Gbps and 10Gbps Ethernet connections, respectively. We deploy one worker and one server on a single machine.

Parameter Settings. We employ the *Adam* optimizer with a learning rate of 0.01 on Pubmed, Reddit, and Products and 0.05 on Arxiv and Papers. The sampling fanouts are set to [2, 2] on Pubmed and Arxiv and to [5, 5] on Reddit, Products, and Papers for the node-wise sampling techniques. We adjust fanouts of the layer-wise and subgraph-wise sampling techniques to ensure that they have numbers of edges that are similar to those of the node-wise sampling techniques. However, it is infeasible to set a similar edge number for BNS-GCN, as it samples only remote neighbors. Here, we set the fanout to 0. We set the number of AD computations to 2 and allow up to $3 \cdot k$ candidate-neighbors for all datasets. The communication pruning is set to 70%. We set m=10, 20, 65, 30, 25 for datasets repectively.

Baselines. We compare ADGNN with five state-of-the-art sampling techniques [4, 6, 14, 34, 46, 48] and two representative end-to-end systems [50, 52].

- **GraphSAGE [14]** is a fundamental GNN model that employs a random sampling approach.
- **AGL-Sampling [46]** samples neighbors only once in the beginning and reuses the sampled results in all subsequent iterations.
- **FastGCN [4]** interprets graph convolution as integral transforms of embedding functions and implements importance sampling using a Monte Carlo approach.
- **BNS-GCN [34]** uses a random sampling technique to select only remote vertices to decrease communication costs.
- **ClusterGCN [6]** partitions graphs into many subgraphs using METIS [19] and trains on only a subset of the subgraphs. Edges that cross between machines are omitted.
- **AliGraph [52]** reduces the communication between workers and graph servers by caching frequently accessed vertices.
- **DistDGL [50]** adapts GNN training by reducing the communication between machines and balancing the workload.

GraphSAGE [14] and AGL-Sampling [46] employ node-wise sampling. FastGCN [4] and BNS-GCN [34] employ layer-wise sampling, while ClusterGCN [6] employs subgraph-wise sampling.

Table 4. Evaluation of Epoch Time (sec) and Test Accuracy (%) on Low Sampling Fanouts

Method	Arxiv	Reddit	Products	Papers
AGL-Sampling	57.71, 0.176	85.33, 0.419	69.18, 0.474	48.85, 3.835
ADGNN+both	64.76, 0.236	90.89, 0.596	74.57, 0.675	56.43, 5.743

Since GraphSAGE [14], FastGCN [4], and ClusterGCN [6] are standalone techniques, we have adapted them to distributed environments. We disregard DGS [35], as it is not open-sourced and lacks details on the explainer implementation, making it difficult to re-engineer.

Training Mode. We perform experiments in full-batch and sampling modes for two main reasons. First, full-batch training typically leads to better convergence and accuracy compared to mini-batch training. Second, sampling helps alleviate computation bottlenecks. We provide evaluation results on CPU clusters and GPU clusters to demonstrate the applicability and superiority. The codes are available online[1].

7.2 Evaluation on Convergence of Sampling Techniques with Low Fanouts

We note that ADGNN can achieve high accuracy even with extremely low sampling rates. For the node-wise sampling techniques, we set the fanout of all layers to 1. For the layer-wise and subgraph-wise techniques, we ensure numbers of edges in the sampled graphs that are similar to those of the node-wise techniques. We exclude BSN-GCN because it cannot maintain numbers of edges in the sampled graphs similar to those of the node-wise techniques.

Fig. 6 reports the accuracy variation over iterations, showing that ADGNN outperforms all baselines and achieves accuracies close to those of full-neighbor training even at a fanout of [1, 1], while the other techniques suffer from significant degradation. This is because we select the optimal neighbor set for each vertex. We observe the following: (i) GraphSAGE and DistDGL have similar convergence since both use random sampling; (ii) AGL-Sampling is more volatile and results in lower accuracy, as it only samples once and may have a larger distribution error; (iii) layer-wise and subgraph-wise sampling (FastGCN and ClusterGCN) may sample fewer neighbors than node-wise sampling while having the same number of edges, resulting in unstable convergence; (iiii) AliGraph uses asynchronous training and usually produces poor results. We do not include AliGraph in efficiency comparisons as it consistently generates extremely low accuracies. Moreover, AGL-Sampling has the potential to lead to permanent information loss and the introduction of irreparable errors. This is why GNN training tends to avoid this mode. Additionally, at lower sampling fanouts, ADGNN clearly outperforms AGL-Sampling in terms of accuracy. At higher sampling fanouts, ADGNN can achieve greater efficiency than AGL-Sampling since it does not take into account communication cost.

