FreshGNN: Reducing Memory Access via Stable Historical Embeddings for Graph Neural Network Training

Abstract

A key performance bottleneck when training graph neural network (GNN) models on large, real-world graphs is loading node features onto a GPU. Due to limited GPU memory, expensive data movement is necessary to facilitate the storage of these features on alternative devices with slower access (e.g. CPU memory). Moreover, the irregularity of graph structures contributes to poor data locality which further exacerbates the problem. Consequently, existing frameworks capable of efficiently training large GNN models usually incur a significant accuracy degradation because of the inevitable shortcuts involved. To address these limitations, we instead propose FreshGNN, a general-purpose GNN mini-batch training framework that leverages a historical cache for storing and reusing GNN node embeddings instead of re-computing them through fetching raw features at every iteration. Critical to its success, the corresponding cache policy is designed, using a combination of gradient-based and staleness criteria, to selectively screen those embeddings which are relatively stable and can be cached, from those that need to be re-computed to reduce estimation errors and subsequent downstream accuracy loss. When paired with complementary system enhancements to support this selective historical cache, FreshGNN is able to accelerate the training speed on large graph datasets such as `ogbn-papers100M` and `MAG240M` by 4.6× up to 23.6× and reduce the memory access by 64.5% (85.7% higher than a raw feature cache), with less than 1% influence on test accuracy.

1 Introduction

Graphs serve as a ubiquitous abstraction for representing relations between entities of interest. Linked web pages, paper citations, molecule interactions, purchase behaviors, etc., can all be modeled as graphs, and hence, real-world applications involving non-i.i.d. instances are frequently based on learning from graph data. To instantiate this learning process, Graph Neural Networks (GNN) have emerged as a powerful family of trainable architectures with successful deployment spanning a wide range of graph applications, including community detection [4], recommender systems [40], fraud detection [7], drug discovery [12] and more. The favorable performance of GNNs is largely attributed to their ability to exploit both entity-level features as well as complementary structural information or network effects via so-called *message passing* schemes [13], whereby updating any particular node embedding requires collecting and aggregating the embeddings of its neighbors. Repeatedly applying this procedure by stacking multiple layers allows GNN models to produce node embeddings that capture local topology (with extent determined by model depth) and are useful for downstream tasks such as node classification or link prediction.

Of course not surprisingly, the scale of these tasks is rapidly expanding as larger and larger graph datasets are collected. As such, when the problem size exceeds the memory capacity of hardware such as GPUs, some form of mini-batch training is the most common workaround [5, 14, 46, 47]. Similar to the mini-batch training of canonical i.i.d. datasets involving images or text, one full training epoch is composed of many constituent iterations, each optimizing a loss function using gradient descent w.r.t. a small batch of nodes/edges. In doing so, mini-batch training reduces memory requirements on massive graphs but with the added burden of frequent data movement from CPU to GPU. The latter is a natural consequence of GNN message passing, which for an L-layer model requires repeatedly loading the features of the L-hop neighbors of each node in a mini-batch. The central challenge of efficient GNN mini-batch training then becomes the mitigation of this data loading bottleneck, which otherwise scales exponentially with L; even for moderately-sized graphs this quickly becomes infeasible.

Substantial effort has been made to address the challenges posed by large graphs using system-level optimizations, algorithmic approximations, or some combination thereof. For example, on the system side, GPU kernels have been used to efficiently load features in parallel or store hot features in a GPU cache [26, 27, 43]; however, these approaches cannot avoid memory access to the potentially large number of nodes that are visited less frequently.

On the other hand, there are generally speaking two lines of work on the algorithm front. The first is based on devising sampling methods to reduce the computational footprint and the required features within each mini-batch. Notable strategies of this genre include neighbor sampling [14], layer-wise sampling [3, 50], and graph-wise sampling [5, 46]. However, neighbor sampling does not solve the problem of exponential growth mentioned previously, and the others may converge slower or to a solution with lower accuracy [49]. Meanwhile, the second line of work [2, 10, 23] stores intermediate node representations computed for each GNN layer during training as *historical embeddings* and reuses them later to reduce the need for recursively collecting messages from neighbors. Though conceptually promising and foundational to our work, as we will later show in Sec. 2.3, these solutions presently struggle to simultaneously achieve *both* high training efficiency and high model accuracy when scaling to large graphs, e.g., those with more than 10^8 nodes and 10^9 edges.

To this end, we propose a new mini-batch GNN training solution with system and algorithm co-design for efficiently handling large graphs while preserving predictive performance. As our starting point, we narrow the root cause of accuracy degradation when using historical embeddings to the non-negligible accumulation of estimation error between true and approximate representations computed using the history. As prior related work has no direct mechanism for controlling this error, we equip mini-batch training with a *historical embedding cache* whose purpose is to *selectively* admit accurate historical embeddings while evicting those likely to be harmful to model performance. In support of this cache and its attendant admission/eviction policy, we design a prototype system called FreshGNN: Reducing mEmory access via Stable Historical embeddings, which efficiently trains large-scale models with high accuracy. In realizing FreshGNN, our primary contributions are as follows:

- We design a novel mini-batch training algorithm for GNNs that achieves scalability without compromising model performance. This is accomplished through the use of a historical embedding cache, with a corresponding cache policy that adaptively maintains node representations (via gradient and staleness criteria to be introduced later) that are likely to be stable across training iterations. In this way, we can economize GNN mini-batch training while largely avoiding the reuse of embeddings that are likely to lead to large approximation errors and subsequently, poor predictive accuracy.

- We create the prototype FreshGNN system to realize the above algorithm, with efficient implementation of the requisite cache operations, subgraph pruning, and data loading for both single-GPU and multi-GPU hardware settings.

- We provide a comprehensive empirical evaluation of FreshGNN across common baseline GNN architectures, large-scale graph datasets, and hardware configurations. Among other things, these results demonstrate that FreshGNN can closely maintain the accuracy of non-approximate neighbor sampling while training 4.6× up to 23.6× faster than state-of-the-art baselines.

2 Background and Motivation

This section introduces GNN basics and specifics of the related scalability challenge that we intend to solve.

2.1 Graph Neural Networks

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph with node set $\mathcal{V}$ and edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, where $n = |\mathcal{V}|$. Furthermore, let $A \in \{0,1\}^{n \times n}$ be the graph adjacency matrix such that $A_{uv} = 1$ if and only if there exists an edge between nodes u and v. Finally, $X \in \mathbb{R}^{n \times d}$ denotes the matrix of d-dimensional node features (i.e., each row is formed by the feature vector for a single node) while $Y \in \mathbb{R}^{n \times c}$ refers to the corresponding node labels with c classes.

Given an input graph defined as above, the goal of a GNN model is to learn a representation h_v for each node v, which can be used for downstream tasks such as node classification or link prediction. This is typically accomplished via a so-called *message passing* scheme [13]. For the $(l+1)$-th GNN layer, this involves computing the hidden/intermediate representation

$$\begin{aligned} h_v^{(l+1)} &= f_W^{(l+1)}\left(h_v^{(l)}, \left\{h_u^{(l)} : u \in \mathcal{N}(v)\right\}\right) \\ &= \phi_W^{(l+1)}\left(h_v^{(l)}, \underset{u \in \mathcal{N}(v)}{\mathrm{AGG}}\left(\psi_W^{(l+1)}\left(h_u^{(l)}, h_v^{(l)}\right)\right)\right), \end{aligned} \tag{1}$$

where $h_v^{(l)}$ and $\mathcal{N}(v)$ are the embedding and neighbors of node v respectively, with $h_v^{(0)}$ equal to the v-th row of X. Additionally, ψ computes messages between adjacent nodes while the operator AGG is a permutation-invariant function (like sum, mean, or element-wise maximum) designed to aggregate these messages. Lastly, ϕ represents an update function that computes each layer-wise embedding. Note that both ϕ and ψ are parameterized by a learnable set of weights W. Within this setting, the goal of training an L-layer GNN is to minimize an application-specific loss function $\mathcal{L}(H^{(L)}, Y)$ with respect to W. This can be accomplished via gradient descent as $W \leftarrow W - \eta \nabla_W \mathcal{L}$, where η is the step size.

2.2 Difficulty in Training Large-Scale GNNs

Graphs used in GNN training can have a large number of nodes containing high-dimensional features (e.g., $d \in \{100, \ldots, 1000\}$) [17,44]. As a representative example, within the widely-adopted Open Graph Benchmark (OGB) [16,

17], the largest graph MAG240M has 240M nodes with 768-dimensional 16-bit float vectors as features (i.e. 350GB total size); real-world industry graphs can be much larger still. On the other hand, as nodes are dependent on each other, full graph training requires the features and intermediate embeddings of all nodes to be simultaneously available for computation, which goes beyond the memory capacity of a single device.

In light of this difficulty with full graph training, the most widely accepted workaround is to instead train with stochastic mini-batches [11, 14, 34, 43, 44]. At each iteration, instead of training all the nodes, a subset/batch are first selected from the training set. Then, the multiple-hop neighbors of these selected nodes (1-hop for each network layer) are formed into the subgraph needed to compute a forward pass through the GNN and later to back-propagate gradients for training. Additionally, further reduction in the working memory requirement is possible by sampling these multi-hop nodes as opposed to using the entire neighborhood [14]. The size of the resulting mini-batch subgraph with sampled neighborhood is much smaller relative to the full graph, and for some graphs can be trained using a single device achieving competitive accuracy [45] and generality [14]. Hence mini-batch training with *neighbor sampling* has become a standard paradigm for large-scale GNN training.

And yet even this standard paradigm is limited by a significant bottleneck, namely, loading the features of sampled multiple-hop neighbors in each mini-batch, which still involves data movement growing exponentially with the number of GNN layers. For example, on MAG240M, training a 3-layer GNN requires that, for each epoch we copy $\sim$1.5TB data from CPU to GPU. In many such cases, this step can occupy more than 85% of the total training time.

