# FreshGNN: Training GNN Models via Faithfully Refreshed Embeddings from a Selective History

Haitian Jiang, Kezhao Huang, David Wipf, Minjie Wang

## Abstract

The performance bottleneck of training graph machine learning models on large graphs is data loading. The node features are stored in slow memory, making it inefficient to load them. Moreover, the locality of data loading is bad due to the irregularity of graph data. It is both important and challenging to optimize the loading of node features. In this paper, we propose FreshGNN, a graph machine learning training system with selective historical cache. It can selectively cache the node embeddings that are recently produced or well trained, and reuse the embeddings instead of re-computation. It can bound the error of staleness and randomness, and be transparently applied to different models. Evaluation on different GNN models shows that it can keep the test accuracy while improve the performance by 2.23× on average, on the large graphs such as OGBN-MAG240M and FriendSter.

**ACM Reference Format:**
Haitian Jiang, Kezhao Huang, David Wipf, Minjie Wang. 2022. FreshGNN: Training GNN Models via Faithfully Refreshed Embeddings from a Selective History. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Graph is ubiquitous to represent relations. Linked Web pages, papers citations, molecule interactions, purchase behaviors, etc., can all be modeled as graphs. Hence, many real-world applications are based on learning from graph data. Recently, Graph Neural Networks (GNNs) emerged to be a promising new technique and demonstrated strong performance on a wide range of graph machine learning applications, such as community detection [3], recommender system [31], fraud detection [6], drug discovery [10] and so on.

The success of GNNs is largely attributed to their capability to exploit structural information via the so-called *message passing paradigm*, where updating a node embedding requires collecting and aggregating embeddings of its neighbors. Repeatedly applying this procedure (i.e., stacking multiply GNN layers) allows GNNs to produce node embeddings that capture the local topology of the nodes.

However, recursively collecting messages from neighbors leads to an exponentially growing cost of data access and computation, which becomes a significant barrier to training GNNs on large graphs. Taking the OGBN-MAG240M graph from the Open Graph Benchmark (OGB) [13] as an example, it has 244M nodes and 1.7B edges, and each node has a feature vector of length 768. To train a 3-layer GNN, *each* node needs to read features from around 50K 3-hop neighbors on average, resulting 150MB of data access.

This scalability challenge have been tackled with different kinds of approximation. The first line of work samples a sparser adjacency matrix $\hat{A}$ from the original adjacency matrix $A$ to reduce the number of messages to collect. Notable strategies are neighbor sampling [11], layer-wise sampling [2, 42] and graph-wise sampling [4, 40]. Although $\hat{A}$ can be designed as an unbiased estimator of the original adjacency matrix $A$, the computed node embeddings may not be the unbiased estimator of the original. Furthermore, the exponential growth still exists, albeit at a lesser degree.

**Table 1.** Comparison between vanilla method and GAS on OGBN-Papers100M using SAGE.

| | Accuracy (%) | Epoch Time (s) | Staleness |
|---|---|---|---|
| **Vanilla** | 66.43 | 351.7 | N/A |
| **GAS** | 58.26 | 1098 | 1658 |
| **GraphFM** | 58.26 | 1098 | 1658 |

In fact, although various proposals based on historical node embeddings have shown promising results on medium sized graphs such as OGBN-Products (2.4M nodes, 123M edges), their performance degrade severely on web-scale graphs that are magnitudes larger. For example, on OGBN-Papers100M (111M nodes, 1.6B edges), model accuracy and training speed are significantly behind the vanilla neighbor sampling. There are two fundamental reasons. First, previous methods have no control on the *staleness* of the historical embeddings. Especially on gigantic graphs where each training epoch is usually composed of many mini-batches, the historical embeddings can be severely outdated, which jeopardizes the model performance. Second, saving the historical embeddings for all the nodes are extremely memory demanding. For example, training a 3-layer GCN model using GNNAutoScale [8] on OGBN-Papers100M requires 230GB memory to save the historical embedding.