7.3 Evaluation on Test Accuracy and Efficiency

To assess the adaptive capability of ADGNN, we determine the test accuracy and epoch time when varying the number of layers. We exclude the results of Full-Neighbor and AGL-Sampling training for two reasons: (i) Full-Neighbor training always achieves the best accuracy but incurs high training costs; (ii) AGL-Sampling may achieve fast training, but it suffers from low accuracy due to permanent information loss. In Tables 5 and 6, ***bold*** represents best results, while *underline* indicates second-best results.

We see that ADGNN achieves the best accuracy and epoch time across most datasets and numbers of GNN layers. Although it also achieves second-best accuracy, e.g., on Reddit (93.47% for ADGNN

[1]https://github.com/songzhen-neu/ADGNN

Table 5. Test Accuracy (%) and Epoch Time (sec) (***bold*** denotes the best, and *underline* represents the second-best)

Method	Pubmed			Arxiv		
	2-layer	3-layer	4-layer	2-layer	3-layer	4-layer
GraphSAGE	83.89, 0.0935	83.02, 0.1432	83.08, 0.1930	66.91, 0.6995	62.02, 1.4350	50.95, 2.1969
FastGCN	80.75, 0.0913	73.41, 0.1362	78.19, 0.1787	51.38, 0.5450	45.57, 0.8738	39.49, 1.1834
ClusterGCN	84.16, 0.0904	82.41, 0.1327	**83.83**, 0.1757	66.12, 0.5351	58.66, 0.8003	**58.65**, **1.1137**
BNS-GCN	83.37, 0.0746	81.20, 0.1378	83.27, 0.1823	67.46, 0.4364	58.65, 0.9805	49.76, 1.3985
DistDGL	83.89, 0.1258	63.89, 0.3289	43.73, 0.3897	66.20, 0.4294	61.00, 0.7622	37.37, 1.1408
ADGNN (ours)	**84.24**, **0.0620**	**84.37**, **0.0896**	83.31, **0.1271**	**68.24**, **0.3224**	**62.52**, **0.7586**	51.55, 1.5771
Full-Neighbor	84.42, 0.0764	82.58, 0.0899	83.91, 0.1310	69.00, 0.5666	62.79, 1.8593	60.48, 3.1694
AGL-Sampling	83.49, 0.0450	81.58, 0.0704	82.35, 0.0957	66.96, 0.2443	61.47, 0.7096	16.13, 1.1845

Table 6. Test Accuracy (%) and Epoch Time (sec) for Reddit, Products and Papers

Method	Reddit			Products		Papers	
	2-layer	3-layer	4-layer	2-layer	3-layer	2-layer	3-layer
GraphSAGE	93.01, 6.2398	92.87, 8.6044	92.48, 11.622	75.87, 7.6784	76.96, 11.528	56.23, 46.498	52.17, 108.50
FastGCN	73.50, 7.7835	50.30, 5.8656	50.30, 5.8656	65.36, 8.0162	76.45, 12.123	56.21, 37.536	51.16, 75.867
ClusterGCN	87.23, 3.3815	77.69, 3.4550	79.06, **2.6575**	67.39, 2.9256	67.52, **2.8473**	51.16, 62.426	39.66, 114.59
BNS-GCN	**93.74**, 7.0609	**93.35**, 5.2432	**92.90**, 7.1006	75.97, 12.511	75.78, 25.541	58.04, 32.148	53.17, 84.174
DistDGL	92.86, 4.0117	92.94, 5.3797	92.74, 6.6590	76.06, 2.8444	**77.85**, 5.4586	44.25, 18.541	28.61, 34.254
ADGNN (ours)	93.47, **0.9630**	93.19, **2.5673**	92.84, 3.7239	**77.12**, **1.8634**	76.43, 4.9543	**58.59**, **14.578**	**53.26**, **26.747**
Full-Neighbor	93.80, 5.5731	93.48, 7.4064	93.34, 11.079	74.69, 11.380	75.87, 33.270	58.14, 23.715	53.30, 106.71
AGL-Sampling	93.02, 0.8131	92.16, 2.1150	92.62, 3.2031	75.45, 1.2704	75.59, 4.3470	58.03, 10.684	52.92, 23.238

vs. 93.74% for BNS-GCN), ADGNN achieves a 7.4× speedup over BNS-GCN. Next, we offer a detailed analysis. FastGCN typically yields lower accuracy than the other sampling techniques when using the same number of edges. This is likely due to FastGCN's reliance on degree importance as the basis for sampling. High-degree vertices are more likely to be selected, resulting in less diverse sampling. We find that other layer-wise and subgraph-wise techniques also tend to suffer from this limitation because they need to retain all incident edges in the sampled graphs. Despite setting the sampling fanout to 0, BNS-GCN still has a large number of edges on Reddit, which is likely a contributing factor to its higher accuracy on this dataset. GraphSAGE maintains an unbiased distribution of the initial graphs, but at the expense of requiring sampling for each iteration. ClusterGCN can achieve good performance with a four-layer GNN on graphs with a small diameter, as it constrains the neighbor set sampled from a local domain.

