Faster Convergence in Mini-batch Graph Neural Networks Training with Pseudo Full Neighborhood Compensation

Qiqi Zhou
Department of CSE,HKUST
qzhouam@connect.ust.hk

Yanyan Shen*
Department of CSE,Shanghai Jiao Tong University
shenyy@sjtu.edu.cn

Lei Chen
Department of CSE,HKUST
DSA Thrust, HKUST (GZ)
leichen@cse.ust.hk

ABSTRACT

Graph Neural Networks (GNNs) have achieved remarkable success in various graph-related tasks. However, training GNNs on large-scale graphs is hindered by the neighbor explosion problem, rendering full-batch training computationally infeasible. Mini-batch training with neighbor sampling is a widely adopted solution, but it introduces gradient estimation errors that slow convergence and reduce model accuracy. In this work, we identify two primary sources of these errors: (1) missing gradient contributions from unsampled target nodes, and (2) inaccuracies in messages computed from sampled nodes. While existing methods largely focus on mitigating the second source, they often overlook the first, resulting in incomplete gradient estimation. To address this gap, we propose the **Pseudo Full Neighborhood Compensation** (PFNC) framework, which leverages historical information to simultaneously compensate for both errors. PFNC is designed to integrate seamlessly with any neighbor sampling technique and significantly lowers memory demands by maintaining only a partial cache of historical embeddings and gradients. Theoretical analysis demonstrates that PFNC provides a closer approximation to the ideal gradient, enhancing convergence. Extensive experiments across multiple benchmark datasets confirm that PFNC accelerates convergence and improves generalization across diverse neighbor sampling strategies.

PVLDB Reference Format:
Qiqi Zhou, Yanyan Shen, and Lei Chen. Faster Convergence in Mini-batch Graph Neural Networks Training with Pseudo Full Neighborhood Compensation. PVLDB, 18(11): 4309 - 4322, 2025.
doi:10.14778/3749646.3749695

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/qzhou77/PFNC-artifact.

1 INTRODUCTION

Graph Neural Networks (GNNs) have achieved great success in many graph-related applications, such as recommendation systems [9], fraud detection [33], biochemistry [11] and combinatorial optimization [21]. These real-world applications can involve large-scale graphs with billions of edges [15].

Training GNNs on large-scale graphs necessitates the use of mini-batch training due to the limited memory capacity of modern GPUs, which cannot accommodate the entire graph. The fundamental idea of GNNs is leveraging a message-passing mechanism to iteratively aggregate information from neighboring nodes. An L-layer GNN is used to obtain information from neighbors within L hops, where the representation (embedding) of a node in the current layer is computed by aggregating the representations of all its neighbors from the previous layer. However, even when only a small mini-batch of labeled nodes is sampled, the iterative dependency on neighbors grows exponentially with L. If exact mini-batch training is employed—i.e., retaining the full set of neighbors within L-hops for the sampled labeled nodes—the process remains computationally expensive and memory-intensive. This is known as the *neighbor explosion* problem [14, 44] and poses a significant challenge for training GNNs on large-scale graphs.

To tackle the *neighbor explosion* problem, several neighbor sampling methods have been developed, including node-wise sampling [14, 37], layer-wise sampling [5, 19, 46], and subgraph sampling [7, 40]. These techniques aim to construct mini-batches by sampling a subset of neighbors for each node involved in the forward propagation of each layer. During the corresponding backward propagation, the model parameters' gradients are computed. Notably, as the number of sampling layers increases, all neighbors of unsampled nodes at layer l are not sampled at layer $l-1$.

Although these sampling methods enable scalable GNN training on large graphs, the growing scale of real-world applications has made training efficiency a critical concern. This issue has received increasing attention in the data management community, where graph data management and mining are long-standing and important research topics. Recent studies in this area predominantly focus on system-level optimizations—for example, reducing the overhead of CPU-GPU or remote data transfers [3, 4, 13, 20, 32, 36, 39, 41, 42], improving sampling efficiency [12, 29], or applying advanced parallelization techniques [1, 22, 30, 35]. In contrast, our work addresses this challenge from a complementary algorithmic perspective. Specifically, we observe that mini-batch training introduces significant gradient estimation errors due to neighbor sampling. It omits the contributions from unsampled nodes during both the forward and backward passes of each layer, in contrast to exact mini-batch training. This omission introduces error into gradient estimation, negatively impacting the convergence speed and generalization performance of GNNs [6, 8].

Figure 1 provides an empirical example illustrating how the error affect convergence. As detailed in Eq. 6 of Section 3.1, we decompose the error into two sources: ① *the inaccuracies in individual message item from each target node* and ② *the missing item in gradient*

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749695

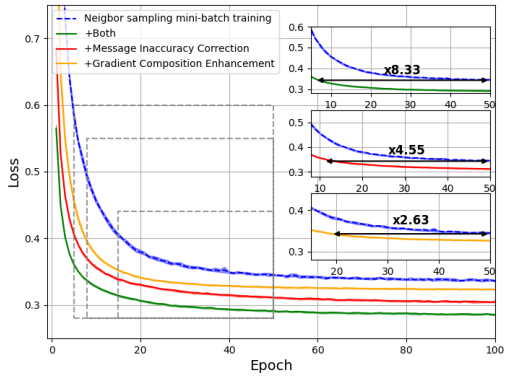


Figure 1: Comparison of training loss curves for neighbor sampling with and without Message Inaccuracy Correction and Gradient Composition Enhancement. The baseline (blue dashed line) uses the default neighbor sampling setup. Applying bias correction or variance reduction individually improves convergence speed, while combining both achieves the best performance. The three subplots on the right are zoomed-in views of the gray dashed rectangular regions in the main plot, providing a detailed comparison of convergence behavior.

composition. Therefore, we compared the effects on convergence when mitigating these two factors individually and together.

In Figure 1, we trained a three-layer GraphSAGE [14] model with node-wise sampling using a fanout of `[1,1,1]` and a batch size of 1024 on the ogbn-arxiv [16] dataset. Other training configurations followed the default settings in the official implementation[1]. The loss curves shown in Figure 1 represent the averages over 10 independent training runs to ensure statistical robustness. The default loss curve for neighbor sampling training is shown as the blue dashed line in Figure 1. Building upon this default neighbor sampling setup, we can apply two ideal methods—Gradient Composition Enhancement and Message Inaccuracy Correction—to mitigate the two error sources. Although these methods are impractical due to prohibitive computational costs, they enable us to explore changes in convergence properties. By applying these methods both individually and simultaneously, we generate three additional loss curves. The two ideal methods are described as follows:

- **Message Inaccuracy Correction**: The node embeddings involved are replaced by values computed using the current parameters and all neighbors. However, the computation graph is still constructed on the sampled mini-batch of nodes, so no additional items in the gradient composition are added.
- **Gradient Composition Enhancement**: For each mini-batch, the gradient is averaged over five different neighbor-sampled mini-batches for the same set of training nodes. This approach involves multiple samplings to obtain more gradient terms than a single sampling would provide, but it does not alter the inaccuracy of the message.

[1]examples/pytorch/graphsage/node_classification.py of https://github.com/dmlc/dgl/

Using the loss value at the 50th epoch of the default training curve as the target, we observe that mitigating inaccuracies in messages alone results in a 4.55x convergence speedup, while addressing the missing items in gradient composition alone results in a 2.63x speedup. Addressing both factors simultaneously achieves an 8.33x convergence speedup compared to the default training setup. This demonstrates that both factors significantly influence convergence and that their effects are additive.

Substantial effort has been made to utilize historical information on node embeddings to reduce the error in gradient estimation [6, 10, 28, 38]. However, these efforts primarily address message inaccuracy (part ①), overlooking opportunities to further reduce error from part ②. Additionally, their approach of maintaining historical embeddings across all layers for every node imposes significant memory demands. For instance, considering the ogbn-papers100M dataset [15], if a three-layer GNN model is used with a hidden dimension of 128 for each layer, approximately 162 GB of memory is required to cache the historical embeddings when using float32 representation.

To comprehensively and efficiently reduce the overall error, there are several significant challenges:

- **Maintaining and utilizing historical information.** Efficiently maintaining historical information and constructing critical compensation for part ② is a non-trivial challenge.
- **Scalability.** Maintaining historical information for all nodes in a straightforward manner incurs prohibitive memory costs, especially for large-scale graphs. This introduces a fundamental data management tension between the rapidly accumulating volume of reusable historical information and the limited memory budget available during training. Thus, it is critical to devise strategies for filtering or reducing the amount of historical information to ensure scalability.
- **Theoretical guarantee.** Leveraging historical information in training introduces new dynamics to the optimization process, and it is essential to provide theoretical guarantees for the estimation error reduction.

In this paper, we propose a novel framework, Pseudo Full Neighborhood Compensation (PFNC), to provides a comprehensive solution to the gradient estimation errors inherent in mini-batch training of GNNs with neighbor sampling. These errors stem from two key sources: inaccuracies in messages derived from sampled nodes and missing gradient contributions from unsampled target nodes. PFNC tackles both by maintaining caches of historical node embeddings and model parameter gradients. During training, it uses this cached data to correct inaccuracies in current computations and approximate missing gradient terms, effectively simulating a fuller neighborhood without the prohibitive cost of sampling all neighbors. This approach reduces both the variance and bias in gradient estimation and is compatible to any sampling strategy employed, making PFNC broadly applicable. Additionally, by prioritizing the retention of historical data based on node importance, PFNC achieves significant memory efficiency compared to methods requiring full embedding storage.

In summary, we have made the following contributions:

- We introduce PFNC, a framework that incorporates two innovative components: (1) a selective historical embedding cache to correct message inaccuracies with minimal memory overhead, and (2) a historical gradient cache to approximate missing gradient contributions.
- We provide theoretical analyses and guarantees for identifying and reducing gradient estimation error.
- The empirical experiments validate that this dual compensation strategy achieves superior convergence speed without sacrificing model performance.

The rest of this paper is structured as follows. Section 2 introduces preliminary concepts and related works. In Section 3, we analyze the sources of gradient estimation error in neighbor sampling to motivate our approach. Section 4 details the PFNC framework, including its components and theoretical foundations. Section 5 presents experimental results that showcase the effectiveness of PFNC. Finally, Section 6 concludes the paper and outlines future research directions.

2 PRELIMINARY AND RELATED WORK

2.1 Notations and Concepts

The notations are summarized in Table 1. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ represent a undirected graph, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges. $\mathcal{V} = \{v_1, \cdots, v_N\}$ with $|\mathcal{V}| = N$. Each node v_i has a feature vector $\boldsymbol{x}_{v_i} \in \mathbb{R}^{d_0}$.

We denote the complete 1-hop neighbors of node v_i as $\mathcal{N}(v_i) = \{v_j \mid (v_i, v_j) \in \mathcal{E}\}$. The complete k-hop neighbors of node v_i is denoted as

$$\mathcal{N}^{(k)}(v_i) = \left\{v_j \mid \exists v_{j'} \in \mathcal{N}^{(k-1)}(v_i),\ (v_{j'}, v_j) \in \mathcal{E}\right\}, \quad k \geq 2,$$

$$\mathcal{N}^{(1)}(v_i) = \mathcal{N}(v_i), \quad \mathcal{N}^{(0)}(v_i) = v_i.$$

For a set of nodes $S \subseteq \mathcal{V}$, we denote its k-hop neighbors as $\mathcal{N}^{(k)}(S) = \bigcup_{v \in S} \mathcal{N}^{(k)}(v)$.