2.3 Existing Mini-Batch Training Overhauls

Both system-level and algorithmic approaches have been pursued in an attempt to alleviate the aforementioned limitations of mini-batch training.

System Optimizations. GNNLab [43] and GNNTier [26] cache the features of frequently visited nodes as assessed by metrics such as node degree, weighted reverse pagerank, and profiling to GPU. However, for commonly-encountered graphs exhibiting a power-law distribution [8], most of the nodes will experience a moderate visiting frequency and hence, the feature cache is unlikely to reduce memory access to them. More generally, because the overall effectiveness of this approach largely depends on graph structure, consistent improvement across graph datasets is difficult to guarantee. PyTorch-Direct [27] proposes to store node features in CUDA Universal Virtual Addressing (UVA) memory so that feature loading can be accelerated by GPU kernels, but the data transfer bandwidth is still limited by PCIe. Even so, data loading remains a bottle-neck with PyTorch-Direct, occupying 66% of the total execution time.

Broader Sampling Methods. While neighbor sampling reduces the size of each mini-batch subgraph, it does not completely resolve recursive, exponential neighbor expansion. Consequently, alternative sampling strategies have been proposed such as layer-wise [3, 50] and graph-wise sampling [5, 46]. However, the resulting impact on model accuracy is graph dependent and prior evaluations [49] on large graphs like ogbn-papers100M indicate that a significant performance degradation may at times occur.

Reusing Historical Intermediate Embeddings. The other line of algorithmic work [2, 10, 23] for revamping mini-batch training approximates the embeddings of some node set $\mathcal{S}$ with their historical embeddings from previous training iterations. This involves modifying the original message passing scheme from Equation (1) to become

$$\begin{aligned} h_v^{(l+1)} &= f_W^{(l+1)}\left(h_v^{(l)}, \left\{h_u^{(l)}\right\}_{u\in\mathcal{N}(v)}\right) \\ &= f_W^{(l+1)}\left(h_v^{(l)}, \left\{h_u^{(l)}\right\}_{u\in\mathcal{N}(v)\setminus\mathcal{S}} \cup \left\{h_u^{(l)}\right\}_{u\in\mathcal{N}(v)\cap\mathcal{S}}\right) \\ &\approx f_W^{(l+1)}\left(h_v^{(l)}, \left\{h_u^{(l)}\right\}_{u\in\mathcal{N}(v)\setminus\mathcal{S}} \cup \underbrace{\left\{\bar{h}_u^{(l)}\right\}_{u\in\mathcal{N}(v)\cap\mathcal{S}}}_{\text{Historical embeddings}}\right). \end{aligned} \tag{2}$$

The above computation is mostly the same as the original, except that now the node embeddings from $\mathcal{S}$ are replaced with their historical embeddings $\bar{h}_u^{(l)}$. A typical choice for $\mathcal{S}$ is any node not included within the current mini-batch, and after each training step, the algorithm will refresh $\bar{h}_v^{(l+1)}$ with the newly generated embedding $h_v^{(l+1)}$. Using historical embeddings avoids recursive aggregation of neighbor embeddings as well as the corresponding backward propagation, which reduces not only the number of raw features to load but also the computation associated with neighbor expansion. Moreover, this training methodology is compatible with arbitrary message passing architectures.

While promising, there remain two major unresolved issues with existing methods that utilize historical embeddings. First, they all require an extra storage of size $O(Lnd)$ for an L-layer GNN, which can be even larger than the total size of node features, a significant limitation. Secondly, historical embeddings as currently used may introduce impactful estimation errors during training. To help illustrate this point, let $\tilde{h}_v^{(l+1)}$ denote the approximated node embedding computed using Equation (2). The estimation error can be quantified by $\|\tilde{h}_v^{(l+1)} - h_v^{(l+1)}\|$, meaning the difference between the approximated embeddings and the authentic embeddings computed via an exact message passing scheme.

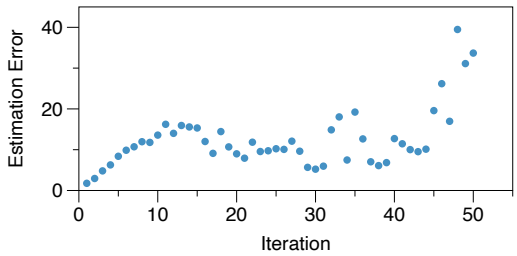


Figure 1: Average estimation error in one training epoch for GAS [10] on `ogbn-products`.

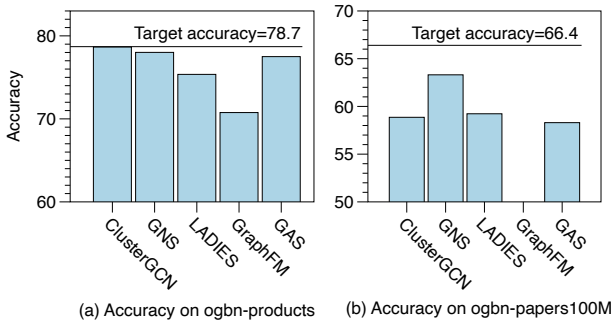


Figure 2: Test accuracy of various mini-batch training algorithms compared with the target accuracy achieved by (expensive) neighbor sampling on: (a) relatively small `ogbn-products` graph where the gap is modest, and (b) larger `ogbn-papers100M` graph where the gap grows significantly.

Figure 1 shows that the estimation error of each mini-batch increases considerably with more training iterations on the `ogbn-products` graph from OGB when using GNNAutoScale (GAS) [10], a representative system based on historical embeddings. The root cause of this problem is that existing methods lack a mechanism for directly controlling the quality of the cached embeddings used for message passing. And as the model parameters are updated by gradient descent after each iteration, the un-refreshed embeddings may gradually drift away from their authentic values resulting in a precipitous accuracy drop compared with the target accuracy from mini-batch training with vanilla/canonical neighbor sampling.

For this reason there remains ample room for new system designs that exploit historical embeddings in a more nuanced way so as to maintain accuracy. This is especially true as graph size grows larger and the performance of existing methods for scalable mini-batch training begins to further degrade as shown in Figure 2.

3 Overview of FreshGNN System

In this work, we propose a new strategy for utilizing historical embeddings, with targeted control of the resulting estimation error to economize GNN mini-batch training on large graphs without compromising accuracy. Intuitively, this strategy is designed to favor historical embeddings with small $\|\bar{h}_u^{(l)} - h_u^{(l)}\|$ when processing each mini-batch, while avoiding the use of those that have drifted away. However, directly computing this error requires knowledge of the authentic embeddings $h_u^{(l)}$, which is only obtainable via exact message passing, the very process we are trying to avoid. We thus adopt an alternative strategy based on a critical observation about the dynamics of the node embeddings during GNN mini-batch training: *most of the node embeddings experience only modest change across the majority of iterations*.

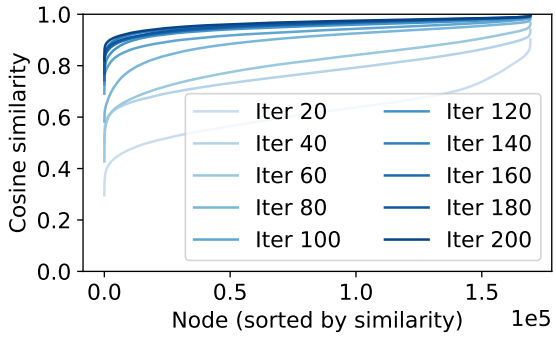


Figure 3: Distribution of cosine similarity between the node embeddings at iteration t and the embeddings at iteration $t-s$ during the mini-batch training of a GraphSAGE model on `ogbn-products`. Here $s = 20$.

Embedding Stability Illustration. Figure 3 showcases this phenomenon using a GCN [20] model on the `ogbn-products` dataset. We measure the cosine similarity of the node embeddings at mini-batch iteration t with the corresponding set at $t-s$ for lag $s = 20$ and plot the resulting distribution across varying t. We find that >78% of the nodes embeddings have >0.95 cosine similarity after iteration 140 (the model converged at around iteration 500). This observation suggests that there exists temporal stability of many node embeddings during GNN mini-batch training, and naturally motivates the use of a *historical embedding cache* capable of selectively storing and reusing such durable node representations.

FreshGNN Training Algorithm. With this in mind, Algorithm 1 shows our mini-batch training process using the historical embedding cache. For each batch, we first generate a subgraph according to a user-defined sampling method (Line 5). A subgraph consisting of the sampled L-hop neighbors for the training nodes is returned, where L is defined by whatever GNN model is chosen. Then for each layer, the nodes in the subgraph are divided into two types: normal nodes ($\mathcal{V}_{normal}$ on Line 9) and those nodes whose embeddings