We provide the speedup findings in Fig. 7. The speedup metric SP is calculated as $SP = T_o/T_n$, where T_o denotes the execution time of old techniques and T_n depicts the execution time of the new improved techniques. ADGNN achieves 1.20–1.51×, 1.35–2.17×, 3.51–8.08×, 1.57–6.71×, and 2.21–4.28× speedups over the existing sampling techniques on Pubmed, Arxiv, Reddit, Products, and Papers, respectively. DistDGL employs a different system framework and different engineering implementations than we do. We conduct an end-to-end comparison with DistDGL. ADGNN achieves 2.03×, 1.33×, 4.17×, 1.53× and 1.27× speedups over DistDGL. BNS-GCN exhibits poor efficiency on Reddit and Products due to their high edge densities. ClusterGCN achieves great performance by retaining only some subgraphs and dropping all remote edges. However, ClusterGCN achieves a sub-optimal accuracy of 87.23% vs. 93.47% on Reddit and 67.39% vs. 77.12% on Products compared to ADGNN.

For clarity, we present the number of edges and vertices for each of the sampling techniques in Fig. 8, where Edges-i denotes the number of edges in the i^{th} layer, RmtNodes-i denotes remote

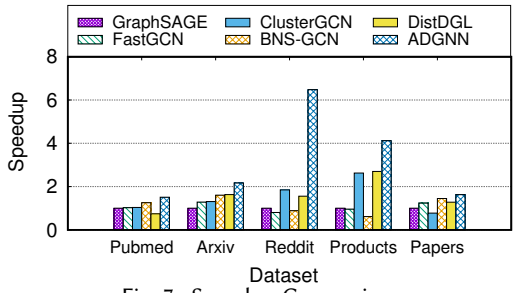


Fig. 7. Speedup Comparison

neighbors, and Nodes-i denotes the total number of neighbors. We see that ADGNN has slightly more edges than GraphSAGE. This is due to the proposed optimization strategy (see Section 5.3), where all neighbors are retained if fanout $k \geq 0.8 \times |\mathcal{N}(v)|$, thereby reducing the computational load of *AD*. Although BNS-GCN achieves the highest accuracy on Reddit, the number of edges in BNS-GCN is nearly one order of magnitude higher than in ADGNN.

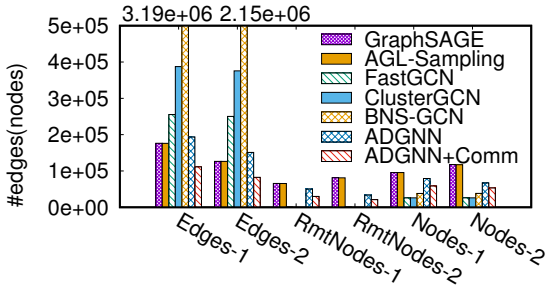


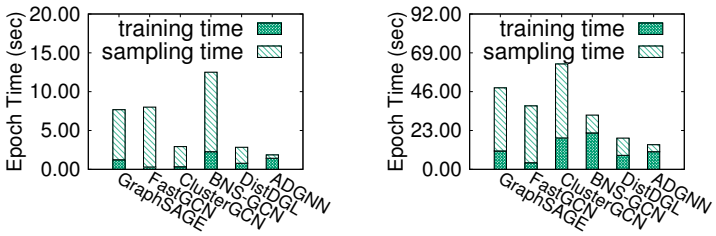
Fig. 8. Edges and Nodes on Reddit

7.4 Components of Epoch Time and Total Time

We report the training and sampling time durations per epoch in Fig. 9. The sampling time of each technique is several times more than the training time. ADGNN reduces the sampling time substantially since we amortize this across multiple epochs. The layer-wise FastGCN and the subgraph-wise ClusterGCN incur less training time than the node-wise techniques. This is because fewer vertices are involved in these methods when they perform neural network operations. Additionally, FastGCN incurs higher sampling time due to the masking operation after sampling vertices for layers. On CPU clusters, the masking operation is more significant due to the small number of cores. DistDGL incurs less training time since different frameworks are used and some system optimizations are tailored to DistDGL. However, we achieve better efficiency and accuracy than DistDGL in an end-to-end comparison.