2.1.1 Graph Neural Networks (GNNs). In the message passing perspective, a GNN layer is constructing and passing messages from source nodes $\mathcal{N}(v)$ to target node v.

For a GNN model with L layers, the representation of node v after L layers of message passing, denoted as $\boldsymbol{h}_v^{(L)} \in \mathbb{R}^{d_L}$, is utilized for various downstream graph-related tasks. v's representation at the l-th layer is computed using an update function $f^{(l)}$, as follows:

$$\boldsymbol{h}_v^{(l)} = f^{(l)}(\boldsymbol{h}_v^{(l-1)}, H_{\mathcal{N}(v)}^{(l-1)}), \quad l \in \{1, \ldots, L\}, \tag{1}$$

where $f^{(l)}$ is parameterized by learnable parameters $\theta^{(l)}$, and $H_{\mathcal{N}(v)}^{(l-1)} = \{\boldsymbol{h}_u^{(l-1)} \mid u \in \mathcal{N}(v)\}$ represents the set of representations of v's neighbors after the $(l-1)$-th layer. The initial representation $\boldsymbol{h}_v^{(0)}$ corresponds to the original node feature $\boldsymbol{x}_v$.

During training, the GNN model is optimized to minimize the following loss function:

$$\min_{\theta^{(1)}, \ldots, \theta^{(L)}, w} \mathcal{L}_{\mathcal{S}} = \sum_{v \in \mathcal{S}} \ell(w; \boldsymbol{h}_v^L), \tag{2}$$

where $\mathcal{S} \subset \mathcal{V}$ is the subset of nodes used for supervised learning, w represents the parameter of output layer giving the prediction based on $\boldsymbol{h}_v^L$ and ℓ is a function to calculate loss.

Table 1: General Notations with Corresponding Descriptions

Notation	Description
Graph Structure	
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	A graph node set $\mathcal{V}$ and edge set $\mathcal{E}$.
$\mathcal{V}$	The set of vertices with size $\lvert\mathcal{V}\rvert = N$.
$\mathcal{E}$	The set of edges.
v_i	The i-th vertex in $\mathcal{V}$.
$\boldsymbol{x}_{v_i}$	Feature vector of node v_i, $\boldsymbol{x}_{v_i} \in \mathbb{R}^{d_0}$.
$\mathcal{N}(v_i)$	The 1-hop neighbors of node v_i.
$\mathcal{N}^{(k)}(v_i)$	The k-hop neighbors of node v_i.
$\mathcal{N}^{(k)}(S)$	The k-hop neighbors of node set S.
Graph Neural Networks	
L	The number of layers in the GNN.
$\boldsymbol{h}_v^{(l)}$	The node representation at the l-th layer.
$\boldsymbol{h}_v^{(0)}$	The initial representation corresponding to the original node feature $\boldsymbol{x}_v$.
$f^{(l)}$	The update function parameterized by $\theta^{(l)}$ for the l-th layer.
$H_{\mathcal{N}(v)}^{(l-1)}$	The set of representations of node v's 1-hop neighbors at layer $(l-1)$.
Mini-Batch Training	
$\mathcal{V}_{\mathcal{B}}$	A mini-batch of training nodes.
$\mathbf{g}_{\theta^{(l)}}$	The gradient of the parameter $\theta^{(l)}$ during backward propagation of *exact mini-batch training*.
$\hat{\mathbf{g}}_{\mathcal{A},\theta^{(l)}}$	The stochastic gradient estimator of the parameter $\theta^{(l)}$ using neighbor sampling algorithm $\mathcal{A}$ in *neighbor sampling mini-batch training*.
$\mathcal{N}_{\mathcal{A},src}^{(L-l+1)}(v_i)$	Source nodes in layer l for node v_i using sampling algorithm $\mathcal{A}$.
$\mathcal{N}_{\mathcal{A},tar}^{(L-l)}(v_i)$	Target nodes in layer l for node v_i using sampling algorithm $\mathcal{A}$.
$\hat{\boldsymbol{h}}_v^{(l)}$	The node representation in neighbor sampling mini-batch training.
$\hat{H}_{\mathcal{N}_{\mathcal{A}}(v)}^{(l-1)}$	The set of neighbor sampling version embeddings of node v's sampled 1-hop neighbors at layer $(l-1)$.
Analysis and Method	
$\hat{\Delta}$	$\mathbf{g}_{\theta^{(l)}} - \hat{\mathbf{g}}_{\mathcal{A},\theta^{(l)}}$.
$\boldsymbol{c}(v_j)$,$\hat{\boldsymbol{c}}(v_j)$	The items within the summation of $\mathbf{g}_{\theta^{(l)}}$ and $\hat{\mathbf{g}}_{\mathcal{A},\theta^{(l)}}$.
K_h, K_g	The sizes of historical embedding cache and historical gradient cache.

2.1.2 Mini-batch Training of GNNs.

Definition 2.1 (Exact Mini-batch Training). In exact mini-batch training, the computation graph of the l-th layer forward propagation for mini-batch $\mathcal{V}_{\mathcal{B}}$ is a bipartite graph from the set of source nodes $\mathcal{N}^{(L-l+1)}(\mathcal{V}_{\mathcal{B}})$ to the set of target nodes $\mathcal{N}^{(L-l)}(\mathcal{V}_{\mathcal{B}})$.

Definition 2.2 (Neighbor Sampling Mini-batch Training). In neighbor sampling mini-batch training, the computation graph of the l-th layer forward propagation for mini-batch $\mathcal{V}_{\mathcal{B}}$ is a bipartite graph from a random subset of $\mathcal{N}^{(L-l+1)}(\mathcal{V}_{\mathcal{B}})$ to a random subset of $\mathcal{N}^{(L-l)}(\mathcal{V}_{\mathcal{B}})$ sampled by a neighbor sampling algorithm $\mathcal{A}$.

Remark 1. *Simply put, compared to exact mini-batch training, neighbor sampling mini-batch training uses a randomly degraded input subgraph for gradient computation to update the parameters of the GNN model. We can explicitly derive the formulas for calculating the parameter gradients of both methods, as shown in Eq. 3 and Eq. 4.*

For *exact mini-batch training*, the gradient of the parameter $\theta^{(l)}$, denoted as $\nabla_{\theta^{(l)}}\mathcal{L}_{\mathcal{V}_\mathcal{B}}$, is computed during backward propagation as follows:

$$\mathbf{g}_{\theta^{(l)}} = \nabla_{\theta^{(l)}}\mathcal{L}_{\mathcal{V}_\mathcal{B}} = \sum_{v_j \in \mathcal{N}^{(L-l)}(\mathcal{V}_\mathcal{B})} \nabla_{\theta^{(l)}} f^{(l)}_{\theta^{(l)}}(\boldsymbol{h}^{(l-1)}_{v_j}, H^{(l-1)}_{\mathcal{N}(v_j)}) \cdot \nabla_{\boldsymbol{h}^{(l)}_{v_j}}\mathcal{L}_{\mathcal{V}_\mathcal{B}} \tag{3}$$

For *neighbor sampling mini-batch training* with the neighbor sampling algorithm $\mathcal{A}$, we can use $\mathcal{N}^{(L+1-l)}_{\mathcal{A}(\xi),src}(v_i) \subseteq \mathcal{N}^{(L+1-l)}(v_i)$ and $\mathcal{N}^{(L-l)}_{\mathcal{A}(\zeta),tar}(v_i) \subseteq \mathcal{N}^{(L-l)}(v_i)$ to denote the source nodes and target nodes in the l-th layer for $v_i \in \mathcal{V}_\mathcal{B}$. These are stochastic subsets defined by $\mathcal{A}$, where ξ and ζ represent the randomness and are omitted in subsequent expressions for simplicity. The corresponding stochastic estimator of $\mathbf{g}_{\theta^{(l)}}$ is then given by:

$$\hat{\mathbf{g}}_{\mathcal{A},\theta^{(l)}} = \sum_{v_j \in \mathcal{N}^{(L-l)}_{\mathcal{A},tar}(\mathcal{V}_\mathcal{B})} \nabla_{\theta^{(l)}} f^{(l)}_{\theta^{(l)}}(\hat{\boldsymbol{h}}^{(l-1)}_{v_j}, \hat{H}^{(l-1)}_{\mathcal{N}_\mathcal{A}(v_j)}) \cdot \nabla_{\hat{\boldsymbol{h}}^{(l)}_{v_j}}\mathcal{L}_{\mathcal{V}_\mathcal{B}}, \tag{4}$$

where the symbol $\hat{\cdot}$ denotes the neighbor sampling version of model parameters' gradients or node embeddings.

2.2 Related Works

Gradient Compensation Methods. The most closely related works to our research are GAS [10], GraphFM [38], and LMC [28], which enhance subgraph sampling in GNN training. GAS includes all 1-hop in-batch neighbors during forward propagation and uses historical embeddings for out-batch nodes. GraphFM adds edges between in- and out-batch nodes to update historical embeddings with momentum, reducing staleness. LMC preserves 1-hop neighborhoods in sampled subgraphs and uses historical compensation in both forward and backward propagation. However, these methods mainly address message inaccuracy, neglecting missing gradient components caused by unsampled target nodes.

Historical Embeddings for Efficiency. Other works like SANCUS [26] and FreshGNN [17] use historical embeddings to reduce computational cost rather than estimation error. SANCUS applies staleness-aware updates in full-graph training to reduce communication, at the cost of bounded approximation. FreshGNN uses historical embeddings to prune subgraphs post-sampling, minimizing memory usage while accepting some accuracy degradation.

Other Complementary Approaches. Recent works improve GNN training efficiency through message or feature compression (e.g., BiFeat [24], AMQP [31], F2CGT [23]) and distributed computation frameworks (e.g., DistDGL [43], SALIENT++ [18], AliGraph [45]). These methods address system-level bottlenecks like feature loading and communication, but not the algorithmic error due to sampling. In contrast, PFNC targets this error and is orthogonal to both lines: it can be combined with compression methods to improve convergence while preserving system efficiency, and integrated into distributed training pipelines as a lightweight, local-cache-based module without interfering with the system design.

3 MOTIVATION

In this section, we conduct a thorough analysis of the difference $\hat{\Delta}$ between $\mathbf{g}_{\mathcal{A},\theta^{(l)}}$ and $\hat{\mathbf{g}}_{\mathcal{A},\theta^{(l)}}$, which quantifies the error in gradient estimation during mini-batch training with neighbor sampling. This error can be decomposed into two parts, as illustrated in Eq. 6 and explained in Remark 2. The mean and variance of the error $\hat{\Delta}$ are detailed in Eq. 7. From this analysis, we observe that the gradient estimator $\hat{\mathbf{g}}_{\mathcal{A},\theta^{(l)}}$ exhibits bias due to the part ①, while both part ① and ② contribute to the variance of the error. This insight motivates us to identify the limitations of existing methods and simultaneously address both sources of error.