Algorithm 1 Mini-Batch Training w/ Historical Emb. Cache

1: **Input**: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, input node features $H^{(0)}$, number of batches B, number of layers L, historical embedding cache $\mathcal{C}$, rate for check-out using gradient p_{grad}, the maximum staleness threshold t_{stale}
2: $i \leftarrow 0$ ▷ Iteration ID
3: $\{\mathcal{B}_1, \cdots, \mathcal{B}_B\} \leftarrow Split(\mathcal{G}, B)$ ▷ Mini-batches of training nodes
4: **for** $\mathcal{B}_b \in \{\mathcal{B}_1, \cdots, \mathcal{B}_B\}$ **do**
▷ Subgraph generator
5: $\mathcal{G}_b \leftarrow sample(\mathcal{G}, \cup_{v \in \mathcal{B}_b})$
6: **for** $l \in \{L-1, \cdots, 1\}$ **do**
7: $\mathcal{V}^{(l)}_{cache} \leftarrow \mathcal{V}^{(l)}_{\mathcal{G}_b} \cap \mathcal{C}^{(l)}$ ▷ History index lookup
8: $\mathcal{G}_b.remove_subgraph(root = \mathcal{V}^{(l)}_{cache})$
▷ Remove nodes for the calculation of $\mathcal{V}^{(l)}_{cache}$
9: $\mathcal{V}^{(l)}_{normal} \leftarrow \mathcal{V}^{(l)}_{\mathcal{G}_b} \setminus \mathcal{V}^{(l)}_{cache}$
10: **end for**
▷ Data loader
11: $H^{(0)}_{normal} \leftarrow Load_feature(\mathcal{V}^{(0)}_{normal})$
12: **for** $l \in \{1, ..., L\}$ **do**
▷ Historical embedding cache
13: $h^{(l)}_v \leftarrow \mathcal{C}^{(l)}[v], \forall v \in \mathcal{V}^{(l)}_{history}$
14: $h^{(l)}_v \leftarrow f^{(l)}_W\left(h^{(l-1)}_v, \left\{h^{(l-1)}_u\right\}_{u \in \mathcal{N}_{\mathcal{G}_b}(v)}\right), \forall v \in \mathcal{V}^{(l)}_{normal}$
15: **end for**
16: $loss = \mathcal{L}(h^{(L)}_{\mathcal{B}_b}, label_{\mathcal{B}_b})$
17: $loss.backward()$
18: $i \leftarrow i + 1$
19: **for** $l \in \{1, \cdots, L\}$ **do**
20: UPDATE$(\mathcal{C}^{(l)}, \mathcal{V}^{(l)}_{normal}, \mathcal{V}^{(l)}_{\mathcal{G}_b}, h^{(l)}, i, t_{stale}, p_{grad})$
21: **end for**
22: **end for**

can be found in the historical cache ($\mathcal{V}_{cache}$ on Line 7). For the latter, embeddings are directly read from the cache (Line 13), while for the normal nodes, embeddings are computed by the neighbor aggregation assumed by the GNN (Line 14). And finally, at the end of each iteration, the algorithm will utilize information from the forward and backward propagation steps to update the historical embedding cache. The goal is to check in stable embeddings that are more reliable for future reuse while checking out those that are not (Line 20).

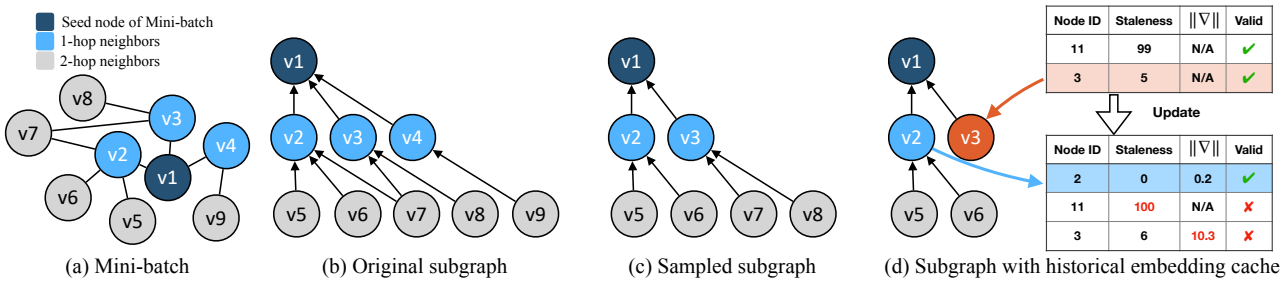
Figure 4 further elucidates Algorithm 1 using a toy example. Here node $v1$ is selected as the seed node of the current mini-batch as in Figure 4(a). Computing $v1$'s embedding requires recursively collecting information from multi-hop neighbors as illustrated by the subgraph in Figure 4(b). And as mentioned previously, neighbor sampling can reduce this subgraph size as shown in Figure 4(c). Figure 4(d) then depicts how the historical embedding cache can be applied to further prune the required computation and memory access. The cache contains node embeddings recorded from previous iterations as well as some auxiliary data related to staleness and gradient magnitudes as needed to estimate embedding stability. In this example, the embeddings of node $v3$ are found in the cache, hence its neighbor expansion is no longer needed and is pruned from the graph. Additionally, after this training iteration, some newly generated embeddings (e.g., node $v2$) will be pushed to the cache for later reuse. Existing cached embeddings may also be evicted based on the updated metadata. In this example, both $v11$ (by staleness) and $v3$ (by gradient magnitude criteria to be detailed later) are evicted from the cache.

Main FreshGNN Components. To instantiate Algorithm 1, and enable accurate, efficient GNN training over large graphs in CPU-GPU or multi-GPU scenarios, we require three novel system components, namely, (i) the historical embedding cache, (ii) the subgraph generator, and (iii) the data loader (which correspond with the colored blocks in Algorithm 1). Figure 5 situates these components within a typical GNN training workflow while supporting details are as follows:

i The **historical embedding cache** is the central component of our FreshGNN design, selectively storing node embeddings in GPU and providing efficient operations for fetching or updating its contents as will be detailed in Section 4. In brief here, we propose two cache policies (based on embedding staleness and gradient magnitudes collected during training) to detect and admit only those node embeddings that are likely to be relatively stable while evicting those that are not.

ii The **subgraph generator** is responsible for producing a pruned subgraph given the current mini-batch as will be discussed further in Section 5. Compared with other GNN systems, a unique characteristic of our proposed approach is that each subgraph structure itself is adaptively generated based on which particular nodes happen to be in the selective historical embedding cache, and the latter is dynamically updated at the end of each training iteration. This essentially creates a reversed data dependency between the stages of mini-batch preparation and mini-batch training, making it difficult to apply pipelining to overlap the two stages like in other GNN systems [28, 43]. To address this challenge, we further partition the workload into two steps: graph sampling and graph pruning, where only the latter step depends on the historical embedding cache. We then adopt a mixed CPU-GPU design that samples graphs in CPU while pruning graphs in GPU; this is complemented by a GPU-friendly data structure for fast graph pruning.

iii The **data loader** is in charge of loading the relevant node



(a) Mini-batch (b) Original subgraph (c) Sampled subgraph (d) Subgraph with historical embedding cache

Figure 4: Illustration of historical embedding cache using an example mini-batch graph.

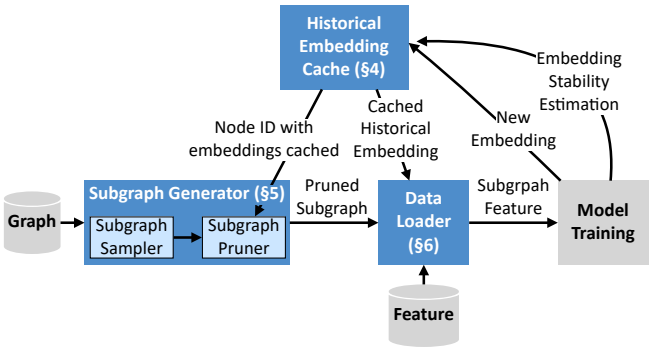


Figure 5: FreshGNN system workflow.

features or intermediate embeddings given a generated subgraph. For each node that is not cached or pruned using historical embeddings, the data loader fetches its raw features. Since these features are typically stored in a slower but larger memory device, FreshGNN further optimizes the data transmission for three scenarios: (1) fetching features from CPU to a single GPU, (2) fetching features from CPU to multiple GPUs, and (3) fetching features from other GPUs. See Section 6 for further details.

4 Historical Embedding Cache

The historical embedding cache design is informed by the following two questions: (1) *What are suitable cache policies for checking in and out node embeddings?* and (2) *What is the best way to implement the historical emebedding cache on GPU efficiently?* We address each in turn below.

4.1 Cache Policy

Caching intermediate node embeddings for later reuse is fundamentally more complicated than caching raw node features because the former are constantly updated during model training. This calls for a cache policy to invalidate unstable entries, where in the context of GNN training, this notion of stability is calibrated by the difference between a true node embedding and the corresponding cached one. Therefore, a naive solution is to recompute all the embeddings in the cache after each training iteration and evict those that have drifted away. However, this solution is of course not viable for large graphs because it involves the very type of high-cost computation we are trying to avoid.

Instead, FreshGNN employs a cache policy based on dynamic feedback from model training. Given a mini-batch graph, denote the set of nodes at layer l as $\mathcal{V}^{(l)}$ and the set of cached nodes as $\mathcal{V}_{cache}$. We then apply two cache policy criteria to estimate the stability of an embedding as follows:

Gradient-based Criteria. For nodes $v \in \mathcal{V}^{(l)}$, we use the magnitude of embedding gradients w.r.t. the training loss as a proxy for node stability at each layer, noting that for a gradient magnitude near zero, there is little impact on the training loss for any downstream gradient signals passing through this particular node at a given layer (which justifies our efficient subgraph pruning step discussed later in Section 5). Moreover, consistently small gradient magnitudes are likely to correlate on average with smaller estimation errors $\|\bar{h}_v^{(l)} - h_v^{(l)}\|$. Hence, FreshGNN admits nodes with small gradients to the cache, with the rate controlled by p_{grad}, the fraction of newly generated embeddings to be admitted. Of the remaining $(1 - p_{grad})$ fraction of the nodes in the mini-batch, if any of these happen to already be in the cache, they will now be evicted.

Conveniently, embedding gradients at any layer are naturally obtained from the backward propagation of the model as illustrated in Figure 6 without computation overhead. In this example, at layer 1, Node 3 fetches its embedding from the cache while Node 2 computes its embedding faithfully by aggregating from its neighbors. During backward propagation, FreshGNN calculates both gradients $\nabla_{h_3^{(1)}} \mathcal{L}$ and $\nabla_{h_2^{(1)}} \mathcal{L}$, and compares their vector norms to decide which to admit or evict; here the cache decides to admit Node 2 while evicting Node 3.