We measure the total execution time for each operation across 200 iterations and we use cluster-2 for experiments on both CPUs and GPUs. The columns in Table 7 represent the following: tensor operation, communication, embedding organization, CPU-GPU transfer, parameter update, and subgraph construction time.

Distributed GNN training necessitates distributed subgraph construction for sampling, which involves tasks such as generating target vertices, establishing routing tables, and constructing initial feature matrices. These operations cannot be easily executed with tensor operations, making GPU execution unsuitable. As observed in *Table 6*, subgraph construction is the most time-consuming operation, regardless of whether training is performed on CPUs or GPUs. Specifically, compared to



(a) Products (b) Papers

Fig. 9. Components of Epoch Time

Table 7. Detailed Total Times (sec) for Components on Arxiv

Method	*tens_op*	*comm*	*emb_org*	*trans*	*upda*	*subg_cons*
Full-Neighbor-CPU	62.10	30.28	62.68	-	8.111	-
BNS-GCN-CPU	34.96	3.750	21.78	-	6.960	349.9
ADGNN-CPU	33.35	15.47	21.08	-	8.315	39.05
ADGNN+both-CPU	26.14	11.21	17.22	-	8.915	21.91
Full-Neighbor-GPU	2.512	31.45	61.78	7.902	9.740	-
BNS-GCN-GPU	1.603	4.942	22.10	2.786	8.326	348.5
ADGNN-GPU	1.601	16.52	21.74	3.841	8.967	39.13
ADGNN+both-GPU	1.436	12.43	16.54	2.987	9.181	19.99

BNS-GCN, ADGNN significantly reduces the time required for distributed subgraph construction from 348.5 seconds to 39.13 seconds. Besides, sampling local neighbors can effectively reduce the number of vertices, resulting in lower costs of distributed embedding organization and distributed subgraph construction.

7.5 Evaluation for GAT

We present the results for GAT in Table 8 to assess the applicability of ADGNN. We set the fanouts of each layer to 2, 8, 80, and 20 for the datasets. ADGNN outperforms all the comparative methods in terms of accuracy and efficiency. Specifically, (i) compared to ClusterGCN, which achieves the second-best training time on Reddit, ADGNN demonstrates a 2.45× speedup and a 4.73% accuracy improvement; and (ii) compared to GraphSAGE, which attains the highest accuracy among all baselines on Reddit, ADGNN achieves a 3.10× speedup and a 0.32% accuracy improvement. AGL-Sampling shows lower accuracy compared to other node-wise sampling techniques (i.e., GraphSAGE, ADGNN) due to the loss of information caused by sampling only once. Next, layer-wise and subgraph-wise sampling methods (i.e., FastGCN and ClusterGCN) exhibit more significant accuracy reduction, because they typically involve fewer vertices when they have the same number of edges as node-wise sampling methods. Finally, ADGNN with both optimizations achieves comparable accuracy to ADGNN while significantly enhancing efficiency.

7.6 Ablation Experiments and Evaluation on Training with GPUs

We conduct ablation experiments to investigate the impact of two optimizations: the adaptive sampling frequency tuner and the *AD*-importance based communication reduction. We compare four methods for each dataset in Table 9, where *ad* denotes the AD-Sampling approach defined in Section 5, *ad* + *adapt* denotes *ad* with the adaptive sampling frequency tuner, *ad* + *comm*

Table 8. Evaluation of Epoch Time (sec) and Accuracy (%) when Extending the Existing Sampling Techniques to GAT

Method	Pubmed	Arxiv	Reddit	Papers
GraphSAGE	0.256, 82.33	2.057, 61.90	17.30, 89.90	117.7, 39.75
FastGCN	0.248, 82.62	1.887, 24.39	19.45, 77.30	83.64, 33.93
ClusterGCN	0.222, 83.65	1.866, 20.77	13.71, 85.49	132.7, 34.91
BNS-GCN	0.237, 83.43	2.274, 59.54	13.75, 88.42	106.4, 40.17
ADGNN	0.215, **84.04**	1.512, **63.00**	5.586, **90.22**	48.68, **41.39**
ADGNN+both	**0.195**, 83.86	**1.127**, 62.78	**3.862**, 90.16	**34.74**,40.68
Full-Neighbor	0.300, 84.48	2.496, 63.74	23.37, 90.66	63.82, 42.06
AGL-Sampling	0.200, 82.25	1.194, 61.69	3.930, 89.71	32.98, 39.48

Table 9. Ablation Study Evaluating Epoch Time (sec) and Accuracy (%)