3.1 Analysis of Gradient Estimation Error

3.1.1 Mean and Variance of $\hat{\Delta}$. For simplicity, we denote $\boldsymbol{c}(v_j)$ and $\hat{\boldsymbol{c}}_\mathcal{A}(v_j)$ as the items under summation in Eq. 3 and Eq. 4, respectively. We first define an auxiliary estimator $\mathbf{g}'_{\mathcal{A},\theta^{(l)}}$ for the convenience of analysis:

$$\mathbf{g}'_{\mathcal{A},\theta^{(l)}} = \sum_{v_j \in \mathcal{N}^{(L-l)}_{\mathcal{A},tar}(\mathcal{V}_\mathcal{B})} \frac{|\mathcal{N}^{(L-l)}(\mathcal{V}_\mathcal{B})|}{|\mathcal{N}^{(L-l)}_{\mathcal{A},tar}(\mathcal{V}_\mathcal{B})|} \boldsymbol{c}(v_j). \tag{5}$$

This auxiliary estimator preserves the computational graph of neighbor sampling mini-batch training but replaces the stochastic messages $\hat{\boldsymbol{c}}(v_j)$ with their exact counterparts $\boldsymbol{c}(v_j)$.

Assume that algorithm $\mathcal{A}$ conducts unbiased sampling from $\mathcal{N}^{(L-l)}(\mathcal{V}_\mathcal{B})$ and then we have $\mathbb{E}_\mathcal{A}\left[\mathbf{g}_{\theta^{(l)}} - \mathbf{g}'_{\mathcal{A},\theta^{(l)}}\right] = 0$. The variance $\mathbf{Var}_\mathcal{A}\left[\mathbf{g}_{\theta^{(l)}} - \mathbf{g}'_{\mathcal{A},\theta^{(l)}}\right] = \mathbf{Var}_\mathcal{A}[\mathbf{g}'_{\mathcal{A},\theta^{(l)}}]$.

Then the difference we aim to analyze, $\hat{\Delta}$, can be decomposed into two parts as follows:

$$\hat{\Delta} = \mathbf{g}_{\theta^{(l)}} - \hat{\mathbf{g}}_{\mathcal{A},\theta^{(l)}} = \underbrace{\left(\mathbf{g}'_{\mathcal{A},\theta^{(l)}} - \hat{\mathbf{g}}_{\mathcal{A},\theta^{(l)}}\right)}_{①} + \underbrace{\left(\mathbf{g}_{\theta^{(l)}} - \mathbf{g}'_{\mathcal{A},\theta^{(l)}}\right)}_{②},$$

$$① = \sum_{v_j \in \mathcal{N}^{(L-l)}_{\mathcal{A},tar}(\mathcal{V}_\mathcal{B})} \left(\frac{|\mathcal{N}^{(L-l)}(\mathcal{V}_\mathcal{B})|}{|\mathcal{N}^{(L-l)}_{\mathcal{A},tar}(\mathcal{V}_\mathcal{B})|}\boldsymbol{c}(v_j) - \hat{\boldsymbol{c}}(v_j)\right) \tag{6}$$

$$② = \sum_{v_j \in \mathcal{N}^{(L-l)}(\mathcal{V}_\mathcal{B}) / \mathcal{N}^{(L-l)}_{\mathcal{A},tar}(\mathcal{V}_\mathcal{B})} \boldsymbol{c}(v_j) + \left(1 - \frac{|\mathcal{N}^{(L-l)}(\mathcal{V}_\mathcal{B})|}{|\mathcal{N}^{(L-l)}_{\mathcal{A},tar}(\mathcal{V}_\mathcal{B})|}\right) \sum_{v_j \in \mathcal{N}^{(L-l)}_{\mathcal{A},tar}(\mathcal{V}_\mathcal{B})} \boldsymbol{c}(v_j).$$

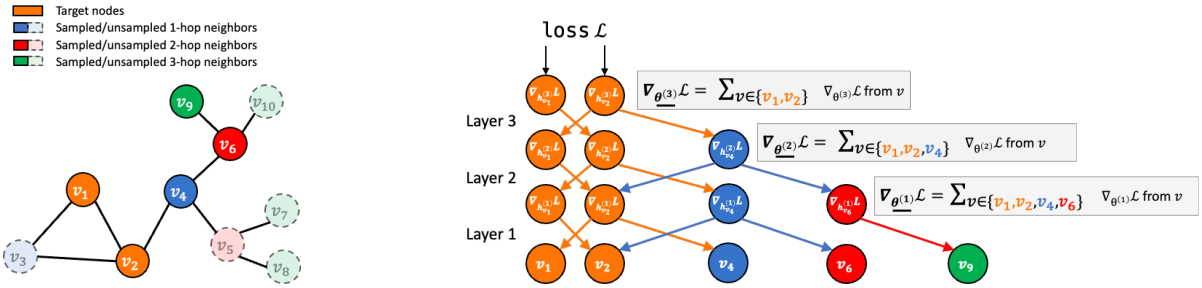
The expectation and variance of $\hat{\Delta}$ are as follows:

$$\begin{aligned} \mathbb{E}_\mathcal{A}\left[\hat{\Delta}\right] &= \mathbb{E}_\mathcal{A}[①] + \mathbb{E}_\mathcal{A}[②] = \mathbb{E}_\mathcal{A}[①], \\ \mathbf{Var}_\mathcal{A}\left[\hat{\Delta}\right] &= \mathbf{Var}_\mathcal{A}[①] + \mathbf{Var}_\mathcal{A}[②] + 2\mathbf{Cov}_\mathcal{A}[①, ②]. \end{aligned} \tag{7}$$

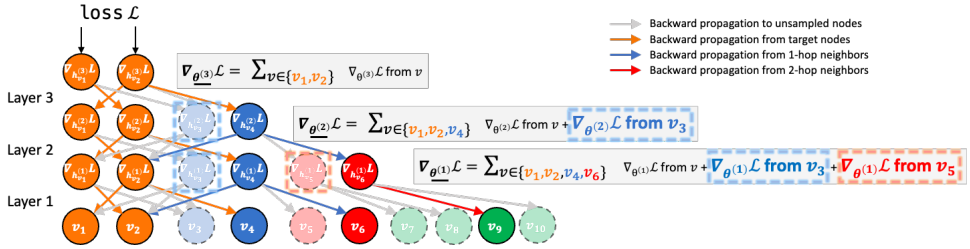
In Eq. 7, the expectation $\mathbb{E}_\mathcal{A}\left[\hat{\Delta}\right]$ is equal to $\mathbb{E}_\mathcal{A}[①]$, which indicates that part ② does not contribute to the bias of the gradient estimation under the assumption of unbiased sampling.

Remark 2. *The decomposition is intended to better illustrate the sources of gradient estimation error. Part ① represents the inaccuracies in the item $\hat{\mathbf{c}}(v_j)$ during backpropagation, which stems from inaccuracies of $\hat{\boldsymbol{h}}^{(l-1)}_{v_j}$ when sampled computational graph is fixed. Part ② primarily accounts for the contributions to the gradient from target nodes that were not sampled and corresponds to the missing items in gradient composition, as illustrated in Figure 2.*

To intuitively demonstrate part ②, Figure 2 compares the backpropagation of a 3-layer GNN using neighbor sampling mini-batch training and exact mini-batch training on an example graph. In Figure 2a, the orange nodes $\{v_1, v_2\}$ represent the current mini-batch



(a) Node-wise sampling on an example graph. **(b) Backward propagation of neighbor sampling mini-batch training with node-wise sampling.**



(c) Comparison of backward propagation between exact mini-batch training and neighbor sampling mini-batch training with node-wise sampling, illustrating the impact of part ②.

Figure 2: Comparison of neighbor sampling mini-batch training and exact mini-batch training.

$\mathcal{V}_{\mathcal{B}}$. The blue, red, and green nodes represent the 1-hop, 2-hop, and 3-hop neighbors of $\mathcal{V}_{\mathcal{B}}$, respectively. The dashed circles indicate nodes that were not sampled by the node-wise neighbor sampling algorithm. Figure 2b visually illustrates the composition of parameter gradients in each layer of the GNN during backpropagation in neighbor sampling mini-batch training, which corresponds to Eq. 4. Figure 2c shows the composition of parameter gradients during backpropagation in exact mini-batch training, as described in Eq. 3, with a focus on comparing the differences in backpropagation and gradient composition between the two mini-batch training methods. It can be observed that for neighbor sampling mini-batch training, since v_3 was not sampled as a 1-hop neighbor of $\mathcal{V}_{\mathcal{B}}$, it is absent from the target nodes in the forward propagation of layer 2. Consequently, compared to exact mini-batch training, $\nabla_{\theta^{(2)}}\mathcal{L}$ lacks the gradient contribution from v_3. Similarly, because v_3 and v_5 were not sampled as 2-hop neighbors, $\nabla_{\theta^{(1)}}\mathcal{L}$ in layer 1 also misses the gradient contributions from v_3 and v_5. These missing gradient contributions are highlighted by dashed rectangles in Figure 2c.

3.2 Improvement Objectives

Our work aims to address two critial limitations of existing approaches for gradient estimation.

First, existing solutions for addressing part ① require maintaining full historical embeddings $\hat{\boldsymbol{h}}_v^{(l)}$ for all nodes across all layers, resulting in prohibitive memory overhead for large graphs. This design arises from their strategies, which are coupled with specially crafted subgraph-wise sampling methods that explicitly mandate the participation of all 1-hop neighbors in the message passing process and the embeddings of out-batch 1-hop neighbor nodes depend solely on their historical embeddings.

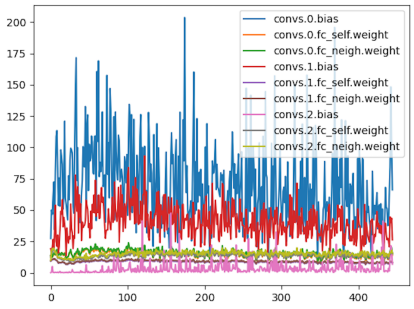


Figure 3: Empirical relative variance of gradients for different layers and parameters over training iterations.

Second, current methods like [10, 28, 38] fundamentally neglect the *missing items in gradient composition* (part ②) caused by unsampled target nodes. This limitation stems from the computation graph constraints in backpropagation: gradient contributions can only be collected when target nodes are explicitly sampled during forward propagation, as demonstrated in Figure 2. This results in high variance in gradient estimation, as illustrated in Figure 3, which shows the empirical relative variance of gradients computed using ten different neighbor samplings under the same experimental conditions as "+Message Inaccuracy Correction" in Figure 1. It is evident that the variance of the gradients is significantly large relative to the gradients themselves, especially for layers handling neighbors from farther hops, as these layers experience exponentially more missing gradient contributions.

Therefore, our objectives are twofold: first, to develop a framework that utilizes a more economical historical node embedding cache to address the inaccuracies in the item $\boldsymbol{c}(\hat{v}_j)$, which contribute to the bias in gradient estimation; and second, to introduce a design that reduces the error arising from missing items in gradient composition, thereby mitigating the high variance in gradient estimation.