Staleness-based Criteria. Because fresh embedding gradients are not available at a given iteration for nodes $v \in \mathcal{V}_{cache} \setminus \mathcal{V}^{(l)}$, we instead choose to bound their *staleness*. The staleness is set to be zero when an embedding is admitted to the cache and will increment by one at each iteration.

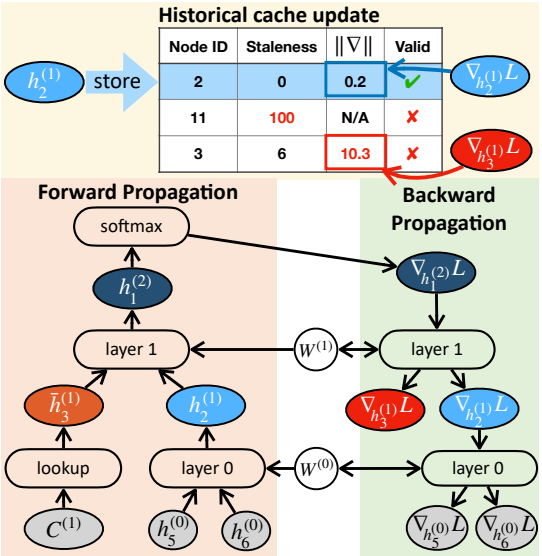


Figure 6: Illustration of the admission/eviction policy of the historical embedding cache. After a training iteration, the cache admits the embedding of Node 2 but evicts Node 3 (which was originally in the cache at the start of the iteration) by the gradient-based criteria. It also evicts Node 11 according to the staleness criteria.

FreshGNN treats any embedding with staleness larger than a threshold t_{stale} as being out-dated and evicts it from the cache. In Figure 6, Node 11 is evicted per this criteria. More broadly, utilizing and limiting staleness is a commonly used technique in training neural networks; the difference between FreshGNN and prior work that incorporates staleness criteria during training is discussed in Section 8.

Resulting Adaptive Cache Size. A notable difference from typical cache usage, which emerges from the above two criteria, is that our historical embedding cache size is not static or preset, but rather implicitly controlled by the two thresholds p_{grad} and t_{stale}. Larger p_{grad} or t_{stale} means more embeddings to be cached but also requires the model to tolerate larger approximation errors introduced by historical embeddings. Moreover, setting $p_{grad} = 0$ or $t_{stale} = 0$ degrades the algorithm to normal neighbor sampling without a cache. In contrast, setting $p_{grad} = 1$ and $t_{stale} = \infty$ results in a policy that is conceptually equivalent to that used by GAS [10] and VR-GCN [2]. So in this sense FreshGNN is a more versatile paradigm w.r.t. previous historical embedding based methods. Later in Section 7 we demonstrate that finding suitable values of p_{grad} and t_{stale} is relatively easy in practice; however, we leave open to future research the possibility of further exploiting this flexible dimension within the FreshGNN design space.

4.2 GPU Implementation

FreshGNN utilizes GPU threads to look up a batch of historical node embeddings in parallel. To achieve this, a node ID mapping array of length $O(|\mathcal{V}|)$ is maintained, where each entry stores the index to the cache table that holds the embedding of the corresponding node. The extra storage of this mapping array is affordable compared with the storage for embeddings. For example, storing the ID mapping array for the 100M nodes of `ogbn-papers100M` requires only 400MB. Given a batch of node IDs, each thread is in charge of fetching one node embedding and each fetching operation can be done in $O(1)$.

FreshGNN also adopts a ring buffer design to efficiently admit new embeddings and evict old ones. As shown in Figure 7, this design maintains an embedding cache header that initially points to the first row of the cache table. New embeddings are added at the header and push the header towards the end of the table. At every t_{stale} iteration, FreshGNN moves the embedding cache header back to the beginning so that the newly added embeddings will overwrite the out-dated ones, which naturally evicts them from the cache. In some corner cases, the newly added embeddings may overwrite the embeddings that are still below the staleness bound. In practice, this rarely happens because the number of recorded embeddings per iteration is similar given that neighbor sampling tends to create mini-batch graphs with similar degrees. Additionally, to evict the embeddings with large gradient magnitudes, FreshGNN simply invalidates the corresponding entries of the node ID mapping array instead of conducting physical deletion. Those invalid slots will be recycled naturally by the ring buffer design.

Since the size of historical embedding cache cannot be determined a priori, we initialize the cache table with a fixed size and reallocate it on-demand. To further increase the lookup efficiency, we combine it with a raw feature cache by filling the empty entries with raw features of high-degree nodes. We further put the features of the highest degree at the end of the embedding table so that as new embeddings are added at the embeddding cache header, less frequent features will be overwritten first.

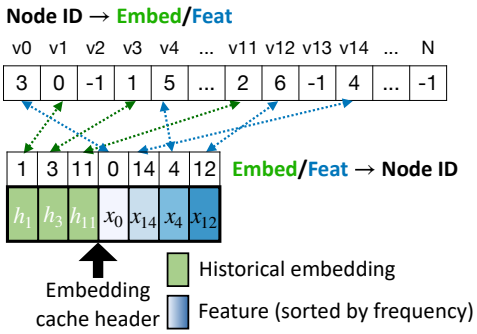


Figure 7: GPU implementation of historical embedding cache.

4.3 Remarks on FreshGNN Convergence

Not surprisingly, formal convergence analysis of our selective history cache and attendant check-in/out policy is generally infeasible when applied to arbitrary multi-layer, non-convex GNN architectures. That being said, in the special case of a single-layer simple graph convolution (SGC) model [39], it can be shown that a version of our approach defaults to the same convergence guarantees as stochastic gradient descent (SGD):

Proposition 4.1. ***(informal version; see supplementary for formal statement and proof)*** *Define the SGC model* $Z = \hat{A}^k XW$ *and training loss* $\ell(W) = \sum_{i=1}^{n} g(W; z_i, y_i)$*, where* g *is a Lipschitz* L*-smooth function of* W *and* y_i *is the label associated with* z_i*, the* i*-th row of* Z*. Then for a historical model with a random selector and bounded staleness for the usage of historical embeddings, updating the weights using gradient descent with step-size* $\eta \leq \frac{1}{L}$ *will converge to a stationary point of* $\ell(W)$*.*

Of course, this regime is admittedly quite limited, and not representative of more nuanced behavior involving the coupling of node embeddings of varying degrees of staleness across different layers. Although some prior analysis of gradient descent with stale features and gradients does exist [2, 32, 36], the setting is decidedly much simpler and does not directly apply to our complex modeling framework; however, we can consider potential adaptations in future work. As a point of reference though, most GNN models with adjustments for scalability (e.g., various sampling methods) do not actually come with any rigorous convergence guarantees.

5 Cache-Aware Subgraph Generator

In FreshGNN, the mini-batch subgraphs are generated adaptively according to the node embeddings stored in the cache. As the cache is in turn updated at the end of each iteration, this prevents FreshGNN from adopting a naive pipelining strategy to parallelize subgraph generation and model training as in other systems. To address this challenge, we decompose subgraph generation into two steps: graph sampling and graph pruning, where only the latter depends on the historical embedding cache. We then adopt a mixed CPU-GPU design to further accelerate them.

Asynchronous CPU Graph Sampling. This step first extracts/samples subgraphs normally for the given mini-batch and then moves them to GPU without querying information from the cache. As a result, graph sampling can be conducted asynchronously with the later GPU computation. We further utilize multithreading instead of multiprocessing to produce multiple subgraphs concurrently in contrast to the existing systems like DGL [37] and PyG [9]. Additionally, we use a task queue to control the production of subgraphs and avoid overflowing the limited GPU memory.

GPU Graph Pruning. The graph pruning step scans the mini-batch graph from the seed node layer to the input node layer. For any cached node, it recursively removes all the multiple-hop neighbors so that the corresponding computation is no longer needed for model training. The remaining challenge is that traditional sparse formats are not suitable for parallel modification in GPU. For example, the Coordinate (COO) format represents a graph using two arrays containing the source and destination node IDs of each edge. Pruning in-coming neighbors of a node in COO requires first locating neighbors using a binary search and then deleting them from both arrays. While for the Compressed Sparse Row (CSR) format, which further condenses one of the arrays to a row index array using prefix summation, locating a node's neighbors can be done in $O(1)$. However, after deleting the edges the row index array needs to be adjusted accordingly. The respective complexities required for these operations are listed in Table 1.

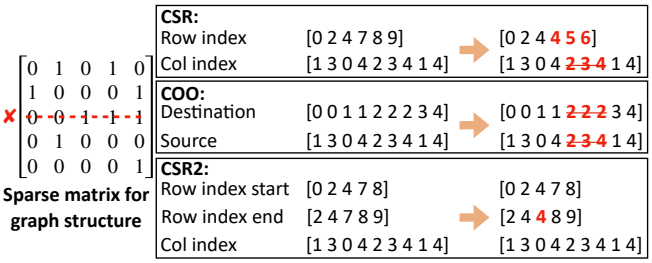


Figure 8: Removing a center node's neighbors using different graph data structures

For FreshGNN, we designed a novel data structure called CSR2 for even faster graph pruning on GPU. CSR2 uses two arrays to represent row indices – the first array records the starting offset of a node's neighbors to the column index array while the second array records its ending offset. As illustrated in Figure 8, for node i, its neighbors are stored in the column index segment starting from $start[i]$ to $end[i]$. The CSR2 format has some redundancy in storing a graph since $start[i]$ equals $end[i-1]$ for any node i. To remove a node's neighbors, one can simply set the corresponding $end[i] = start[i]$ without any changes to the column index array, resulting in an $O(1)$ operation. The data structure is also suitable for parallel processing on GPU because there is no data race condition. See Table 1 for complexity comparisons with prior formats.

6 Data Loader

Once a subgraph has been pruned, GNN training still needs the features of the remaining nodes, a task performed by the

Table 1: Storage complexity and the complexity for pruning the neighbors of one node in different sparse formats. $N_{neighbors}$ is the number of neighbors of the node to be pruned.