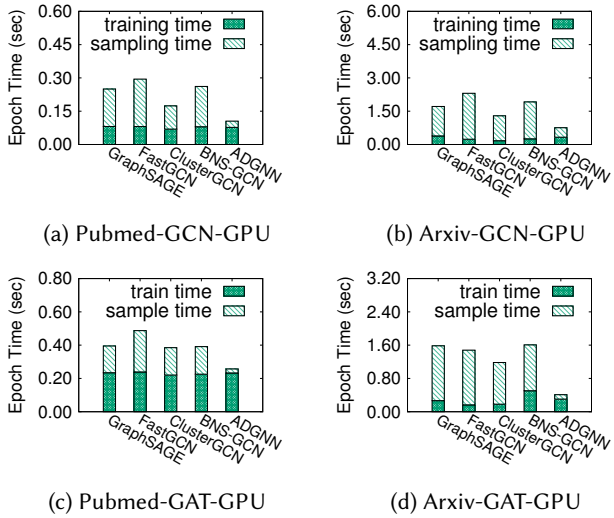
Method	*ad*	*ad+adpt*	*ad+comm*	*ad+both*
Pubmed	0.062, 84.24	0.066, 84.40	0.057, 84.14	0.062, 84.24
Arxiv	0.322, 68.24	0.428, 68.27	0.286, 66.34	0.286, 66.73
Reddit	0.963, 93.47	0.953, 93.60	0.796, 92.85	0.794, 92.97
Products	1.863, 77.12	1.720, 77.00	1.421, 75.53	1.415, 75.60
Papers	14.58, 58.59	14.21, 58.76	11.30, 58.37	10.58, 58.61

means *ad* with *AD*-importance based communication reduction, and *ad + both* includes both optimizations. The results show that the adaptive sampling frequency tuner can improve the accuracy of both *ad* and *ad + comm*, and it can even improve the accuracy and reduce the training time on Reddit by controlling convergence fluctuations. Communication reduction can cut the training cost without significantly degrading accuracy. Even under low-sampling fanouts, the proportion of the communication cost remains high, particularly for large datasets. For example, when applied to Papers, communication reduction alone results in a performance improvement of 29% (from 14.58s to 11.30s), with only a slight accuracy degradation from 58.59% to 58.37%. Finally, *ad + both* achieves similar accuracy to GraphSAGE while providing higher speedups. Overall, ADGNN outperforms the state-of-the-art distributed sampling technique BNS-GCN by a speedup of 1.20–8.89× and the sampling-based distributed GNN system DistDGL by a speedup of 1.50–5.05×.

We conduct experiments to assess performance on GPU clusters. As previously mentioned, we use full-batch training for comparison, and we present results only for Pubmed and Arxiv, which can be accommodated in GPU memory. As shown in Fig. 10, training GCN models on GPUs can significantly reduce training time and achieve higher speedups, improving from 1.16–1.65× to 1.65–2.79× on Pubmed, and from 1.19–1.81× to 1.71–3.05× on Arxiv. We provide results of training GAT on GPUs. ADGNN achieves speedups of 1.50-1.89× and 2.89-3.93× over the other methods on Pubmed and Arxiv, respectively. Regardless of the underlying hardware architecture (GPU or CPU clusters), ADGNN effectively mitigates the high-cost bottleneck of sampling and achieves high accuracy.

7.7 Test on Various Fanouts and Adaptive Sampling Frequency Tuner

We evaluate ADGNN, GraphSAGE, BNS-GCN, and ADGNNR on Arxiv and Products, where ADGNNR is a variant of ADGNN that retains all local neighbors. The sampling rates range from



(a) Pubmed-GCN-GPU (b) Arxiv-GCN-GPU

(c) Pubmed-GAT-GPU (d) Arxiv-GAT-GPU

Fig. 10. Evaluation on GPUs

Table 10. Accuracy (%) Evaluation with Varying Fanouts

Sampling Rate	$\frac{1}{g}\times 100\%$	10%	20%	30%	40%
GraphSAGE-Arxiv	59.23	66.91	67.93	68.27	**69.01**
BNS-GCN-Arxiv	N/A	N/A	N/A	67.71	68.41
ADGNNR-Arxiv	N/A	N/A	N/A	66.63	68.47
ADGNN-Arxiv	**67.81**	**68.21**	**68.59**	**68.52**	68.99
GraphSAGE-Products	70.12	75.87	**75.91**	75.35	75.67
BNS-GCN-Products	N/A	N/A	N/A	N/A	75.20
ADGNNR-Products	N/A	N/A	N/A	N/A	75.34
ADGNN-Products	**75.85**	**77.12**	75.82	**75.60**	**75.70**