4 PSEUDO FULL NEIGHBORHOOD COMPENSATION

In this section, we propose the Pseudo Full Neighborhood Compensation (PFNC) framework to achieve the improvement objectives. PFNC consists of two components: 1) a Historical Embedding Cache to address message inaccuracies (error ①) during forward propagation, as illustrated in Figure 4; and 2) a Historical Gradient Cache to enhance gradient composition (error ②) during backward propagation, as shown in Figure 7.

Algorithm 1 outlines the details of mini-batch GNN training using the PFNC framework. Unlike conventional training processes, PFNC leverages historical information to refine both the node embeddings obtained during the forward pass and the model parameter gradients calculated during the backward pass. Furthermore, after completing the model update, the caches of historical information are refreshed based on the data from the current iteration.

In the following sections, we will introduce these two components individually, explaining how each component mitigates its respective sources of error while providing the corresponding theoretical foundations.

4.1 Message Inaccuracy Correction via Historical Embedding Cache

4.1.1 Correction Strategy. To address error ①, we adopt a mixing operation similar to existing works [28, 38], which integrates historical values into the computed node embedding during forward propagation. The update procedure can be defined as follows:

$$\hat{\boldsymbol{h}}_v^{(l)} = f^{(l)}(\hat{\boldsymbol{h}}_v^{(l-1)}, \hat{H}^{(l-1)}_{\mathcal{N}_{\mathcal{A}}(v)}) \tag{8}$$

$$\hat{\boldsymbol{h}}_v^{(l)} \leftarrow \beta\hat{\boldsymbol{h}}_v^{(l)} + (1-\beta)\bar{\boldsymbol{h}}_v^{(l)} \text{ [If } \bar{\boldsymbol{h}}_v^{(l)} \text{ not in cache, skip.]} \tag{9}$$

$$\bar{\boldsymbol{h}}_v^{(l)} \leftarrow \hat{\boldsymbol{h}}_v^{(l)} \text{ [If cache overflows, evict by importance.]} \tag{10}$$

Here, Equation 8 computes the initial embedding $\hat{\boldsymbol{h}}_v^{(l)}$ for node v at layer l by applying GNN message passing on the previous layer's embeddings. Equation 9 then mixes this with the cached historical embedding $\bar{\boldsymbol{h}}_v^{(l)}$, using $\beta \in (0, 1)$ as a weighting factor to balance current and past information, improving accuracy if the historical value exists in the cache; otherwise, it skips this step. Finally, Equation 10 updates the cache by storing the new embedding $\hat{\boldsymbol{h}}_v^{(l)}$, evicting less important entries if the cache is full, with importance detailed in Section 4.1.2.

Cache Update Policy. Let S_{cache} denote the set of nodes currently stored in the cache, and let K_h be the cache size. The newly computed node embeddings at the current step, obtained via Eq. 8, correspond to the set $\mathcal{N}^{(l)}_{\mathcal{A}_t,tar}(\mathcal{V}_{\mathcal{B}_t})$, denoted as S_{new}. To update the cache, we first update the historical embeddings and importance of the nodes in the intersection $S_{\text{cache}} \cap S_{\text{new}}$. Subsequently, the cache is updated to retain the top K_h nodes with the highest importance scores from the union $S_{\text{cache}} \cup S_{\text{new}}$. This policy guarantees that when a historical node embedding is accessed, it reflects the latest computation while maintaining efficiency by prioritizing nodes based on their importance.

Figure 4 illustrates an example where the historical cache retains embeddings and their associated importance for nodes v_2 and v_7. After the message passing in layer 2, node v_2 utilizes the cache to mix its historical value. For cache updating, we first update $\bar{\boldsymbol{h}}^{(2)}_{v_2}$ and importance for v_2 in the cache , and then compare the importance of v_1, v_2, v_4, and v_7 to retain the two with the highest importance.

Theorems 4.1 and 4.2 support this mixing operation in Eq. 9 can reduce error ①.

Algorithm 1 Mini-batch Training with PFNC

Input: Sampling algorithm $\mathcal{A}$, cache sizes K_g, K_h, mixing ratios $\{\alpha^{(l)}\}, \{\beta\}$
Output: Optimized parameters Θ_{T+1}
Initialize caches $\{Q_g^{(l)}\}, \{Q_h^{(l)}\}$ and parameter index
Initialize model parameters Θ_1
for $t = 1$ **to** T **do**
  Sample batch $\mathcal{V}_{\mathcal{B}_t}$ and construct computation graph via $\mathcal{A}$
  // **Forward Pass:**
  for each layer l and each target node v in layer l **do**
    Compute embeddings $\hat{\boldsymbol{h}}_v^{(l)}$ via Eq. 8
    if $\bar{\boldsymbol{h}}_v^{(l)} \in Q_h^{(l)}$ **then**
      Apply mixing: $\hat{\boldsymbol{h}}_v^{(l)} \leftarrow \beta\hat{\boldsymbol{h}}_v^{(l)} + (1-\beta)\bar{\boldsymbol{h}}_v^{(l)}$
    end if
  end for
  // **Backward Pass:**
  for each layer l **do**
    Compute gradients $\hat{\mathbf{g}}_{\mathcal{A}_t,\theta^{(l)}}$ via Eq. 4
    Retrieve historical gradients $\{\hat{\mathbf{g}}_{\mathcal{A}_\tau,\theta^{(l)}}\}_{\tau\in\mathcal{S}_t}$ from $Q_g^{(l)}$
    Compute enhanced gradient: $\hat{\mathbf{g}}_{PFNC_t,\theta^{(l)}} = \alpha^{(l)}\hat{\mathbf{g}}_{\mathcal{A}_t,\theta^{(l)}} + \frac{(1-\alpha^{(l)})}{|\mathcal{S}_t|}\sum_{\tau\in\mathcal{S}_t}\hat{\mathbf{g}}_{\mathcal{A}_\tau,\theta^{(l)}_\tau}$
  end for
  // **Parameter Update:**
  Update Θ_t using $\hat{\mathbf{g}}_{PFNC_t}$
  // **Cache Maintenance:**
  for each layer l **do**
    Update $Q_h^{(l)}$ with new embeddings and importance
    Update $Q_g^{(l)}$ by enqueuing the new gradient and dequeuing the oldest one
  end for
end for

Theorem 4.1. *Assume that (i) $\nabla_{\theta^{(l)}} f^{(l)}_{\theta^{(l)}}$ is γ-Lipschitz continuity; (ii) norms $\|\nabla_{\theta^{(l)}} f^{(l)}_{\theta^{(l)}}(\boldsymbol{h}^{(l-1)}_{v_j}, H^{(l-1)}_{\mathcal{N}(v_j)})\|_F$ is bounded by $G_1 > 0$; (iii) $\|\nabla_{\hat{\boldsymbol{h}}^{(l)}_{v_j}}\mathcal{L}_{\mathcal{V}_{\mathcal{B}}}\|_2$ and $\|\nabla_{\boldsymbol{h}^{(l)}_{v_j}}\mathcal{L}_{\mathcal{V}_{\mathcal{B}}}\|_2$ are both bounded by $G_2 > 0$*

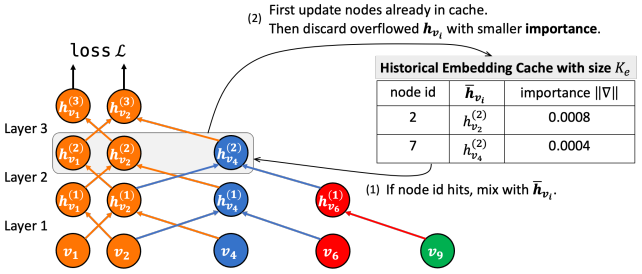


Figure 4: Forward propagation with historical embedding cache and cache update.

and $G_2 \to 0$ as the optimization process approaches convergence; Then, the bias of gradient estimation $\hat{\mathbf{g}}_{\mathcal{A},\theta^{(l)}}$ is bounded by the variance of $\|\hat{\boldsymbol{h}}_{v_j}\|_2$.

Proof. As $\|\mathbf{Aa}-\mathbf{Bb}\|_2 \le \|\mathbf{A}\|_F\|\mathbf{a}-\mathbf{b}\|_2 + \|\mathbf{A}-\mathbf{B}\|_F\|\mathbf{b}\|_2$, we can bound $\|①\|_2$ by

$$\begin{aligned}
\|①\|_2 &\le \sum_{v_j \in \mathcal{N}^{(L-l+1)}_{\mathcal{A},tar}(\mathcal{V}_\mathcal{B})} \left\| \frac{|\mathcal{N}^{(L-l)}(\mathcal{V}_\mathcal{B})|}{|\mathcal{N}^{(L-l)}_{\mathcal{A},tar}(\mathcal{V}_\mathcal{B})|}\boldsymbol{c}(v_j) - \hat{\boldsymbol{c}}(v_j) \right\|_2 \\
&\le |\mathcal{N}^{(L-l)}(\mathcal{V}_\mathcal{B})| \max_{v_j \in \mathcal{N}^{(L-l)}_{\mathcal{A},tar}(\mathcal{V}_\mathcal{B})} \left\| \nabla_{\theta^{(l)}} f^{(l)}_{\theta^{(l)}}(\boldsymbol{h}^{(l-1)}_{v_j}, H^{(l-1)}_{\mathcal{N}(v_j)}) \cdot \nabla_{\boldsymbol{h}^{(l)}_{v_j}} \mathcal{L}_{\mathcal{V}_\mathcal{B}} \right. \\
&\quad \left. - \frac{|\mathcal{N}^{(L-l)}_{\mathcal{A},tar}(\mathcal{V}_\mathcal{B})|}{|\mathcal{N}^{(L-l)}(\mathcal{V}_\mathcal{B})|} \nabla_{\theta^{(l)}} f^{(l)}_{\theta^{(l)}}(\hat{\boldsymbol{h}}^{(l-1)}_{v_j}, \hat{H}^{(l-1)}_{\mathcal{N}_\mathcal{A}(v_j)}) \cdot \nabla_{\hat{\boldsymbol{h}}^{(l)}_{v_j}} \mathcal{L}_{\mathcal{V}_\mathcal{B}} \right\|_2 \\
&\le |\mathcal{N}^{(L-l)}(\mathcal{V}_\mathcal{B})| G_1 G_2 + \gamma G_1 |\mathcal{N}^{(L-l)}_{\mathcal{A},tar}(\mathcal{V}_\mathcal{B})| \|\boldsymbol{H}^{(l)} - \hat{\boldsymbol{H}}^{(l)}\|_F \\
&\le |\mathcal{N}^{(L-l)}(\mathcal{V}_\mathcal{B})| G_1 G_2 + \gamma G_1 |\mathcal{N}^{(L-l)}_{\mathcal{A},tar}(\mathcal{V}_\mathcal{B})| \sum_{v_j \in \mathcal{V}} \mathbf{Var}[\|\hat{\boldsymbol{h}}_{v_j}\|_2]
\end{aligned}$$

□

Theorem 4.2. *Provided that the variance of the historical embedding satisfies $\mathbf{Var}[\|\bar{\boldsymbol{h}}_{v_j}\|_2] < \frac{1+\beta}{1-\beta}\mathbf{Var}[\|\hat{\boldsymbol{h}}_{v_j}\|_2]$, mixing historical values $\bar{\boldsymbol{h}}_{v_j}$ into $\hat{\boldsymbol{h}}_{v_j}$ can reduce its variance.*

Proof. Given the convexity of the norm and the fact that $\bar{\boldsymbol{h}}_{v_j}$ and $\hat{\boldsymbol{h}}_{v_j}$ are independent prior to mixing, we have

$$\begin{aligned}
&\mathbf{Var}[\|\beta\hat{\boldsymbol{h}}^{(l)}_{v_j} + (1-\beta)\bar{\boldsymbol{h}}^{(l)}_{v_j}\|_2] \\
\le&(\beta)^2\mathbf{Var}[\|\hat{\boldsymbol{h}}^{(l)}_{v_j}\|_2] + (1-\beta)^2\mathbf{Var}[\|\bar{\boldsymbol{h}}^{(l)}_{v_j}\|_2] < \mathbf{Var}[\|\hat{\boldsymbol{h}}^{(l)}_{v_j}\|_2]
\end{aligned}$$

□

4.1.2 Importance of Historical Embeddings. A significant aspect of our design is the selective maintenance of historical values to save memory. Consequently, in Eq. 9, the mixing operation is executed only when $\bar{\boldsymbol{h}}^{(l)}_v$ is currently present in the cache.