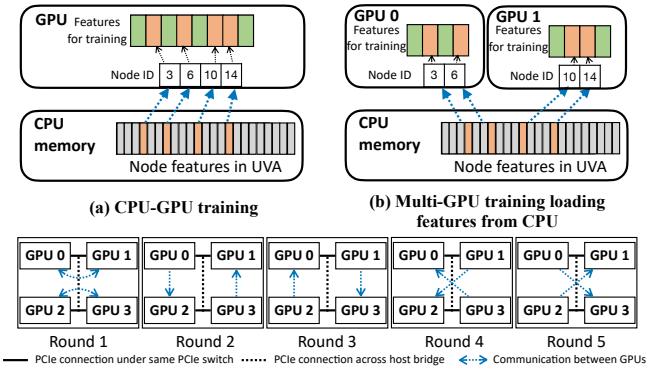
	Prune Complexity	Storage Complexity
CSR	$O(\lvert\mathcal{V}_{sample}\rvert + N_{neighbors})$	$O(\lvert\mathcal{V}_{sample}\rvert + \lvert\mathcal{E}_{sample}\rvert)$
COO	$O(\log(\lvert\mathcal{E}_{sample}\rvert) + N_{neighbors})$	$O(2\lvert\mathcal{E}_{sample}\rvert)$
CSR2 (ours)	$O(1)$	$O(2\lvert\mathcal{V}_{sample}\rvert + \lvert\mathcal{E}_{sample}\rvert)$

FreshGNN data loader. Note that even though the number of nodes in the pruned subgraph is largely reduced, the corresponding features are typically stored in a device with slower but larger memory, and the loading process is still critical to performance.

However, the indices for unpruned nodes and their features reside in different devices. The index is calculated in the GPU (computation device) while the needed features are stored in the CPU or remote GPUs (storage device). Under such conditions, the naive way to fetch the features is via **two-sided** communication. This involves first transferring node indices from the computation device to the storage device, compacting the corresponding features on the storage device, and then sending the packed features back to the computation device. This process introduces extra communication for transferring indices and synchronization between the computation and storage devices.

One-sided Communication. FreshGNN employs one-sided communication to address this problem. Based on Unified Virtual Addressing (UVA) [27], the memory of the CPU and GPUs are mapped to a unified address. This enables the computation device to directly fetch features from a mapped buffer of the storage device using the node index. As shown in Figure 9(a), nodes needed for training are partly pruned by the cache (green color), and for the unpruned ones (orange color), the GPU fetches the node features using their node ID directly from node features mapped with UVA. In Figure 9(b), multiple GPUs can concurrently fetch data using UVA for parallel training.

Multi-round Communication. With a larger number of GPUs available during training, all node features can be partitioned and stored across multiple machines such that GPUs serve as both computation and storage devices. Therefore, each GPU fetches the features of the relevant unpruned nodes from other GPUs, resulting in all-to-all communication between every pair of GPUs. UVA can still be used to perform one-sided memory access in this scenario. However, as GPUs are connected asymmetrically, link congestion could badly degrade the overall bandwidth. To address this problem, in addition to one-sided communication, FreshGNN breaks cross-GPU communication into multiple rounds to avoid congestion and fully utilize the bi-directional bandwidth on links. Figure 9(c) shows a typical interconnection among four GPUs, where GPUs are first connected via PCIe and then bridged via a host. For this topology, FreshGNN will decompose the all-to-all communication into five rounds. In round one, data is exchanged only between GPUs under the same PCIe switch, while the remaining four rounds are for exchanging data between GPUs across the host bridge. The data transmission at each round is bi-directional to fully utilize the bandwidth of the underlying hardware. This multi-round communication can effectively avoid congestion in all-to-all communication.



(c) Multi-round communication among GPUs connected with PCIe

Figure 9: Feature data loading under different scenarios. CPU-GPU training in (a) uses one-sided communication to directly read from CPU memory. Multiple GPUs initiate one-sided communication concurrently for CPU data in (b). Communication among GPUs is scheduled in multiple rounds in (c) to improve bandwidth.

7 Evaluation

In this section, we first describe our experimental setup, followed by results covering system efficiency and model accuracy. We conclude with an empirical study of our cache effectiveness as well as ablations over our system optimizations.

7.1 Experimental Setup

Table 2: Graph dataset details, including input node feature dimension (Dim.) and number of classes (#Class).

Dataset	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	Dim.[1]	#Class
`ogbn-arxiv` [17]	2.9M	30.4M	128	64
`ogbn-products` [17]	2.4M	123M	100	47
`ogbn-papers100M` [17]	111M	1.6B	128	172
`MAG240M` [16]	244.2M	1.7B	768	153
`Twitter` [41]	41.7M	1.5B	768	64
`Friendster` [42]	65.6M	1.8B	768	64

Environments. Single GPU experiments were conducted on a server with two AMD EPYC 7742 CPUs (2×64 cores in total) and an NVIDIA A100 (40GB) GPU. The software environment on this machine is configured with Python v3.9, PyTorch v1.10, CUDA v11.3, DGL v0.9.1, and PyG v2.2.0. The multi-GPU experiments were conducted on an AWS p3.16xlarge machine with two Intel Xeon E5-2686 CPUs (2×16 cores in total) and eight NVIDIA Tesla V100 (16G) GPUs.

Datasets. The dataset statistics are listed in Table 2. Among them, the two smallest datasets, `ogbn-arxiv` and `ogbn-products`, are included only for model accuracy comparisons and to provide contrast with larger datasets. `ogbn-papers100M` and `MAG240M` [16] are used to test both accuracy and speed. Note that `MAG240M` is the largest publicly-available graph dataset. Following the common practice of previous work [11, 43], we also use the graph structure from `Twitter` [41] and `FriendSter` [42] with artificial features to test the speed of different systems.

GNN models & Training details. We employed three widely-used GNN architectures for our experiments: GraphSAGE [14], GCN [20], and GAT [33]. All models have 3 layers and 256 as the hidden size. To measure their baseline model performance, we train them using mini-batch neighbor sampling in DGL. We follow the settings of their OGB leaderboard submissions to set the sampling fan-out as 20, 15 and 10; the batch size is chosen to be 1000. We set $p_{grad} = 0.9$ and $t_{stale} = 200$ for FreshGNN for all experiments (with the exception of Section 7.4, where we study the impact of these thresholds).

7.2 System Efficiency

We first demonstrate the system advantage of FreshGNN against state-of-the-art alternatives for GNN mini-batch training, including DGL [37], PyG [9], PyTorch-Direct [27], GNNLab [43] and representative mini-batch training algorithms such as ClusterGCN [5] and GAS [10].

Figure 10 compares the time for training a GraphSAGE model for one epoch on the four large-scale graph datasets using a single GPU. FreshGNN significantly outperforms all the other baselines across all the datasets. Compared with the widely-used GNN systems DGL and PyG, FreshGNN is 5.3× and 23.6× faster respectively on `ogbn-papers100M`. Both PyTorch-Direct and FreshGNN utilize CUDA UVA memory to accelerate feature loading, but FreshGNN is still 4.6× faster because it can reduce the number of features to load by a large margin. With respect to other mini-batch training algorithms, FreshGNN is orders of magnitude faster than GAS and ClusterGCN on `ogbn-papers100M`. GAS also runs out

[1]The first three datasets use float32 while the latter three use float16.

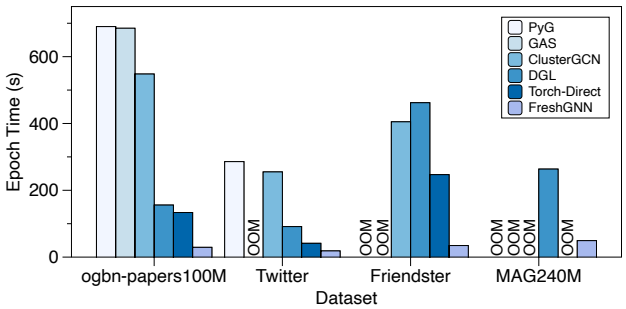


Figure 10: Epoch time comparisons training a GraphSAGE model using a single GPU.

of memory on graphs that have either more nodes/edges or larger feature dimensions due to the need to store the historical embeddings of all the nodes. Incidentally, on the largest dataset `MAG240M` only DGL and FreshGNN avoid OOM, but with FreshGNN executing 5.3× faster.

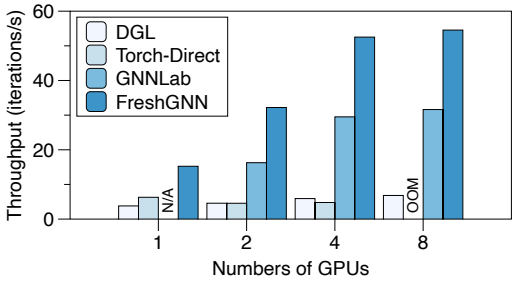


Figure 11: Scalability comparison for training a GraphSAGE model on `obgn-papers100M` using multiple GPUs.

FreshGNN can also easily scale to multiple GPUs. In this setting, we include a new baseline GNNLab [43] which partitions GPUs to sampling or training workers; hence GNNLab does not have single-GPU performance. Figure 11 compares the throughput (measured as the number of iterations computed per second) when training GraphSAGE on `ogbn-papers100M`. Both DGL and PyTorch-Direct deliver almost no speedup because of the data loading bottleneck that cannot be parallelized via the addition of more GPUs. Meanwhile, FreshGNN enjoys a nearly linear speedup from 1 to 4 GPUs and is up to 2.0× faster than GNNLab. However, the throughput growth mostly saturates when advancing from 4 to 8 GPUs for both FreshGNN and GNNLab. In this regime we find that graph sampling becomes a new bottleneck since it is conducted in CPU; we defer further consideration of this issue to future work.