($\frac{1}{g} \times 100$)% to 40%, where ($\frac{1}{g} \times 100$)% represents the case of fanout=1. Here, N/A indicates that BNS-GCN and ADGNNR cannot achieve such low sampling rates because they retain all local neighbors. By comparing with BNS-GCN and ADGNNR, we can observe the effect of sampling local vertices on accuracy. As shown in Table 10, ADGNN consistently achieves the highest accuracy across nearly all sampling rates. We observe that retaining all local vertices in ADGNNR leads to a decline in accuracy. This is attributed to the inability to guarantee the minimal aggregation difference without sampling local vertices. Notably, on Arxiv, ADGNN with a 20% sampling rate achieves higher accuracy than both ADGNNR and BNS-GCN (preserving all local neighbors) at 40%. On Products, the highest accuracy is achieved at a 10% sampling rate. This is because retaining more neighbors can lead to overfitting on the training set. Even in such circumstances, ADGNN still achieves the highest accuracy. ADGNN also effectively mitigates overfitting at low sampling rates.

In practice, we cannot select the optimal value for m manually, and nor can we predefine an effective default value because the best value depends on the data and runtime AD. Hence, automatic adjustment of m without any prior knowledge becomes necessary. Moreover, we perform experiments to investigate the effects of adaptively adjusting parameter m. Table 11 shows the results. A small value of m ensures higher accuracy but increases execution time. Although a large

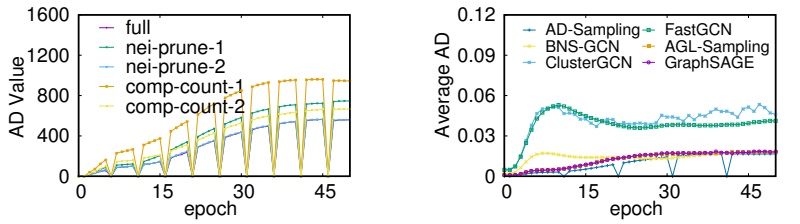
Table 11. Evaluation of the Adaptive Tuner on Reddit

Method	*m*=5	*m*=45	*m*=85	*m*=125	*m*=165	*adapt*
Time (sec)	3.652	1.173	0.928	0.902	0.904	0.953
Accuracy (%)	93.45	93.38	93.38	93.36	93.31	93.60

value of m allows for improved accuracy by selecting the right recomputation timing, determining the optimal value of m before training is not feasible. By dynamically adjusting m, the re-sampling timing can be optimized for simultaneous improvements in accuracy and efficiency.

7.8 Evaluation on *AD* and Hyperparameter

We present the results of our experiments with various hyperparameter settings to evaluate their influence on Pubmed and Arxiv in Fig 11a. Specifically, we investigate the effect of different hyperparameters on the *AD* values, where *nei-prune-i* refers to computing *AD* on a subset of neighbors with a size of $i \times k$, and *comp-count-i* denotes the number of *AD* calculations executed on each layer. As shown in Fig 11, *nei-prune-2* achieves similar *AD* values to the approach without approximation, and *comp-count-2* slightly degrades the performance. Based on these results, we conclude that setting the value of i to 2 achieves both high accuracy and efficiency. Note that a worse *AD* can be obtained when using a low value of i, i.e., *nei-prune-1* and *comp-count-1* generate the worst results. Since satisfactory results are already achieved when $i = 2$, we only demonstrate the results up to $i = 2$ to avoid overlapping lines in the experiment.



(a) Hyperparameter Sensitivity (b) Aggregation Difference Comparison

Fig. 11. Hyperparameter and AD Comparisons on Arxiv

We report the average value of *AD* for all training vertices in Fig. 11b. As can be seen, ADGNN produces the smallest *AD* among all the sampling techniques due to the proposed aggregation-difference aware sampling technique. AGL-Sampling and GraphSAGE always generate the same *AD*, since both use a random sampling approach. However, GraphSAGE usually achieves better convergence, as it involves more vertices, which leads to better distribution fitting. In contrast, AGL-Sampling, due to its offline sampling strategy, results in information loss in the graph, which leads to a degradation in accuracy. Additionally, BNS-GCN produces a poor *AD* on Pubmed and a good *AD* on Arxiv, which is consistent with the accuracy results presented in Table 5. FastGCN involves fewer vertices due to its degree-importance sampling strategies and, as a result, suffers from worse convergence.

8 CONCLUSION

Distributed training of GNNs is hampered by substantial computation and communication bottlenecks. Neighbor-Sampling has been proposed as an efficient way to decrease training costs. However, the existing sampling techniques and sampling-based distributed systems face challenges of high costs incurred by distributed sampling (e.g., distributed subgraph construction), accuracy degradation with a small fanout, and poor scalability when using GPU training. We propose

a sampling-based system for GNN training called ADGNN that utilizes a hybrid online-offline sampling architecture and employs a novel distributed sampling technique named Aggregation-Difference Aware Sampling (AD-Sampling). This technique achieves high accuracy even with a small sampling fanout. To reduce network communication costs, we present a layer-wise *AD*-importance based communication reduction technique for remote vertices. Our experiments offer evidence that ADGNN can outperform existing sampling techniques and systems by up to nearly a factor of 9 for distributed GNN training on both CPU and GPU clusters.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (62072083, U2241212 and 62072082) and the Fundamental Research Funds for the Central Universities (N2216017).