The decision not to retain historical embeddings for all nodes is based on two key observations:

First, given the sampling parameters (such as fan-out), many nodes have a limited number of neighbors. Therefore, the variance introduced by neighbor sampling at these nodes can be minimal. Figure 5 depicts the neighbor count distribution for the ogbn-arxiv dataset, revealing that most nodes have a relatively small number of neighbors. For instance, with node-wise sampling and a fan-out of 5 in the current GNN layer, nearly half of all nodes experience minimal randomness from neighbor sampling.

Second, as training progresses, the model may learn robust patterns that mitigate the effects of sampling. This indicates that, for certain nodes, the computed embeddings for the next layer may not vary significantly, even when different subsets of neighbors are sampled. We observe a trend where embeddings with a larger gradient norm tend to exhibit smaller variance. Figure 6 illustrates the relationship between empirical embedding variance (calculated using ten neighbor sampling) and the gradient norm (averaged across these sampling) for the node embeddings of last layer, recorded over five epochs of training a three-layer GraphSAGE model on ogbn-arxiv with fan-outs of `[5,5,5]`.

Based on these observations, we use the norm of embedding gradients as a valuable metric for assessing the importance of historical embeddings in reducing error. This is because the bias $\mathbb{E}_\mathcal{A}[①]$ is upper bounded by the variance of node embeddings, and utilizing historical embeddings with smaller variance can further reduce the bias. Consequently, during the maintenance of the historical embedding cache, we employ this importance measure to evict less significant historical information, resulting in a more economical historical node embedding cache. We also have Theorem 4.3 to ensure the advantages of cache eviction in the worst case compared to maintaining historical embedding caches for all nodes.

Theorem 4.3. *In the worst case, the variance resulting from cache eviction is smaller than that obtained by maintaining historical caches for all nodes.*

Proof Sketch. In the worst case, where the historical embeddings have a variance that does not satisfy the condition $\mathbf{Var}[\|\bar{\boldsymbol{h}}_{v_j}\|_2] < \frac{1+\beta}{1-\beta}\mathbf{Var}[\|\hat{\boldsymbol{h}}_{v_j}\|_2]$, mixing with these historical values will lead to an increase in the variance of $\hat{\boldsymbol{h}}_{v_j}$. By retaining only the historical embeddings with the smallest variance in the cache, as dictated by our cache update policy, we can achieve a smaller variance in the worst case.

4.2 Gradient Composition Enhancement via Historical Gradient Cache

To reduce the high variance caused by error ② in $\hat{\Delta}$. Ideally, we can use $\hat{\mathbf{g}}^{(K)}_{\mathcal{A},\theta^{(l)}} = \frac{1}{K}\sum_{i=1}^{K}\hat{\mathbf{g}}_{\mathcal{A}_i,\theta^{(l)}}$, which represents the average estimation based on K neighbor samplings, as a more accurate estimator. Compared with Eq. 7, $\hat{\Delta}^{(K)} = \mathbf{g}_{\theta^{(l)}} - \hat{\mathbf{g}}^{(K)}_{\mathcal{A},\theta^{(l)}}$ has the same expectation and smaller variance:

$$\mathbb{E}_\mathcal{A}\left[\hat{\Delta}^{(K)}\right] = \mathbb{E}_\mathcal{A}\left[\hat{\Delta}\right], \mathbf{Var}_\mathcal{A}\left[\hat{\Delta}^{(K)}\right] = \frac{1}{K}\mathbf{Var}_\mathcal{A}\left[\hat{\Delta}\right].$$

It is worth noting that using $\hat{\Delta}^{(K)}$ is orthogonal to efforts aimed at addressing the estimation bias from part ①.

Although $\hat{\mathbf{g}}^{(K)}_{\mathcal{A},\theta^{(l)}}$ is a better gradient estimation, it is an idealized version since each estimation requires K samplings, resulting in K times the computational cost (i.e., K forward and backward passes). In practice, we can reduce the computational cost by using the historical values of $\hat{\mathbf{g}}_{\mathcal{A},\theta^{(l)}}$ from previous iterations to provide an approximation for the current iteration at the cost of some staleness.

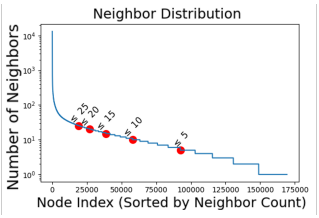


Figure 5: Neighbor count distribution of ogbn-arxiv.

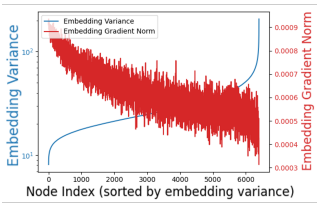


Figure 6: Embedding variance and embedding gradient norm.

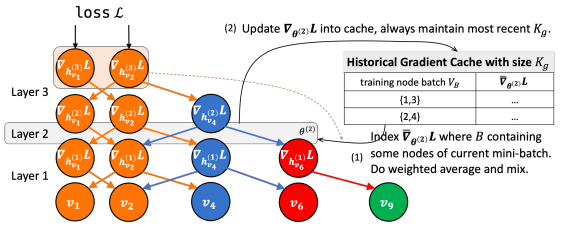


Figure 7: Backward propagation with historical gradient cache and cache update.

Here Let $\mathcal{V}_{\mathcal{B}_t}$ be the batch of training nodes at iteration t and $\hat{\mathbf{g}}_{\mathcal{A}_t,\theta_t^{(l)}}$ be the gradient estimation at layer l at iteration t. The PFNC gradient estimator at iteration t, $\hat{\mathbf{g}}_{PFNC_t}$, is defined as follows:

$$\hat{\mathbf{g}}_{PFNC_t,\theta_t^{(l)}} = \alpha\, \hat{\mathbf{g}}_{\mathcal{A}_t,\theta_t^{(l)}} + (1-\alpha)\frac{1}{|\mathcal{S}_t|}\sum_{\tau\in\mathcal{S}_t} \hat{\mathbf{g}}_{\mathcal{A}_\tau,\theta_\tau^{(l)}}. \tag{11}$$

$\mathcal{S}_t$ is the set of the most recent iterations in which some v in $\mathcal{V}_{\mathcal{B}_t}$ has been sampled, i.e.

$$\mathcal{S}_t = \left\{ \tau_v \middle| \begin{array}{r} \tau_v = \max\{\ \tau \mid \tau \in \{t-K_g,\ldots,t-1\} \\ \text{and } v \in \mathcal{V}_{\mathcal{B}_\tau}\ \},\, v \in \mathcal{V}_{\mathcal{B}_t} \end{array} \right\}$$

The set $\mathcal{S}_t$ is restricted to the most recent K_g iterations to control the staleness, so the historical gradient cache is just a K_g-sized queue. α is also introduced to balance the trade-off between the staleness and the variance reduction.

Figure 7 illustrates an example where the training nodes in the sampled mini-batch are v_1 and v_2. The cache contains historical gradients for training nodes $\{v_1, v_3\}$ and $\{v_2, v_4\}$. To compensate for the missing gradients in $\nabla_{\theta^{(2)}}\mathcal{L}_{\{v_1,v_2\}}$, we access historical gradients that contain supervisory information from v_1 and v_2. For the corresponding historical gradient $\bar{\nabla}_{\theta^{(2)}}\mathcal{L}_{\mathcal{V}_\mathcal{B}}$ of the training nodes batch that includes v_1 or v_2, we compute the mean and mix it with the currently computed gradient. The gradient queue is updated by adding the new gradient associated with the batch $\mathcal{V}_\mathcal{B} = \{v_1, v_2\}$ and pop the oldest one associated with $\mathcal{V}_\mathcal{B} = \{v_1, v_3\}$.

4.3 Complexity Analysis

We compare the memory requirements of several methods, including GAS [10], GraphFM [38], LMC [28], and PFNC.

GAS, GraphFM, and LMC all maintain historical node embeddings $\hat{\boldsymbol{h}}_{v_j}^{(l-1)}$ for every node in every layer, which requires at least $O(NdL)$ memory, where d is the dimension of node embeddings and L is the number of layers. Additionally, LMC requires extra memory to store the historical $\nabla_{\hat{\boldsymbol{h}}_{v_j}^{(l)}}\mathcal{L}_{\mathcal{V}_\mathcal{B}}$, resulting in a total memory requirement of $O(2NdL)$.

In contrast, PFNC significantly reduces memory requirements. PFNC maintains only a K_h-sized historical embedding cache and a K_g-sized queue of historical gradients and tow index structure with `Index.shape` $= [N,]$, leading to a memory complexity of $O(K_h dL + K_g M + 2N)$, where M is the number of model parameters and N is the number of nodes in the graph. This design makes PFNC more memory-efficient, especially in scenarios with a large number of nodes or layers, as shown in Table 2.

Table 2: Comparison of Additional Memory Costs

GAS	GraphFM	LMC	PFNC
$O(NdL)$	$O(NdL)$	$O(2NdL)$	$O(K_h dL + K_g M + 2N)$

Since the parameter size of GNN models is typically very small compared to the feature vectors of input nodes and K_g is usually not large, the memory overhead for maintaining the gradient queue is minimal. Additionally, significant memory savings are achieved for the historical embedding cache, as $K_h \ll N$. The index structure, which serves as a pointer for each training node, incurs negligible memory overhead relative to other training memory requirements. Consequently, the time overhead for calculating $\hat{\mathbf{g}}_{PFNC_t}$ is also minimal in the context of the overall training process, a fact that will be further substantiated by the runtime results in subsequent experiments. For example, in the ogbn-products dataset, with parameters $N = 2.4\times10^6$, $d = 128$, $L = 3$, $K_h = 10^4$, $K_g = 16$, and $M \approx Ld^2 \approx 10^4$, PFNC uses less than 1% of memory to cache historical information.

5 EXPERIMENTS

In this section, we first elaborate on the experimental settings in Section 5.1. Then, we demonstrate the benefits of equipping PFNC with existing gradient compensation methods in Section 5.2. Next, we show the improvements PFNC can bring to standard neighbor sampling methods in Section 5.3. Finally, we conduct an ablation study in Section 5.5 and perform hyperparameter analysis in Section 5.6.