FreshGNN is not only fast but also reliably converges to the desired target accuracy. Figure 12 plots the time-to-accuracy curve of different training systems. All the baselines here are using mini-batch neighbor sampling without any further approximation so they all converge to the same accuracy

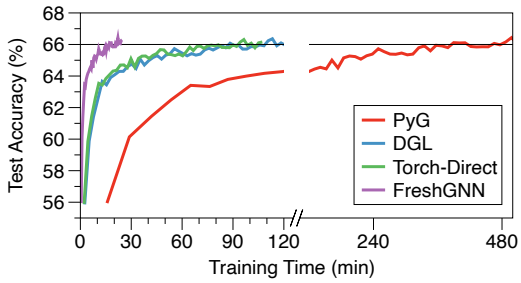


Figure 12: Test accuracy curve for GraphSAGE on `ogbn-papers100M`.

of ∼66%. FreshGNN can reach this same accuracy in 25 minutes while the slowest baseline (PyG) takes more than 6 hours.

7.3 Model Accuracy

Table 3 compares the test accuracy of FreshGNN with other mini-batch training algorithms: GAS [10], ClusterGCN [5], and another historical-embedding-based algorithm GraphFM [23]. Following the common practice in [6, 11], the target accuracy is obtained from training the models using neighbor sampling.

In general, nearly all algorithms perform relatively well on small graphs such as `ogbn-arxiv` and `ogbn-products` with only a few exceptions. However, when scaling to larger graphs such as `ogbn-papers100M`, all baselines experience a severe accuracy drop (from 7% to 18%) while running out of memory on `MAG240M`. By contrast, FreshGNN only experiences a less than 1% accuracy difference on both large datasets. Besides, FreshGNN can also demonstrate a similar accuracy pattern in the training process with normal mini-batch training, with the figure shown in Appendix.

7.4 Cache Effectiveness

Recall that p_{grad} and t_{stale} are the two thresholds that control the admission and eviction criteria of the historical embedding cache. In this section, we study their impact on system performance and model accuracy. We will show that a straightforward choice of the two thresholds can lead to the aforementioned competitive system speed as well as model accuracy.

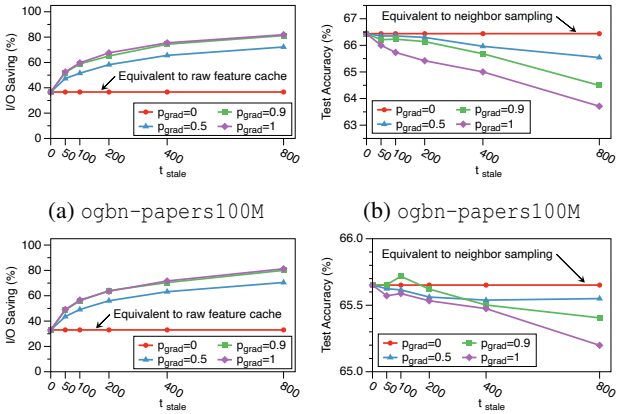
Impact on System Performance. In typical cache systems, the I/O saving percentage is equal to the cache hit rate. However, hitting the cached historical embeddings of a node will prune away all the I/O operations that would otherwise be needed to load the features of multi-hop neighbors, meaning the I/O savings can potentially be much larger than the cache hit rate. We plot this saving percentage w.r.t. neighbor sampling (i.e., no caching) under different p_{grad} and t_{stale} in Figure 13 (a) and (c). Because the historical embedding cache in FreshGNN is initialized via raw node features (Section 4.2), the red $p_{grad} = 0$ line reflects the performance of neighbor sampling with a raw feature cache. As expected, larger p_{grad} or t_{stale} results in more I/O reduction. On both graph datasets, a raw feature cache can only reduce I/O by $< 40\%$ but the historical embedding cache can reduce I/O by more than 60% when choosing $t_{stale} > 200$.

Table 3: Test accuracy of different training algorithm minus the target accuracy obtained by neighbor sampling (larger is better). Bold numbers denote the best performing method.

		Small datasets		Large datasets	
	Methods	`arxiv`	`products`	`papers100M`	`MAG240M` [2]
SAGE	Target Accuracy	70.91	78.66	66.43	66.14
	GAS	+0.44	-1.19	-8.17	OOM
	ClusterGCN	-3.10	+0.00	-7.57	OOM
	GraphFM	**+0.62**	-7.90	-18.40	OOM
	FreshGNN	+0.60	**+0.38**	**-0.15**	**-0.51**
GAT	Target Accuracy	70.93	79.41	66.13	65.16
	GAS	**-0.04**	-2.23	-8.67	OOM
	ClusterGCN	-3.91	-2.92	-8.08	OOM
	GraphFM	-22.67	-16.54	OOM	OOM
	FreshGNN	-0.50	**-0.54**	**-0.71**	**-0.36**
GCN	Target Accuracy	71.24	78.57	65.78	65.24
	GAS	+0.44	-1.91	-12.29	OOM
	ClusterGCN	-3.13	**+0.40**	-12.43	OOM
	GraphFM	**+0.47**	-15.70	-18.70	OOM
	FreshGNN	-0.71	-0.31	**-0.16**	**-0.29**

Impact on Model Accuracy. We plot the corresponding accuracy curves under different configurations in Figure 13 (b) and (d). As expected, larger p_{grad} or t_{stale} means more relaxed control on the embedding errors which consequently results in lower test accuracy. Beyond this, there are two interesting findings with practical significance. First, we can set p_{grad} very close to one without an appreciable impact on the model accuracy. This aligns well with the observation in Figure 3 that most of the node embeddings are temporally stable and can be safely admitted to the cache. And secondly, with a proper p_{grad} value, GNN models can tolerate node embeddings that were last updated hundreds of iterations ago. By cross-checking the I/O saving figures, we find that this is also a sweet spot in terms of system performance, which eventually led us to choose $p_{grad} = 0.9$ and $t_{stale} = 200$ for all the experiments (obviously excluding the Figure 13 results).

[2] We report the validation accuracy for MAG240M due to the absence of ground-truth labels for the test set.



(a) ogbn-papers100M (b) ogbn-papers100M

(c) MAG240M (d) MAG240M

Figure 13: (a) (c) The percentage saving of I/O for loading node features and (b) (d) the test accuracy on ogbn-papers100M and MAG240M under different choices of p_{grad} and t_{stale}.

7.5 Ablation Study of System Optimizations

In this section, we study the system performance of individual components. All the results are tested on ogbn-papers100M with a three-layered GraphSAGE model.

Subgraph Generator. Figure 14(a) shows the improvement of FreshGNN's graph sampler v.s. DGL's. On ogbn-papers100M, with 32 CPU threads, FreshGNN is able to reduce the sampling time per epoch from 72 seconds to 11 seconds (6.5×). In terms of scalability when using more threads, the speedup of FreshGNN's sampler is 26× while DGL is only 7.5×.

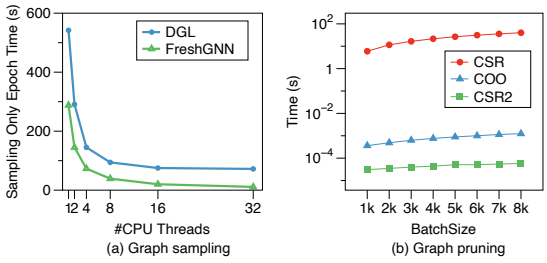


Figure 14: Effectiveness of FreshGNN's subgraph generator (measured on ogbn-papers100M).

Figure 14 (b) measures the time to prune the cached nodes and their neighbors from a subgraph using different graph data structures. Overall, CSR2 is orders-of-magnitude faster than CSR and COO regardless of the batch size in use. Specifically, CSR requires frequent CPU-GPU synchronization when invalidating the neighbors of the pruned nodes in the column index. As a result, graph pruning in total takes 99% of the iteration time. Subgraph pruning using COO is faster, which reduces the pruning overhead to 4.5% of the iteration time, but is still much slower than CSR2. The subgraph pruning time using CSR2 is negligible – it only occupies 26ms per iteration.

Data Loader. Figure 15 shows the improvement afforded by the optimizations of FreshGNN's data loader for multi-GPU communication. Compared with the unoptimized communication utilizing NCCL all-to-all operations, the one-sided communication is 23% faster on average on PCIe and NVLink GPUs. After scheduling using the multi-round communication pattern, the bandwidth is increased by 145% and 85%, respectively.

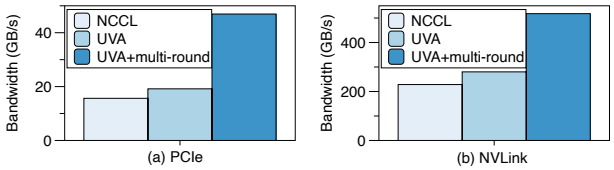


Figure 15: Optimizations for multi-GPU communication

7.6 Extension to heterogeneous graphs

While our primary focus was on the most popular large-scale benchmark settings, FreshGNN can also be naturally extended to heterogeneous graphs. We show one such use case here on MAG240M, which was homogenized in the previous experiments. Here we are using the heterogeneous form, and R-GNN [31] is the most popular heterogeneous GNN. So we compare the R-GraphSAGE model on FreshGNN with the neighbor sampling baseline on DGL. The accuracy is almost the same, but FreshGNN is 21.9× faster.

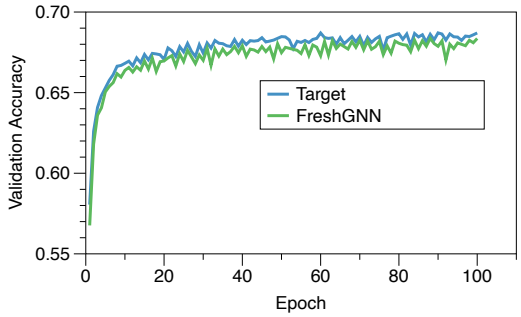


Figure 16: Validation accuracy curve for R-GraphSAGE on MAG240M.