REFERENCES

[1] Jiyang Bai, Yuxiang Ren, and Jiawei Zhang. 2021. Ripple Walk Training: A Subgraph-based Training Framework for Large and Deep Graph Neural Network. In *IJCNN*. 1–8.
[2] Muhammed Fatih Balin and Ümit V. Çatalyürek. 2022. (LA)yer-neigh(BOR) Sampling: Defusing Neighborhood Explosion in GNNs. *CoRR* abs/2210.13339 (2022).
[3] Huiyuan Chen, Chin-Chia Michael Yeh, Fei Wang, and Hao Yang. 2022. Graph Neural Transport Networks with Non-local Attentions for Recommender Systems. In *WWW*. 1955–1964.
[4] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *ICLR*.
[5] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *ICML*. 941–949.
[6] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *KDD*. 257–266.
[7] Weilin Cong, Rana Forsati, Mahmut T. Kandemir, and Mehrdad Mahdavi. 2020. Minimal Variance Sampling with Provable Guarantees for Fast Training of Graph Neural Networks. In *KDD*. 1393–1403.
[8] Hanjun Dai, Zornitsa Kozareva, Bo Dai, Alexander J. Smola, and Le Song. 2018. Learning Steady-States of Iterative Algorithms over Graphs. In *ICML*, Vol. 80. 1114–1122.
[9] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *NeurIPS*. 3837–3845.
[10] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. *CoRR* abs/1903.02428 (2019).
[11] Chen Gao, Xiang Wang, Xiangnan He, and Yong Li. 2022. Graph Neural Networks for Recommender System. In *WSDM*. 1623–1625.
[12] Yunjun Gao, Xiaoze Liu, Junyang Wu, Tianyi Li, Pengfei Wang, and Lu Chen. 2022. ClusterEA: Scalable Entity Alignment with Stochastic Training and Normalized Mini-batch Similarities. In *KDD*. 421–431.
[13] Yu Gu, Kaiqiang Yu, Zhen Song, Jianzhong Qi, Zhigang Wang, Ge Yu, and Rui Zhang. 2022. Distributed Hypergraph Processing Using Intersection Graphs. *IEEE Trans. Knowl. Data Eng.* 34, 7 (2022), 3182–3195.
[14] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NeurIPS*. 1024–1034.
[15] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *NeurIPS*, Vol. 33. 22118–22133.
[16] Wen-bing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive Sampling Towards Fast Graph Representation Learning. In *NeurIPS*. 4563–4572.
[17] Zijian Huang, Meng-Fen Chiang, and Wang-Chien Lee. 2022. LinE: Logical Query Reasoning over Hierarchical Knowledge Graphs. In *KDD*. 615–625.
[18] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* 2, 187–198.
[19] George Karypis and Vipin Kumar. 1998. Multilevel k-way Partitioning Scheme for Irregular Graphs. *J. Parallel Distributed Comput.* (1998), 96–129.
[20] Yiming Li, Yanyan Shen, Lei Chen, and Mingxuan Yuan. 2023. Orca: Scalable Temporal Graph Neural Network Training with Theoretical Guarantees. *Proc. ACM Manag. Data* 1, 1 (2023), 52:1–52:27.
[21] Xiaoze Liu, Junyang Wu, Tianyi Li, Lu Chen, and Yunjun Gao. 2023. Unsupervised Entity Alignment for Temporal Knowledge Graphs. In *WWW*. 2528–2538.