5.1 Experimental Settings

5.1.1 Datasets. We perform node classification benchmark on six datasets, including three small citation network datasets: *Cora*, *Citeseer*, and *Pubmed* [27], and three large datasets: *ogbn-arxiv* [16], *ogbn-products* [16], and *Yelp*[40]. The statistics of the datasets are summarized in Table 3. These node classification datasets all use accuracy as the evaluation metric.

5.1.2 Baselines. To thoroughly evaluate the effectiveness of PFNC, we select a variety of baseline methods, including existing gradient compensation methods and standard neighbor sampling methods without gradient compensation.

The gradient compensation method baselines include GAS [10], GraphFM [38], and LMC [28]. For standard neighbor sampling method baselines, we choose GraphSAGE [14], LADIES [47], LABOR [2] and Cluster-GCN [7] as representatives of node-wise, layer-wise, and subgraph-wise sampling methods.

Table 3: Dataset statistics.

Datasets	Classes	Nodes	Edges	Features
Cora	7	2,708	5,278	1,433
CiteSeer	6	3,327	4,552	3,703
PubMed	3	19,717	44,324	500
Yelp	100	716,847	6,977,409	300
ogbn-arxiv	40	169,343	1,157,799	128
ogbn-products	47	2,449,029	61,859,140	100
ogbn-papers100M	172	111M	1.6B	128

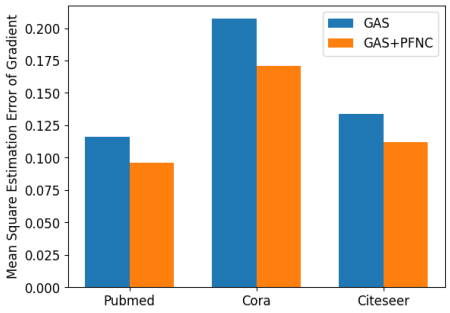


Figure 8: The estimation error of gradients.

Table 4: Comparison of the effects of incorporating PFNC into existing gradient estimation improvement methods on large graph datasets. The comparison focuses on three aspects: the final converged test accuracy, the number of epochs and the training time to reach the target test accuracy. The target test accuracy is the lower final converged test accuracy with and without PFNC, indicated by an underscore.

		Yelp	ogbn-arxiv	ogbn-products
GAS	acc	40.18 ± 0.16	71.46 ± 0.24	76.01 ± 0.17
	epoch	488	244	235
	time (s)	518.30	44.19	428.40
GAS+**PFNC**	acc	40.64 ± 0.20 (**+0.46**)	71.71 ± 0.18 (**+0.25**)	76.25 ± 0.43 (**+0.14**)
	epoch	351 (↓ **28**%)	244 (↓ **0**%)	200 (↓ **15**%)
	time (s)	376.13 (↓ **27.4**%)	44.58 (↓ **−0.9**%)	365.32 (↓ **14.7**%)
GraphFM	acc	42.64 ± 0.60	71.41 ± 0.09	74.16 ± 0.20
	epoch	447	249	180
	time (s)	863.47	86.03	657.74
GraphFM+**PFNC**	acc	42.89 ± 0.32 (**+0.25**)	71.51 ± 0.13 (**+0.10**)	74.32 ± 0.35 (**+0.16**)
	epoch	439 (↓ **2**%)	206 (↓ **17**%)	128 (↓ **29**%)
	time (s)	845.34 (↓ **2.1**%)	72.20 (↓ **16.0**%)	481.87(↓ **26.7**%)
LMC	acc	41.64 ± 1.28	71.49 ± 0.10	74.20 ± 0.52
	epoch	498	296	246
	time (s)	519.41	44.61	520.86
LMC+**PFNC**	acc	42.12 ± 1.58 (**+0.48**)	71.58 ± 0.14 (**+0.09**)	74.66 ± 0.28 (**+0.46**)
	epoch	437 (↓ **12**%)	284 (↓ **4**%)	194 (↓ **21**%)
	time (s)	466.63 (↓ **10.2**%)	44.47 (↓ **0.3**%)	417.24 (↓ **19.9**%)
Average epoch reduction		**14.0%**	**7.0%**	**21.7%**
Average acceleration		**13.23%**	**5.13%**	**20.43%**

For the three gradient compensation methods, we use the official implementations and configurations from their official repositories[2][3][4]. For the standard neighbor sampling methods, we use the implementations and configurations provided in the DGL [34] library's example code.

5.1.3 Metrics. Our experiments evaluate the effectiveness of PFNC by comparing both the convergence speed and the final model performance before and after integrating PFNC into baseline methods. To establish a fair baseline, models are trained for a sufficient number of epochs to ensure convergence.

For assessing convergence speed, a metric similar to those used in existing studies [28] is adopted. Specifically, the model performance achieved by the baseline method at convergence serves as the target value. Under identical training conditions, the number of epochs required for both PFNC and the baseline method to first reach this target performance is compared.

For comparing final model performance, the models trained with the PFNC method and the baseline method are evaluated based on their performance at convergence after being trained for the same number of epochs.

5.1.4 Hyperparameters. We set α to 0.9 and K_g to $\min\{\mathsf{bn}, 16\}$, where bn represents the number of batches in one epoch. Additionally, we set β to 0.95 and K_h to $0.01N$, where N is the number of nodes in the graph. Other training configurations can be found in our artifact. Analysis of these hyperparameters are in Section 5.6.

5.1.5 Implementation Details. We implement our method as a package based on the PyTorch [25]. In addition, our hardware experiments are NVIDIA A100-SXM4-80GB and AMD EPYC 7413 24-Core.

[2]https://github.com/MIRALab-USTC/GNN-LMC
[3]https://github.com/rusty1s/pyg_autoscale
[4]https://github.com/divelab/DIG/tree/dig/dig/lsgraph

5.2 Comparison with Other Gradient Compensation Methods

In this section, we primarily investigate whether PFNC can further improve the gradient estimation compared to other existing gradient compensation methods, thereby enhancing the convergence speed and generalization performance of GNN models.

Existing gradient compensation methods typically rely on specially crafted subgraph-wise sampling techniques that include all 1-hop neighbors of training nodes. These methods explicitly require the historical embeddings of all 1-hop neighbors to engage in message passing, which cannot be omitted, necessitating the maintenance of historical embeddings for all nodes. Consequently, the historical embedding cache used by these methods cannot be replaced by our more economical version. However, our historical gradient cache, designed to enhance gradient composition, can be integrated with these methods to potentially provide additional benefits. We explore its impact in further reducing gradient estimation errors and improving convergence.

5.2.1 PFNC accelerates convergence without sacrificing accuracy. Table 4 presents a comparison of convergence accuracy and speed before and after integrating PFNC into the GAS, GraphFM, and LMC methods on four large datasets. The reported accuracy values are the mean and standard deviation across 5 runs. From the table, we can observe that adding PFNC on top of existing gradient compensation methods slightly improves convergence accuracy, increasing accuracy by up to 0.48.

For comparing convergence speed, we compare the number of epochs and training time required to first reach the target test accuracy. The target test accuracy is the convergence accuracies of baselines, indicated by an underline in the table. The training time to reach the target accuracy is calculated by multiplying the

Table 5: Comparison of small graph datasets. This table follows the same structure as Table 4.

		Cora	CiteSeer	PubMed
GAS	acc	82.16 ± 0.42	71.10 ± 0.83	78.96 ± 0.53
	epoch	190	119	200
	time (s)	2.74	1.74	2.82
GAS+**PFNC**	acc	82.26 ± 0.81 (**+0.10**)	71.61 ± 1.04 (**+0.51**)	79.08 ± 0.69 (**+0.08**)
	epoch	70 (↓ **63.2**%)	92 (↓ **22.7**%)	141 (↓ **29.5**%)
	time (s)	1.36 (↓ **50.4**%)	1.67 (↓ **4.2**%)	2.54 (↓ **10.0**%)

Table 7: PFNC performance on real-world large graph ogbn-papers100M.

		ogbn-papers100M
GraphSAGE	acc	61.19
	epoch	40
	time (s)	668.98
GraphSAGE+**PFNC**	acc	61.87 (**+0.68**)
	epoch	26 (↓ **35.0**%)
	time (s)	443.69 (↓ **33.7**%)

number of epochs by the time per epoch (averaged over 100 epochs). From the table, we can see that PFNC significantly accelerates convergence. On average, PFNC reduces the number of epochs to reach the target accuracy by 14.0%, 7.0%, and 21.7% on the Yelp, ogbn-arxiv, and ogbn-products datasets, respectively, and reduces time to first reach target test accuracy by 13.23%, 5.13%, and 20.43%. The average epoch reduction rate and the average acceleration rate are very close, indicating that the time per epoch does not change significantly when using PFNC, i.e. the computational overhead of PFNC is negligible. It is worth noting that in Table 4, for the GAS baseline on `ogbn-arxiv`, PFNC improves the final accuracy but does not reduce the number of epochs required to reach the target accuracy. This may be due to the characteristics of the `ogbn-arxiv` dataset (e.g., a lower average degree leading to smaller gradient variance), and the fact that GAS already partially mitigates message inaccuracies, leaving limited room for further acceleration by PFNC.

We also conducted similar experiments on small-scale graph datasets. Results are shown in Table 5. For small graphs, we only used GAS as the baseline method because the repositories for GraphFM and LMC do not provide configurations small graph datasets. As seen in the table, the performance of PFNC on small graphs is similar to that on large graph datasets, effectively accelerating convergence and improving accuracy. PFNC improves the convergence accuracy of GAS by 0.10, 0.51, and 0.08 on the Cora, CiteSeer, and PubMed datasets, respectively. Additionally, it reduces the number of epochs required for convergence by 63.2%, 22.7%, and 29.5%, and decreases time to first reach target test accuracy by 50.4%, 4.2%, and 10.0%, respectively. The result in Cora stands out compared to the other datasets, which may be attributed to the dataset's relatively simpler structure and lower complexity. This characteristic makes it easier to fit, and when combined with the reduction in bias and variance of gradient estimates achieved by PFNC, the acceleration in convergence becomes particularly pronounced.

5.2.2 Gradient Estimation Error Comparison. To further illustrate the improvement of PFNC on gradient estimation, we compare the L2 errors between the mini-batch gradients computed by each method and the exact full-batch gradients. Figure 8 shows the gradient estimation errors on the Pubmed, Cora, and Citeseer datasets, using GAS as the baseline, before and after applying PFNC. To ensure controlled experimental conditions, we first load the model parameters from the corresponding epoch's checkpoint in full-graph training before gradient computation, thereby eliminating interference from model parameter differences. The displayed errors are averaged across all batches at epochs 10, 20, 30, 40, and 50 to ensure statistical reliability. It is evident that PFNC consistently reduces the gradient estimation errors.

5.3 PFNC accommodates diverse neighbor sampling methods

As we mentioned in Section 5.2, existing gradient compensation methods are typically restricted to their specially crafted subgraph-wise sampling methods. A key advantage of PFNC is its flexibility to be used with various neighbor sampling methods. In this section, we compare PFNC with representative methods from three mainstream neighbor sampling categories: node-wise, layer-wise, and subgraph-wise. Specifically, the methods included in this comparison are GraphSAGE, LADIES, LABOR and Cluster-GCN.