8 Related Work

Full-Graph GNN Systems. There exists a rich line of work on scaling full-graph GNN training, where the general idea is to split the graph into multiple partitions that can fit onto the computing devices. Unlike mini-batch training, full-graph

training updates all the nodes and edges simultaneously. Notable systems include NeuGraph [24], ROC [18], and DistGNN [25], which lower the high memory footprint and data access costs by designing smart data partitioning or data swap strategies. In a different vein, DGCL [1] designs a new communication pattern to reduce network congestion. Other systems like Dorylus [32] and GNNAdvisor [38] optimize the computation of GNN training by exploring properties of the graph structure. And BNS-GCN [35] proposed boundary-node-sampling to reduce the communication incurred by boundary nodes during full graph training.

Finally, the SANCUS [30] framework reduces the amount of network communication by leveraging locally available node embeddings with bounded staleness to improve accuracy. However, although SANCUS resembles FreshGNN in some high-level conceptual aspects, the two systems are targeting very different training paradigms. Moreover, the cache policy of FreshGNN utilizes both gradient-based and staleness criteria to detect reliable embeddings. Additionally, we note that accuracy results for SANCUS on large-scale datasets have not yet been reported [30].

Data Movement Optimizations for GNNs. Previous work has shown that data movement is the bottle-neck for GNN mini-batch training on large graphs. And when using different ways of graph partitioning [19, 26, 49] and data placement [11, 27, 43], this data movement can be reduced under various training settings. For instance, DistDGL [49] replicates high-degree nodes, together with sparse embedding updates, to reduce the communication workload for distributed training. And MariusGNN [34] addresses the scenario of out-of-core training, reducing data swaps between disk and CPU memory by reordering training samples for better locality. FreshGNN's selective historical embedding method is orthogonal with DistDGL and MariusGNN, and can potentially be applied to them.

Training with Staleness. Staleness criteria have long been adopted for efficiency purposes when training deep neural networks. Systems including SSP [15], MXNet [21], and Poseidon [48] update model parameters asynchronously across different devices, helping to reduce global synchronization among devices and improve parallelism. Another branch of work, including PipeDream [29] and PipeSGD [22], pipeline DNN training and utilize stale parameters to reduce pipeline stall. They limit the staleness by periodically performing global synchronization. FreshGNN, however, selectively stores and reuses stale node embeddings instead of parameters which is unique to GNN models.

9 Conclusion

In this paper, we propose FreshGNN, a general framework for training GNNs on large, real-world graphs. At the core of our design is a new mini-batch training algorithm that leverages a historical cache for storing and reusing GNN node embeddings to avoid re-computation from raw features. To identify stable embeddings that can be cached, FreshGNN designates a cache policy using a combination of gradient-based and staleness criteria. Accompanied with other system optimizations, FreshGNN is able to accelerate the training speed of GNNs on large graphs by at least 4.6× over state-of-the-art systems, with less than 1% influence on model accuracy.

References

[1] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. DGCL: An efficient communication library for distributed gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 130–144, New York, NY, USA, 2021. Association for Computing Machinery.

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

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

[4] Zhengdao Chen, Xiang Li, and Joan Bruna. Supervised community detection with line graph neural networks. *arXiv preprint arXiv:1705.08415*, 2017.

[5] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 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*, pages 257–266, 2019.

[6] Jialin Dong, Da Zheng, Lin F Yang, and Geroge Karypis. Global neighbor sampling for mixed cpu-gpu training on giant graphs. *arXiv:2106.06150*, 2021.

[7] Yingtong Dou, Zhiwei Liu, Li Sun, Yutong Deng, Hao Peng, and Philip S Yu. Enhancing graph neural network-based fraud detectors against camouflaged fraudsters. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 315–324, 2020.

[8] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, page 251–262, New York, NY, USA, 1999. Association for Computing Machinery.

[9] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[10] Matthias Fey, Jan E Lenssen, Frank Weichert, and Jure Leskovec. GNNAutoScale: Scalable and expressive graph neural networks via historical embeddings. In *International Conference on Machine Learning*, pages 3294–3304. PMLR, 2021.

[11] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 551–568, 2021.

[12] Thomas Gaudelet, Ben Day, Arian R Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy BR Hayter, Richard Vickers, Charles Roberts, Jian Tang, et al. Utilizing graph machine learning within drug discovery and development. *Briefings in bioinformatics*, 22(6):bbab159, 2021.

[13] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pages 1263–1272. PMLR, 2017.

[14] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1025–1035, Red Hook, NY, USA, 2017. Curran Associates Inc.

[15] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. *Advances in Neural Information Processing Systems*, 26, 2013.

[16] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. OGB-LSC: A large-scale challenge for machine learning on graphs. *arXiv:2103.09430*, 2021.

[17] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in Neural Information Processing Systems*, 33:22118–22133, 2020.

[18] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, March 2-4, 2020*, pages 187–198, Austin, TX, USA, 2020. mlsys.org.

[19] George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

[20] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations*, ICLR '17, Palais des Congrès Neptune, Toulon, France, 2017.

[21] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. *Advances in Neural Information Processing Systems*, 27, 2014.

[22] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. Pipe-sgd: A decentralized pipelined sgd framework for distributed deep net training. *Advances in Neural Information Processing Systems*, 31, 2018.

[23] Zekun Li, Shu Wu, Zeyu Cui, and Xiaoyu Zhang. GraphFM: Graph factorization machines for feature interaction modeling. *arXiv:2105.11866*, 2021.

[24] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: Parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 443–458, Renton, WA, July 2019. USENIX Association.

[25] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K Ahmed, and Sasikanth Avancha. DistGNN: Scalable distributed training for large-scale graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[26] Seung Won Min, Kun Wu, Mert Hidayetoglu, Jinjun Xiong, Xiang Song, and Wen-mei Hwu. Graph neural network training and data tiering. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '22, page 3555–3565, New York, NY, USA, 2022. Association for Computing Machinery.

[27] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. Pytorch-direct: Enabling

gpu centric data access for very large graph neural network training with irregular accesses. *arXiv:2101.07956*, 2021.

[28] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 533–549, 2021.

[29] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. Sancus: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proceedings of the VLDB Endowment*, 15(9):1937–1950, 2022.

[31] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks, 2017.

[32] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 495–514, 2021.

[33] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, Vancouver, BC, Canada, 2018. OpenReview.net.

[34] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. MariusGNN: Resource-efficient out-of-core training of graph neural networks. In *Eighteenth European Conference on Computer Systems (EuroSys' 23)*, 2023.

[35] Cheng Wan, Youjie Li, Ang Li, Nam Sung Kim, and Yingyan Lin. BNS-GCN: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. *Proceedings of Machine Learning and Systems*, 4:673–693, 2022.

[36] Cheng Wan, Youjie Li, Cameron R Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. PipeGCN: Efficient full-graph training of graph convolutional networks with pipelined feature communication. *arXiv preprint arXiv:2203.10428*, 2022.

[37] 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. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[38] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 515–531, 2021.

[39] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In *International Conference on Machine Learning*, pages 6861–6871. PMLR, 2019.

[40] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. Graph neural networks in recommender systems: a survey. *ACM Computing Surveys (CSUR)*, 2020.

[41] Jaewon Yang and Jure Leskovec. Patterns of temporal variation in online media. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM '11, page 177–186, New York, NY, USA, 2011. Association for Computing Machinery.

[42] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[43] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. GNNLab: a factored system for sample-based GNN training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 417–434, 2022.

[44] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 974–983, 2018.

[45] Jiaxuan You, Zhitao Ying, and Jure Leskovec. Design space for graph neural networks. *Advances in Neural Information Processing Systems*, 33:17009–17021, 2020.

[46] Hanqing Zeng, Muhan Zhang, Yinglong Xia, Ajitesh Srivastava, Andrey Malevich, Rajgopal Kannan, Viktor Prasanna, Long Jin, and Ren Chen. Decoupling the depth and scope of graph neural networks. *Advances in Neural Information Processing Systems*, 34, 2021.

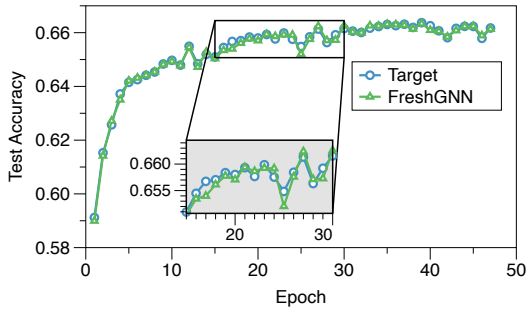
[47] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. GraphSAINT: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019.

[48] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 181–193, USA, 2017. USENIX Association.

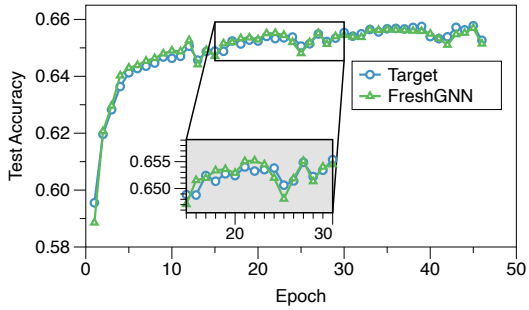
[49] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. DistDGL: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 36–44. IEEE, 2020.

[50] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. Layer-dependent importance sampling for training deep and large graph convolutional networks. *Advances in neural information processing systems*, 32, 2019.

A Comparison of FreshGNN Trainig Curves



(a) SAGE



(b) GCN

Figure 17: Test accuracy curve for FreshGNN and target neighbor sampling with same mini-batches and initial parameters on `ogbn-papers100M`.