This paper proposes a new framework that can efficiently scale GNN training to web-scale graphs of billions of edges with negligible impact on model performance. At its core is a novel strategy called FreshGNN that retrieves **F**aithfully **R**efreshed **E**mbeddings from **S**elective **H**istory. We formulate the strategy as a classical cache policy, where the node historical embeddings are being admitted or evicted based on various criterion during training. The goal is to keep high-quality historical embeddings that have little impact to

model prediction in the cache for reuse, which we achieved by measuring the staleness defined by the time interval to the latest admission of an embedding, and its gradient w.r.t. to the loss produced by the current mini-batch. FreshGNN is agnostic to the GNN model in use and can be easily plugged into existing GNN architecture, and works with and without sampling. Our provides an easy-to-use interface for users to train arbitrary models with FreshGNN and an optimized runtime that accelerates the operations related to sampling, feature fetching and interaction with the historical embedding cache.

To summarize, the paper has the following contributions,

- A new framework that scales GNN training to large graphs without compromising model performance, accomplished by historical embedding cache that controls the quality of embedding.
- The detailed design of caching policies for such a cache, which takes into account of staleness and gradients. Further more, we provide theoretical analysis on their convergence.
- A practical implementation of FreshGNN which consists of an easy-to-use interface and an optimized runtime.
- A comprehensive evaluation of FreshGNN across a variety of model baselines, graph datasets and hardware configurations. The results show that FreshGNN can achieve very close accuracy to vanilla neighbor sampling while being up to 30x faster than previous work based on historical embeddings.

## 2 Background

In this section, we introduce the basic ideas of graph neural networks and the scalability challenge in graph neural networks that we are about to solve.

### 2.1 Graphs and Graph Neural Networks

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an graph with $n$ nodes. The node set is $\mathcal{V} = \{v_1, v_2, \cdots, v_n\}$ and the edge set is $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. Let $\mathbf{A} \in \{0,1\}^{n \times n}$ be the graph adjacency matrix, where $A_{ij} = 1$ if and only if there exists an edge between $v_i$ and $v_j$. Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be the node feature matrix and $\mathbf{Y} \in \mathbb{R}^n$ be the lable. For each node $i$, it has a $d$-dimentional node feature vector $X_i$ and a label $Y_i$.

The goal of GNN is to learn a representation $\mathbf{h}_v$ for each node $v$, which can be used for node classification or link prediction tasks. It follows a *message passing* scheme. For the $(l+1)$-th layer of a GNN, the hidden representation is calculated as below:

$$\begin{aligned} h_v^{(l+1)} &= f_{\mathbf{W}}^{(l)}\left(h_v^{(l)}, \left\{h_u^{(l)} : u \in \mathcal{N}(v)\right\}\right) \\ &= \phi_{\mathbf{W}}^{(l+1)}\left(h_v^{(l)}, \underset{u \in \mathcal{N}(v)}{\mathrm{AGG}}\left(\psi_{\mathbf{W}}^{(l+1)}\left(h_u^{(l)}, h_v^{(l)}\right)\right)\right) \end{aligned} \tag{1}$$

where $h_v^{(l)}$ is the embedding in layer $l$ for node $v$, $\mathcal{N}(v)$ is the set of neighbor nodes of $v$. The $h_v^{(0)}$ is set to the node feature $X_v$. $\phi$ and $\psi$ are the update function and message function respectively, parameterized by the weight $\mathbf{W}$. The aggregation function AGG is a permutation-invariant function like sum, mean, or element-wise maximum.

In the matrix form, each layer can be represented as

$$\mathbf{H}^{(l)} = \sigma\left(\mathbf{H}^{(l-1)}, \hat{\mathbf{A}}; \mathbf{W}^{(l-1)}\right) \tag{2}$$

Here $\sigma$ denotes the non-linear activation function such as ReLU or GELU, which is inside the update function $\phi$ in the node-wise form. And $\mathbf{W}^{(l)}$ is the weight matrix for each layer. The propagation matrix $\hat{\mathbf{A}}$ is used for message passing and aggregation of node features through the layers.

### 2.2 Graph Neural Networks in Practice

Graphs used in GNN training can have a large number of nodes attached with high-dimensional (100-1000) features [13, 36]. In Open Graph Benchmark (OGB) [13], the largest graph dataset lsc-MAG240M has 240M nodes with feature length at 768, the total size of node features reaches up to 700GB. On the other hand, as nodes in a graph are dependent to each other, full graph training requires the features and intermetiate embeddings of all nodes to be simultaneously present in the computing device, which goes beyond the memory capacity of a single device.