We conducted experiments on the largest graph in the datasets used in this paper, ogbn-products, comparing convergence accuracy and the number of epochs required to reach the target accuracy. The results are shown in Table 6. PFNC effectively accelerates convergence across all three types of sampling methods. It reduces the number of epochs by 20.1%, 15.4%, 13.6% and 22.2% for GraphSAGE, LADIES, LABOR and Cluster-GCN respectively. Additionally, PFNC slightly improves convergence accuracy, with increases of 0.36, 0.15, 0.39 and 0.12 , respectively. These results demonstrate that PFNC can accommodate various neighbor sampling methods well.

5.4 Scalability

To show the real-world applicability of PFNC at scale, we have added an evaluation on the extremely large graph ogbn-papers100M, which includes over 100 million nodes and 1.6 billion edges, thereby pushing the scalability limits of GNN training. As described in Table 7, PFNC improves both convergence speed and accuracy on this massive graph: +0.68 accuracy gain, 35.0% fewer epochs, and 33.7% reduction in training time, using a 3-layer GraphSAGE with fanout [10,10,10] and learning rate 1×10^{-4}. These results demonstrate that PFNC remains effective and stable under large-scale settings, affirming its real-world applicability.

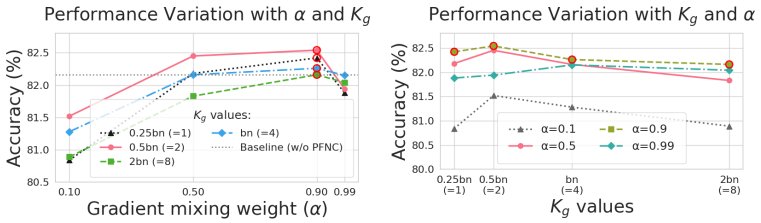
5.5 Ablation Study

5.5.1 Component Ablation. To evaluate the contributions of PFNC's two components (historical embedding cache and historical gradient cache), we conducted ablation experiments on the ogbn-products dataset using the GraphSAGE model. The experimental setup aligns with Section 5.3, and the results are summarized in Table 8. The results indicate that each component independently enhances both convergence speed and accuracy. However, their combination within the full PFNC framework yields the best performance, reducing the epochs required to achieve target accuracy by 20.1% and improving model accuracy by 0.36.

5.5.2 Cache Policy Ablation. We compare our importance-based eviction strategy with a naive First-In-First-Out (FIFO) policy under

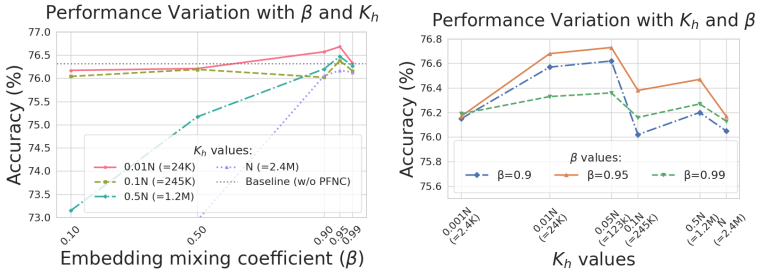
Table 6: The convergence accuracy of different neighbor sampling methods on node classification benchmarks.

Method Type			ogbn-products
Node-wise	GraphSAGE	acc	76.32 ± 0.33
		epoch	91
		time(s)	134.30
	GraphSAGE +**PFNC**	acc	76.68 ± 0.12 **(+0.36)**
		epoch	72 (↓ **20.1%**)
		time(s)	111.07 (↓ **17.3%**)
Edge-wise	LADIES	acc	77.85 ± 0.28
		epoch	97
		time(s)	1646.13
	LADIES +**PFNC**	acc	78.02 ± 0.14 **(+0.15)**
		epoch	82 (↓ **15.4%**)
		time(s)	1471.15 (↓ **10.6%**)
	LABOR	acc	79.19 ± 0.24
		epoch	22
		time(s)	159.31
	LABOR +**PFNC**	acc	79.58 ± 0.18 **(+0.39)**
		epoch	19 (↓ **13.6%**)
		time(s)	146.53 (↓ **8.0%**)
Subgraph-wise	ClusterGCN	acc	76.26 ± 0.21
		epoch	162
		time(s)	662.78
	ClusterGCN +**PFNC**	acc	76.38 ± 0.16 **(+0.12)**
		epoch	126 (↓ **22.2%**)
		time(s)	537.54 (↓ **18.9%**)



(a) Varying α, grouped by K_g. **(b) Varying K_g, grouped by α.**

Figure 9: Accuracy variation under different α and K_g configurations.



(a) Varying β, grouped by K_h. **(b) Varying K_h, grouped by β.**

Figure 10: Accuracy variation under different β and K_h configurations.

Table 8: Ablation experiment results on the ogbn-products dataset using the GraphSAGE model.

Configuration	Convergence Accuracy (%)	Epochs to Target Accuracy
No PFNC (Baseline)	76.32 ± 0.33	91
Only Hist. Emb. Cache	76.51 ± 0.28 **(+0.19)**	78 (↓ **14.3%**)
Only Hist. Grad. Cache	76.48 ± 0.10 **(+0.16)**	81 (↓ **11.0%**)
Full PFNC	76.68 ± 0.12 **(+0.36)**	72 (↓ **20.1%**)

Table 9: Comparison of importance-based vs. FIFO cache eviction strategies on ogbn-products.

K_h	0.001N	0.01N	0.05N	0.1N	0.5N	N
FIFO	76.17	76.02	76.26	76.14	76.21	**76.16**
Importance	**76.19**	**76.68**	**76.73**	**76.38**	**76.47**	76.15

varying cache sizes. As shown in Table 9, the importance-based strategy consistently outperforms FIFO in terms of final accuracy across a wide range of cache sizes. The performance gain is especially notable in the moderate cache regimes (e.g., $0.01N$ to $0.5N$), where eviction decisions have the most impact. These results empirically validate that our design better prioritizes the historical embeddings of nodes that are more critical for reducing gradient estimation error, leading to more efficient use of cache space and better model generalization.

5.6 Hyperparameter Analysis

5.6.1 α and K_g in Historical Gradient Cache. We investigate the impact of key parameters α (gradient mixing weight) and K_g (historical gradient cache size) on the performance of PFNC, using GAS with and without PFNC on the Cora dataset as an example. We varied α in the range [0.1, 0.5, 0.9, 0.99] and K_g in [0.25bn, 0.5bn, bn, 2bn], where bn represents the number of batches in one epoch.

The results for GAS with PFNC are presented in Figure 9, with the highest values for each K_g highlighted in red circle. For reference, the performance of GAS without PFNC is 82.16 and denoted as the dash line. Figure 9a shows that, with K_g fixed, the performance of PFNC initially improves and then declines as α increases. This trend can be attributed to the role of $1-\alpha$, which determines the weight of the historical gradients (Eq. 11). A smaller α leads to higher staleness, while a larger α may inadequately correct inaccuracies in the current gradient. Notably, $\alpha = 0.9$ yields the best performance in the evaluated hyperparameter range and is set as our default configuration. Regarding K_g, a larger value can introduce significant staleness to the compensation process, especially when α is small, leading to a noticeable decrease in performance. Therefore, we recommend keeping K_g at or below bn to ensure optimal performance. From Figure 9b, we observe that for all values of α, increasing K_g from 0.25bn to 0.5bn consistently improves accuracy. This suggests that maintaining a modest number of recent gradients provides stable gains. However, further increasing K_g to 1bn or 2bn leads to diminishing or slightly degraded performance, particularly for smaller α. This can be attributed to over-reliance on stale gradients. It also supports our design choice of restricting the historical gradient cache to the most recent K_g iterations.

5.6.2 β and K_h Historical Embedding Cache. We analyzed β (embedding mixing weight) and K_h (historical embedding cache size) using GraphSAGE with PFNC on the ogbn-products dataset. We varied β in [0.1, 0.5, 0.9, 0.95, 0.99] and K_h in [0.001N, 0.01N, 0.05N, 0.1N, 0.5N, N], where N is the number of nodes. In Figure 10a, as β decreases, $\frac{1+\beta}{1-\beta}$ shrinks, making the condition in Theorem 4.2

harder to satisfy. This increases gradient error when mixing with more historical embeddings, explaining the poor accuracy at $\beta = 0.1$ with full cache ($K_h = N$). However, our importance-based caching (Theorem 4.3) mitigates this worst-case scenario. At $K_h = 0.01N$, accuracy remains stable across β, outperforming full cache consistently, as selective retention of important embeddings filter out those nodes with high variance. We recommend $\beta = 0.95$ and $K_h = 0.01N$ for optimal, robust performance. In Figure 10b, we vary the embedding cache size K_h and group results by mixing weight β. With a properly chosen β, the performance remains stable across a wide range, and the highest accuracy is typically achieved with medium-sized caches (e.g., $0.01N \sim 0.05N$). When the cache is too small (e.g., $0.001N$), there is insufficient historical information to effectively reduce gradient estimation error, which leads to a slight performance drop. On the other hand, an excessively large cache may also slightly degrade performance due to the inclusion of unimportant or noisy historical embeddings.

5.6.3 Summary. These results demonstrate that while the exact choice of cache size affects peak performance, PFNC remains effective and stable across a reasonably well-defined range of cache sizes, particularly near the recommended defaults ($K_g = \min(\mathsf{bn}, 16)$, $\alpha = 0.9$; $K_h = 0.01N$, $\beta = 0.95$). This suggests that PFNC is not overly sensitive to cache hyperparameters, and can be used reliably without extensive tuning.

5.7 Memory Cost

To empirically validate the memory efficiency of PFNC, we conduct experiments to measure peak GPU memory usage during training. First, we compare the peak GPU memory consumption of PFNC with three representative baseline methods: GAS, GraphFM, and LMC. All methods are tested using the same batch size, and all caches are stored in GPU memory. As shown in Table 10, PFNC substantially reduces memory usage, requiring only 11.53 GB, in contrast to 19.58 GB (GAS), 30.57 GB (GraphFM), and 32.38 GB (LMC). Second, we measure the peak GPU memory usage of PFNC on the large-scale ogbn-papers100M dataset under varying cache sizes K_h. As shown in Table 11, PFNC remains highly memory-efficient even at industrial scales, requiring only 7.47 GB with $K_h = 0.01N$, and remains feasible up to $K_h = 0.2N$, whereas baseline methods encounter out-of-memory (OOM) errors due to their reliance on full caching. These confirm PFNC's practical memory advantage and scalability beyond the theoretical analysis discussed in Section 4.3.

Table 10: Peak GPU Memory Usage on ogbn-products.

Method	GAS	GraphFM	LMC	PFNC
Peak GPU Memory (GB)	19.58	30.57	32.38	**11.53**

Table 11: Peak GPU Memory Usage of PFNC on ogbn-papers100M with Varying Cache Sizes.