In this experiment, we compare the test accuracy after each epoch in the training process of FreshGNN and the target neighbor sampling on `ogbn-papers100M`. The weight of the two counterparts are initialized as the same, and for each iteration, the sampled mini-batch is the same. For both GraphSAGE and GCN model, the curves are aligned quite well, showing that using the historical embedding cache has little change on the update of parameters.

B Convergence Analysis of the Historical Cache Applied to SGC

In this section, we analyze the convergence of the single-layer SGC model using a historical cache with a decoupled loss.

B.1 Model and Notations

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with n nodes, the adjacency matrix is $A \in \mathbb{R}^{n \times n}$ and the available feature matrix is $X \in \mathbb{R}^{n \times d}$, where d is the feature dimension. The normalized adjacency matrix for graph propagation is defined as $\hat{A} = (D+I)^{-\frac{1}{2}}(A+I)(D+I)^{-\frac{1}{2}} \in \mathbb{R}^{n \times n}$, $D_{u,u} = \sum_v A_{u,v}$.

The SGC model can be formulated as

$$Z = \hat{A}^k XW. \tag{3}$$

Here the Z and W denote the embedding matrix and the weight matrix, respectively. For simplicity, we refer to $\hat{A}^k X$ as $\hat{X}$.

For the historical cache, we use a random selector to determine the staleness of the historical embedding we use for a node, where 0 staleness means not using historical embeddings. The series of matrices S are diagonal binary matrices indicating whether or not to use a historical embedding and which one to use. S_0 selects the nodes that are calculated from neighbor aggregation, i.e. not using history, while $S_\tau, \tau = 1, \cdots, s$ select the historical embeddings with corresponding staleness. So the historical model is:

$$\tilde{Z}^{(0)} = \hat{X}W^{(0)} \tag{4}$$

$$\tilde{Z}^{(t)} = S_0^{(t)}\hat{X}W^{(t)} + \sum_{\tau=1}^{s} S_\tau^{(t)}\tilde{Z}^{(t-\tau)} \tag{5}$$

$$\sum_{\tau=0}^{s} S_\tau^{(t)} = I. \tag{6}$$

Here s is the maximum threshold for the staleness of historical embeddings. We assume S are random and can be different at different iteration t. They are constructed as follows: $\xi_1, \xi_2, \cdots, \xi_n$ are independent and identically distributed with $\mathbb{P}(\xi_i = \tau) = p_\tau, \tau = 0, 1, \cdots, s, i = 1, 2, \cdots, n$, and $\sum_{\tau=0}^{s} p_\tau = 1$. The i-th element in the diagonal of S_τ, $S_{\tau,ii} = \mathbb{I}_{\{\xi_i=\tau\}}$, where $\mathbb{I}$ is the indicator function. So $\mathbb{E}[S_{\tau,ii}] = \mathbb{E}\left[\mathbb{I}_{\{\xi_i=\tau\}}\right] = p_\tau$ and $\mathbb{E}[S_\tau] = p_\tau I$. It can be easily shown that $S_i S_j = S_i \cdot \mathbb{I}_{\{i=j\}}$

B.2 Loss Functions

Suppose we have a metric function $\mathcal{L}$ with respect to the embedding Z and label Y, we can define the loss function of W as $\ell(W) = \mathcal{L}(Z, Y) = \sum_{i=1}^{n} \mathcal{L}(Z_i, Y_i)$ and $\tilde{\ell}(W) = \mathcal{L}(\tilde{Z}, Y) = \sum_{i=1}^{n} \mathcal{L}(\tilde{Z}_i, Y_i)$.

Note that here the Z in the loss function ℓ is the embedding calculated without using history. In the historical model, we are using $\nabla\tilde{\ell}(W)$ for the update of parameter, and we want to show that it can still make W converge in $\ell(W)$ thought it is not the real gradient of $\ell(W)$.

Assumption B.1. *The loss function $\ell(W)$ is L-smooth w.r.t. W, i.e. $\|\nabla\ell(W^+) - \nabla\ell(W)\|_F \le L\|W^+ - W\|_F$.*

B.3 Proof of the Main Proposition

Proposition B.1. *Under Assumption B.1, for the historical model in eq. (5), if we use a fixed step size $\eta = \frac{1}{L}$, then the weight W will converge to a stationary point of $\ell(W)$.*

Proof. At timestamp $t > 1$,

$$Z^{(t)} = \hat{X}W^{(t)}$$

$$\tilde{Z}^{(t)} = S_0^{(t)}\hat{X}W^{(t)} + \sum_{\tau=1}^{s} S_\tau^{(t)}\tilde{Z}^{(t-\tau)}.$$

Since $S_i S_j = S_i \cdot \mathbb{I}_{\{i=j\}}$, we have

$$S_0^{(t)} Z^{(t)} = S_0^{(t)}\tilde{Z}^{(t)}.$$

This means

$$\left[S_0^{(t)}\nabla_{Z^{(t)}}\mathcal{L}(Z^{(t)}, Y)\right]_{i,:} = \nabla_{Z_i^{(t)}}\mathcal{L}(Z_i^{(t)}, Y_i) \cdot \mathbb{I}_{\left\{S_{0,ii}^{(t)}=1\right\}}$$
$$= \nabla_{\tilde{Z}_i^{(t)}}\mathcal{L}(\tilde{Z}_i^{(t)}, Y_i) \cdot \mathbb{I}_{\left\{S_{0,ii}^{(t)}=1\right\}} = \left[S_0^{(t)}\nabla_{\tilde{Z}^{(t)}}\mathcal{L}(\tilde{Z}^{(t)}, Y)\right]_{i,:}.$$

Therefore,

$$S_0^{(t)}\nabla_{Z^{(t)}}\mathcal{L}(Z^{(t)}, Y) = S_0^{(t)}\nabla_{\tilde{Z}^{(t)}}\mathcal{L}(\tilde{Z}^{(t)}, Y).$$

The gradient of $\ell(W^{(t)})$ is

$$\nabla\ell(W^{(t)}) = \hat{X}^\top\nabla_{Z^{(t)}}\mathcal{L}(Z^{(t)}, Y),$$

while the "gradient" we use for updating the parameter is

$$\nabla\tilde{\ell}(W^{(t)}) = \hat{X}^\top S_0^{(t)}\nabla_{\tilde{Z}^{(t)}}\mathcal{L}(\tilde{Z}^{(t)}, Y) = \hat{X}^\top S_0^{(t)}\nabla_{Z^{(t)}}\mathcal{L}(Z^{(t)}, Y).$$

So the update rule for W is

$$W^{(t+1)} \leftarrow W^{(t)} - \eta\hat{X}^\top S_0^{(t)}\nabla_{\tilde{Z}^{(t)}}\mathcal{L}(\tilde{Z}^{(t)}, Y)$$
$$= W^{(t)} - \eta\hat{X}^\top S_0^{(t)}\nabla_{Z^{(t)}}\mathcal{L}(Z^{(t)}, Y).$$

Since ℓ is Lipschitz smooth, it holds true that

$$\ell(W^{(t+1)}) \le \ell(W^{(t)}) - \eta\left\langle\nabla\ell(W^{(t)}), \hat{X}^\top S_0^{(t)}\nabla_{Z^{(t)}}\mathcal{L}(Z^{(t)}, Y)\right\rangle$$
$$+ \frac{L\eta^2}{2}\left\|\hat{X}^\top S_0^{(t)}\nabla_{Z^{(t)}}\mathcal{L}(Z^{(t)}, Y)\right\|_F^2.$$

Take expectation for $S_0^{(t)}$, we have

$$\mathbb{E}\left[\ell(W^{(t+1)}) \mid W^{(t)}\right]$$
$$\le \ell(W^{(t)}) - \eta\left\langle\nabla\ell(W^{(t)}), \hat{X}^\top\mathbb{E}\left[S_0^{(t)}\right]\nabla_{Z^{(t)}}\mathcal{L}(Z^{(t)}, Y)\right\rangle$$
$$+ \frac{L\eta^2}{2}\left\|\hat{X}^\top\mathbb{E}\left[S_0^{(t)}\right]\nabla_{Z^{(t)}}\mathcal{L}(Z^{(t)}, Y)\right\|_F^2$$
$$+ \frac{L\eta^2}{2} tr\left(\nabla_{Z^{(t)}}\mathcal{L}^\top\left(Var\left[S_0^{(t)}\right] \odot (\hat{X}\hat{X}^\top)\right)\nabla_{Z^{(t)}}\mathcal{L}\right)$$
$$\le \ell(W^{(t)}) - \eta p_0\left(1 - \frac{L\eta}{2}\right)\left\|\nabla\ell(W^{(t)})\right\|_F^2.$$

Then by the law of total expectation,

$$\mathbb{E}\left[\ell(W^{(t+1)})\right] = \mathbb{E}\left[\mathbb{E}\left[\ell(W^{(t+1)}) \mid W^{(t)}\right]\right]$$
$$\le \mathbb{E}\left[\ell(W^{(t)})\right] - \eta p_0\left(1 - \frac{L\eta}{2}\right)\left\|\mathbb{E}\left[\nabla\ell(W^{(t)})\right]\right\|_F^2.$$

So,

$$\left\|\mathbb{E}\left[\nabla\ell(W^{(t)})\right]\right\|_F^2 \leq \frac{\mathbb{E}\left[\ell(W^{(t)})\right] - \mathbb{E}\left[\ell(W^{(t+1)})\right]}{\eta p_0\left(1-\frac{L\eta}{2}\right)};$$

$$\min_{t=0,\cdots T}\left\|\mathbb{E}\left[\nabla\ell(W^{(t)})\right]\right\|_F^2 \leq \frac{\ell(W^{(0)}) - \ell(W^*)}{(T+1)\eta p_0\left(1-\frac{L\eta}{2}\right)}.$$

Finally, as the iteration number satisfies $T \to \infty$, the expected gradient of the loss goes to 0, and thus the convergence of the historical model to a stationary point is guaranteed. □