Efforts have been made the alleviate the problem of memory capacity by partitioning the data and computation into multiple devices. However, due to the dependence from graph structure, frequent communication has to be performed across devices at every layer of the model, resulting in large overheads. Furthermore, the communication overhead becomes more severe with growing graph size and the consequently increasing device number. In the example of training lsc-MAG240M, each device should fetch data over 500GB at each iteration. Compared with typical GPU memory (~32GB) and cross-GPU bandwidth (~15GB/s), the communication is infeasible. Approximation such as using stale embeddings [23, 28] and ignore some boundary nodes [27] can reduce communication workload, but they lead to large accuracy decrease due to the loss of information.

The most widely accepted solution to such issues is to train and update the parameters in a stochastic way, featured by the neighbor sampling method [11]. In each iteration, instead of training all the nodes, it selects a subset of nodes from the training set and generates the subgraph structure of them by expanding their neighbor nodes in multiple hops. And neighbor sampling further reduces the working memory by sampling from the neighbor nodes in place of the entire neighborhood. Compared with full graph training, the size of the subgraph with limited neighborhood is much smaller and can be trained by a single device. The good accuracy [37] and

generality [11] further make neighbor sampling a standard way of GNN training [9, 14, 35? ].

### 2.3 Optimizations for Mini-batched Training

Even with neighbor sampling, training is still bottlenecked by the computation and data movement brought by the exponential growth of multi-hop graph neighbors among different layers [15]. The large dimension of node features and the limited GPU memory exacerbate the issue by even larger and more frequent data movement.

Two types of work had been done to alleviate this issue in mini-batch training. Shown in Figure 2(a), the first kind is to cache the raw features of frequently visited nodes by metrics like node degree, weighted reverse pagerank, and profiling [19, 20, 35]. While by the power law, most of the nodes in the graphs have moderate visiting frequency. Feature cache cannot save the memory access to them. Moreover, their effectiveness greatly depends on the graph structure, leading to an unstable improvement as the graph datasets vary.

The second line of work [1, 8, 38] is to use embeddings produced at previous iterations to replace the neighbor expansions so as to reduce data movement, named historical embeddings. In Figure 2(b), they employ historical table to store the produced embedding for every node and reuse them in the following iterations to avoid neighbor expansion. However, the size of the historical table is $O(LVH)$, which is even larger than node features that are $O(VH)$ and leads to frequent data movement. Moreover, these methods are coupled with unacceptable staleness. Example on a graph with 100M nodes shows that they are using historical embeddings that were generated 1600 iterations before on average, and produces model with accuracy of 10% less than the one from neighbor sampling.

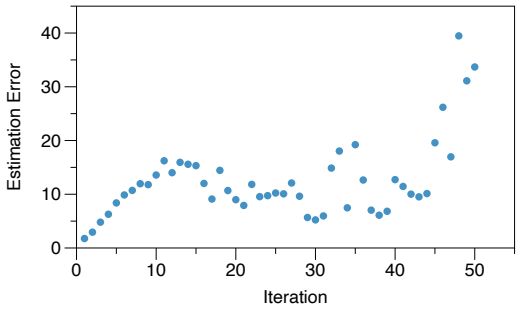



**Figure 1.** The mean $\ell$-2 estimation error of embeddings in GAS within one epoch with the growing staleness.

## 3 System Overview

In this work, we propose FreshGNN, a GNN system that selectively caches node embeddings to replace computation from raw data. The optimization is based on the observation