[22] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *ATC*. 443–458.
[23] Qianwen Ma, Chunyuan Yuan, Wei Zhou, and Songlin Hu. 2021. Label-Specific Dual Graph Neural Network for Multi-Label Text Classification. In *ACL/IJCNLP*. 3855–3864.
[24] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj D. Kalamkar, Nesreen K. Ahmed, and Sasikanth Avancha. 2021. DistGNN: scalable distributed training for large-scale graph neural networks. In *SC*. 76:1–76:14.
[25] Yeonhong Park, Sunhong Min, and Jae W. Lee. 2022. Ginex: SSD-enabled Billion-scale Graph Neural Network Training on a Single Machine via Provably Optimal In-memory Caching. *Proc. VLDB Endow.* 15, 11 (2022), 2626–2639.
[26] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. SANCUS: Staleness-Aware Communication-Avoiding Full-Graph Decentralized Training in Large-Scale Graph Neural Networks. *Proc. VLDB Endow.* 15, 9 (2022), 1937–1950.
[27] Jingshu Peng, Yanyan Shen, and Lei Chen. 2021. GraphANGEL: Adaptive aNd Structure-Aware Sampling on Graph NEuraL Networks. In *ICDM*. 479–488.
[28] Zhihao Shi, Xize Liang, and Jie Wang. 2023. LMC: Fast Training of GNNs via Subgraph Sampling with Provable Convergence. *arXiv preprint arXiv:2302.00924* (2023).
[29] Zhen Song, Yu Gu, Jianzhong Qi, Zhigang Wang, and Ge Yu. 2022. EC-Graph: A Distributed Graph Neural Network System with Error-Compensated Compression. In *ICDE 2022*. 648–660.
[30] Zhen Song, Yu Gu, Zhigang Wang, and Ge Yu. 2022. DRPS: efficient disk-resident parameter servers for distributed machine learning. *Frontiers Comput. Sci.* 16, 4 (2022), 164321.
[31] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *OSDI*. 495–514.
[32] Alok Tripathy, Katherine A. Yelick, and Aydin Buluç. 2020. Reducing communication in graph neural network training. In *SC*. 1–14.
[33] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903*.
[34] Cheng Wan, Youjie Li, Ang Li, Nam Sung Kim, and Yingyan Lin. 2022. BNS-GCN: Efficient Full-Graph Training of Graph Convolutional Networks with Partition-Parallelism and Random Boundary Node Sampling. In *MLSys*. 673–693.
[35] Xinchen Wan, Kai Chen, and Yiming Zhang. 2022. DGS: Communication-Efficient Graph Sampling for Distributed GNN Training. In *ICNP*. 1–11.
[36] Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. 2023. Scalable and Efficient Full-Graph GNN Training for Large Graphs. *Proc. ACM Manag. Data* 1, 2 (2023), 143:1–143:23.
[37] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. NeutronStar: Distributed GNN Training with Hybrid Dependency Management. In *SIGMOD*. 1301–1315.
[38] Zhigang Wang, Yu Gu, Yubin Bao, Ge Yu, Jeffrey Xu Yu, and Zhiqiang Wei. 2021. HGraph: I/O-Efficient Distributed and Iterative Graph Computing by Hybrid Pushing/Pulling. *IEEE Trans. Knowl. Data Eng.* 33, 5 (2021), 1973–1987.
[39] Junyang Wu, Tianyi Li, Lu Chen, Yunjun Gao, and Ziheng Wei. 2023. SEA: A Scalable Entity Alignment System. In *SIGIR*. 3175–3179.
[40] Yirong Wu, Jialin Chen, and Tinglong Tang. 2022. Feature Enhanced Graph Neural Network for Few-Shot Image Classification. In *CSCWD*. 513–518.
[41] Yifan Xing, Tong He, Tianjun Xiao, Yongxin Wang, Yuanjun Xiong, Wei Xia, David Wipf, Zheng Zhang, and Stefano Soatto. 2021. Learning Hierarchical Graph Neural Networks for Image Clustering. In *ICCV*. 3447–3457.
[42] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*.
[43] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *KDD*. 974–983.
[44] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*.
[45] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2019. Accurate, Efficient and Scalable Graph Embedding. In *IPDPS*. 462–471.
[46] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. 2020. AGL: A Scalable System for Industrial-purpose Graph Machine Learning. *PVLDB* 13, 12 (2020), 3125–3137.
[47] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *NeurIPS*. 5171–5181.
[48] Xin Zhang, Yanyan Shen, and Lei Chen. 2022. Feature-Oriented Sampling for Fast and Scalable GNN Training. In *ICDM*. 723–732.

[49] Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. 2023. DUCATI: A Dual-Cache Training System for Graph Neural Networks on Giant Graphs with the GPU. *Proc. ACM Manag. Data* 1, 2 (2023), 166:1–166:24.
[50] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *SC*. 36–44.
[51] Jason Zhu, Yanling Cui, Yuming Liu, Hao Sun, Xue Li, Markus Pelger, Tianqi Yang, Liangjie Zhang, Ruofei Zhang, and Huasha Zhao. 2021. TextGNN: Improving Text Encoder via Graph Neural Network in Sponsored Search. In *WWW*. 2848–2857.
[52] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *PVLDB* 12 (2019), 2094–2105.
[53] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. 2019. Layer-Dependent Importance Sampling for Training Deep and Large Graph Convolutional Networks. In *NeurIPS*. 11247–11256.

Received April 2023; revised July 2023; accepted August 2023