Cache Size K_h	$0.01N$	$0.1N$	$0.2N$	$0.3N$
Peak GPU Memory (GB)	7.47	33.61	63.07	OOM

6 CONCLUSION

In this paper, we address the critical challenge of gradient estimation errors in mini-batch GNN training with neighbor sampling. Existing methods predominantly focus on mitigating message inaccuracies while overlooking the impact of missing gradient contributions from unsampled nodes. To bridge this gap, we propose the Pseudo Full Neighborhood Compensation (PFNC) framework, which systematically compensates for both error sources through two novel components: (1) a selective historical embedding cache that corrects message inaccuracies with minimal memory overhead, and (2) a historical gradient cache that enhances gradient composition using stale gradients from previous iterations.

Theoretical analysis demonstrates that PFNC provides a better approximation to the ideal gradient by reducing both variance and bias in gradient estimation. Extensive experiments across six benchmark datasets validate that PFNC accelerates convergence compared to baseline neighbor sampling methods and improves test accuracy. Notably, PFNC achieves these improvements with only 1% of the memory required by existing gradient compensation methods, making it particularly suitable for large graphs. The framework's compatibility with diverse sampling strategies—including node-wise, layer-wise, and subgraph-wise approaches—further underscores its practical utility.

As future work, we plan to extend PFNC to dynamic graphs and heterogeneous GNN architectures. Since mini-batch training with neighbor sampling is also critical in these settings, the associated gradient estimation errors can similarly impact training. Additionally, we plan to investigate adaptive strategies for cache management and mixing coefficients to further optimize memory-performance tradeoffs. By addressing fundamental limitations in GNN training efficiency, this work paves the way for broader adoption of graph neural networks in large-scale industrial applications.

ACKNOWLEDGMENTS

Lei Chen's work is partially supported by National Key Research and Development Program of China Grant No. 2023YFF0725100, National Science Foundation of China (NSFC) under Grant No. U22B2060, Guangdong-Hong Kong Technology Innovation Joint Funding Scheme Project No. 2024A0505040012, the Hong Kong RGC GRF Project 16213620, RIF Project R6020-19, AOE Project AoE/E-603/18, Theme-based project TRS T41-603/20R, CRF Project C2004-21G, Key Areas Special Project of Guangdong Provincial Universities 2024ZDZX1006, Guangdong Province Science and Technology Plan Project 2023A0505030011, Guangzhou municipality big data intelligence key lab, 2023A03J0012, Hong Kong ITC ITF grants MHX/078/21 and PRP/004/22FX, Zhujiang scholar program 2021JC02X170, Microsoft Research Asia Collaborative Research Grant, HKUST-Webank joint research lab and 2023 HKUST Shenzhen-Hong Kong Collaborative Innovation Institute Green Sustainability Special Fund, from Shui On Xintiandi and the InnoSpace GBA. Yanyan Shen is supported by the National Key Research and Development Program of China (2024YFB4505203), Shanghai Municipal Science and Technology Major Project (2021SHZDZX0102), and SJTU Global Strategic Partnership Fund (2021 SJTU-HKUST).

REFERENCES

[1] Xin Ai, Qiange Wang, Chunyu Cao, Yanfeng Zhang, Chaoyi Chen, Hao Yuan, Yu Gu, and Ge Yu. 2024. NeutronOrch: Rethinking Sample-Based GNN Training under CPU-GPU Heterogeneous Environments. *Proc. VLDB Endow.* 17, 8 (April 2024), 1995–2008. https://doi.org/10.14778/3659437.3659453

[2] Muhammed Fatih Balin and Ümit Çatalyürek. 2023. Layer-Neighbor Sampling—Defusing Neighborhood Explosion in GNNs. *Advances in Neural Information Processing Systems* 36 (2023), 25819–25836.

[3] Chaoyi Chen, Dechao Gao, Yanfeng Zhang, Qiange Wang, Zhenbo Fu, Xuecang Zhang, Junhua Zhu, Yu Gu, and Ge Yu. 2023. NeutronStream: A Dynamic GNN Training Framework with Sliding Window for Graph Streams. *Proc. VLDB Endow.* 17, 3 (Nov. 2023), 455–468. https://doi.org/10.14778/3632093.3632108

[4] Fahao Chen, Peng Li, and Celimuge Wu. 2023. Dgc: Training dynamic graphs with spatio-temporal non-uniformity using graph partitioning by chunks. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–25.

[5] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247* (2018).

[6] Jianfei Chen, Jun Zhu, and Le Song. 2017. Stochastic training of graph convolutional networks with variance reduction. *arXiv preprint arXiv:1710.10568* (2017).

[7] 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 *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 257–266.

[8] Weilin Cong, Rana Forsati, Mahmut Kandemir, and Mehrdad Mahdavi. 2020. Minimal variance sampling with provable guarantees for fast training of graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1393–1403.

[9] Shaohua Fan, Junxiong Zhu, Xiaotian Han, Chuan Shi, Linmei Hu, Biyu Ma, and Yongliang Li. 2019. Metapath-guided heterogeneous graph neural network for intent recommendation. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2478–2486.

[10] Matthias Fey, Jan E Lenssen, Frank Weichert, and Jure Leskovec. 2021. Gnnautoscale: Scalable and expressive graph neural networks via historical embeddings. In *International conference on machine learning*. PMLR, 3294–3304.

[11] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. 2017. Protein interface prediction using graph convolutional networks. *Advances in neural information processing systems* 30 (2017).

[12] Shihong Gao, Yiming Li, Yanyan Shen, Yingxia Shao, and Lei Chen. 2024. Etc: Efficient training of temporal graph neural networks over large-scale dynamic graphs. *Proceedings of the VLDB Endowment* 17, 5 (2024), 1060–1072.

[13] Shihong Gao, Yiming Li, Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. 2024. Simple: Efficient temporal graph neural network training at scale with dynamic data placement. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–25.

[14] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[15] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. Ogb-lsc: A large-scale challenge for machine learning on graphs. *arXiv preprint arXiv:2103.09430* (2021).

[16] 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. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[17] Kezhao Huang, Haitian Jiang, Minjie Wang, Guangxuan Xiao, David Wipf, Xiang Song, Quan Gan, Zengfeng Huang, Jidong Zhai, and Zheng Zhang. 2024. FreshGNN: Reducing Memory Access via Stable Historical Embeddings for Graph Neural Network Training. *Proc. VLDB Endow.* 17, 6 (Feb. 2024), 1473–1486. https://doi.org/10.14778/3648160.3648184

[18] Tim Kaler, Alexandros-Stavros Iliopoulos, Philip Murzynowski, Tao B. Schardl, Charles E. Leiserson, and Jie Chen. 2023. Communication-Efficient Graph Neural Networks with Probabilistic Neighborhood Expansion Analysis and Caching. arXiv:2305.03152 [cs.LG] https://arxiv.org/abs/2305.03152

[19] Ruoyu Li, Sheng Wang, Feiyun Zhu, and Junzhou Huang. 2018. Adaptive graph convolutional neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.

[20] Yiming Li, Yanyan Shen, Lei Chen, and Mingxuan Yuan. 2025. A Caching-based Framework for Scalable Temporal Graph Neural Network Training. *ACM Transactions on Database Systems* 50, 1 (2025), 1–46.

[21] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. 2018. Combinatorial optimization with graph convolutional networks and guided tree search. *Advances in neural information processing systems* 31 (2018).

[22] Zhiyuan Li, Xun Jian, Yue Wang, Yingxia Shao, and Lei Chen. 2024. DAHA: Accelerating GNN Training with Data and Hardware Aware Execution Planning. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1364–1376.

[23] Yuxin Ma, Ping Gong, Tianming Wu, Jiawei Yi, Chengru Yang, Cheng Li, Qirong Peng, Guiming Xie, Yongcheng Bao, Haifeng Liu, and Yinlong Xu. 2024. Eliminating Data Processing Bottlenecks in GNN Training over Large Graphs via Two-level Feature Compression. *Proc. VLDB Endow.* 17, 11 (July 2024), 2854–2866. https://doi.org/10.14778/3681954.3681968

[24] Yuxin Ma, Ping Gong, Jun Yi, Zhewei Yao, Cheng Li, Yuxiong He, and Feng Yan. 2022. Bifeat: Supercharge gnn training via graph feature quantization. *arXiv preprint arXiv:2207.14696* (2022).

[25] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[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 (May 2022), 1937–1950. https://doi.org/10.14778/3538598.3538614

[27] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective classification in network data. *AI magazine* 29, 3 (2008), 93–93.

[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, Tianyi Li, Qing Sun, Yanfeng Zhang, Christian S Jensen, and Ge Yu. 2023. ADGNN: towards scalable GNN training with aggregation-difference aware sampling. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–26.

[30] Zhen Song, Yu Gu, Qing Sun, Tianyi Li, Yanfeng Zhang, Yushuai Li, Christian S Jensen, and Ge Yu. 2024. DynaHB: A Communication-Avoiding Asynchronous Distributed Framework with Hybrid Batches for Dynamic GNN Training. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3388–3401.

[31] Borui Wan, Juntao Zhao, and Chuan Wu. 2023. Adaptive message quantization and parallelization for distributed full-graph gnn training. *Proceedings of Machine Learning and Systems* 5 (2023), 203–218.

[32] Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. 2023. Scalable and efficient full-graph gnn training for large graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–23.

[33] Daixin Wang, Jianbin Lin, Peng Cui, Quanhui Jia, Zhen Wang, Yanming Fang, Quan Yu, Jun Zhou, Shuang Yang, and Yuan Qi. 2019. A semi-supervised graph attentive network for financial fraud detection. In *2019 IEEE International Conference on Data Mining (ICDM)*. IEEE, 598–607.

[34] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[35] Qiange Wang, Yao Chen, Weng-Fai Wong, and Bingsheng He. 2023. Hongtu: Scalable full-graph GNN training on multiple gpus. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–27.

[36] Qiange Wang, Yao Chen, Weng-Fai Wong, and Bingsheng He. 2025. Scalable and Load-Balanced Full-Graph GNN Training on Multiple GPUs. *IEEE Transactions on Knowledge and Data Engineering* (2025).

[37] 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 *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 974–983.

[38] Haiyang Yu, Limei Wang, Bokun Wang, Meng Liu, Tianbao Yang, and Shuiwang Ji. 2022. GraphFM: Improving large-scale GNN training via feature momentum. In *International Conference on Machine Learning*. PMLR, 25684–25701.

[39] Hao Yuan, Yajiong Liu, Yanfeng Zhang, Xin Ai, Qiange Wang, Chaoyi Chen, Yu Gu, and Ge Yu. 2023. Comprehensive evaluation of gnn training systems: A data management perspective. *arXiv preprint arXiv:2311.13279* (2023).

[40] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931* (2019).

[41] 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. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–24.

[42] Yuhao Zhang and Arun Kumar. 2023. Lotan: Bridging the gap between gnns and scalable graph analytics engines. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2728–2741.

[43] 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 *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 36–44.

[44] Qiqi Zhou, Yanyan Shen, and Lei Chen. 2023. Narrow the Input Mismatch in Deep Graph Neural Network Distillation. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 3581–3592. https://doi.org/10.1145/3580305.3599442

[45] 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. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2094–2105. https://doi.org/10.14778/3352063.3352127

[46] 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. *Advances in neural information processing systems* 32 (2019).

[47] 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 *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems, NeurIPS.*