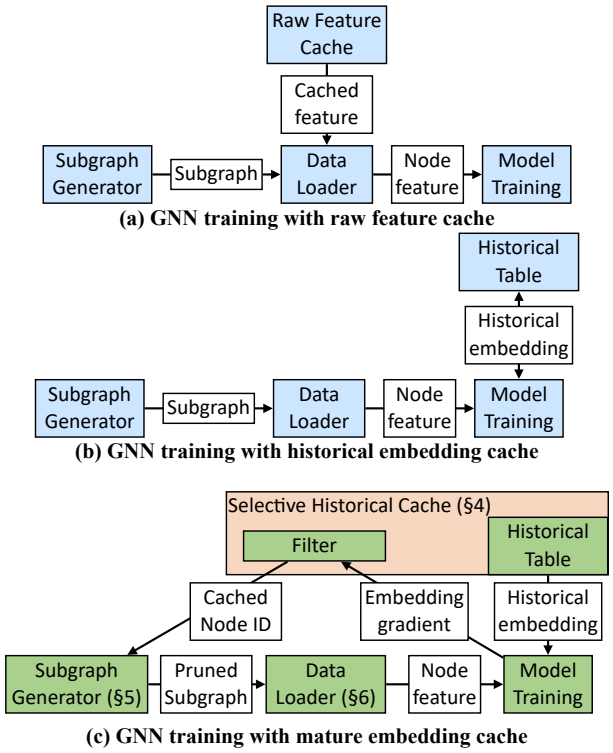



**Figure 2.** Comparing between existing cache optimizations and FreshGNN

that some nodes are already well-learnt and contribute little to model update, which is skippable in training. We name them as mature embeddings. As a result, we can filter the produced embedding to find out the mature embeddings and reuse them to replace neighbor expansion during training.

Figure 2(c) shows the overview of FreshGNN, to support such optimization. FreshGNN is still based on historical embedding, but has three main components that differs with existing systems. First, instead of caching raw features or nonselective historical embeddings, FreshGNN has selective historical cache which filters and reuses node embeddings produced by model training. To be selective, the filter receives gradient from training to update the cache. Second, to save workload using historical cache, a subgraph generator produces pruned subgraph with the cached nodes and neighbors removed from the graph structure according to historical cache. Finally, the data loader efficiently moves node features that are still needed from storage device to the computation device after pruning, and feeds them to GNN training.

**Selective historical cache.** The historical cache stores the node embeddings produced at previous iterations, namely historical embedding. For a node with historical embedding cached, instead of aggregating from its neighbor nodes, we can use historical embedding as the approximate aggregation

result to save computation and memory access workload. Figure 3 is a high-level illustration of the optimization. However, historical cache introduces error into the training because the historical embedding is an estimation of aggregation results. Large error may influence the model quality and lead to poor accuracy. To solve this, we propose error-aware cache mechanism to guide cache update. Error-aware cache mechanism utilizes gradient from training iterations to judge the error of generated embeddings. It stores the generated embeddings with small gradient and evicts the cached embeddings with large gradient. Moreover, it evicts the stale embeddings to further reduce the error.

**Subgraph generator.** As the cached nodes should be pruned from subgraph structure, the subgraph generator is dependent on the status of selective historical cache. As a result, the efficiency of subgraph generation is critical to overall performance. Naive method either uses CPU [7, 29**?** ] or GPU [14, 35] to generate the pruned subgraph. However, due to the weak computing capability of CPU and limited memory resources on the GPU, subgraph generation is either inefficient merely on CPU or just impractical on GPU solely. FreshGNN employs a decoupled two-stage method to take the advantage of both CPU and GPU. It performs graph sampling asynchronizedly in parallel to get the unpruned subgraphs by CPU threads, and prunes the subgraph efficiently by GPU in real time before the forward propagation. Pipelined execution and a new graph data structure are proposed to optimize the decoupled stages respectively.

**Data loader.** The data loader fetches the features of unpruned nodes. To improve data loading performance under different training settings, FreshGNN utilizes one-sided and two-sided communication for data loading. The one-sided communication uses UVM for GPU-oriented memory access to reduce the round-trips between device, which is suitable for CPU-GPU and multi-GPU training. And the two-sided communication can utilize multiple CPU threads while configuring memory access pattern to improve bandwidth, which works well under disk-CPU-GPU scenario.

While FreshGNN optimizes the data preparation and training efficiency, FreshGNN's method is independent to specific GNN model, and can be used to improve different GNN models out of box.

# 4 Selective Historical Cache

Mature embedding cache selectively filters and reuses mature embeddings of the nodes in the subgraph, and prunes their neighbors to save computation and memory access workload. The selectivity helps to improve performance while maintaining the quality of the GNN model. This section introduces the workflow of selective historical cache that optimizes memory access as well as the cache policy for selecting nodes with mature embeddings.

## 4.1 Cache Workflow

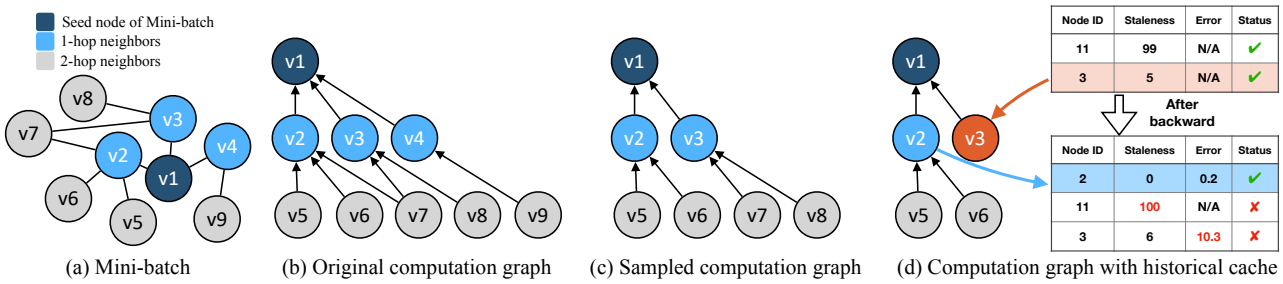
The workflow of selective historical cache has three parts that are look-up, check-in, and check-out. By looking up, the historical cache finds out the nodes in the subgraph with cached embedding, and reuses the embedding in place of neighbor expansion. After training the iteration, historical cache checks in the node embeddings produced by the model. Furthermore, the historical embeddings used in current iteration can also be checked out if they lead to large loss.

Figure 3 shows an example workflow. For a given mini-batch in Figure 3(a), GNN training needs multiple hops of its neighbors as shown in the graph structure in Figure 3(b). Neighbor sampling help to reduce the size of graph structure by limiting the number of neighbors of center nodes and generates subgraph in Figure 3(c). But the performance is still dominant by exponentially growing number of nodes. Figure 3(d) shows mature embedding cache. It contains the embeddings as well as some metadata like the node ID, staleness, error, and validity status of the embeddings. In the subgraph, v3 has cached embedding while v2 does not, so the embedding of v3 is directly read from the cache while that of v2 is obtained from the computation of its neighbor expansion. After this training iteration, the staleness and error of cache are updated according to training results. Embeddings of v11 and v3 get evicted due to large staleness and large error respectively, while the newly produced embedding of v2 gets cached.

Algorithm 1 shows the training process using historical cache. It is based on mini-batch training. For each batch, it generates a subgraph according to user-defined sampling method (Line 5). The sampling method returns the subgraph consists of sampled $L$-hop neighbors for the training nodes, where the $L$ is defined by the GNN model. For each layer, by looking up in the historical table, the nodes in the subgraph are divided into two types: nodes whose embeddings can be reused from the cache ($\mathcal{V}_{cache}$ in Line 7) and normal nodes ($\mathcal{V}_{normal}$ in Line 10). For the cached nodes, their embeddings are directly read from the historical cache (Line 13), while for normal nodes, their embedding are computed by the neighbor aggregation (Line 14). After layers of forward and backward propagation, the gradient of both historical and normal embeddings are used to update historical cache. It first selectively checks out the used historical embeddings (Line 20) and then checks in the newly computed embeddings (Line 21) according to the cache policy that we will discuss below using the gradient results.

## 4.2 Cache Policy

The cache policy for selective historical cache is very different from raw feature cache. Raw feature cache stores the features of nodes that are meant to be read as input, which produces the same computation results. Therefore raw feature cache is always full for best utilization and efficiency.




(a) Mini-batch (b) Original computation graph (c) Sampled computation graph (d) Computation graph with historical cache

**Figure 3.** Overview of mature embedding cache

**Algorithm 1** Training with layered historical embedding.

**Input**: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, input node features $\mathbf{H^0}$, number of batches $B$, number of layers $L$, history table $\mathcal{T}$, rate for check-out using history $r$, the maximum staleness threshold for the table $s$
$t_{curr} \leftarrow 0$
$\{\mathcal{B}_1, ..., \mathcal{B}_B\} \leftarrow Split(\mathcal{G}, B)$
**for** $\mathcal{B}_b \in \{\mathcal{B}_1, ..., \mathcal{B}_B\}$ **do**
  $\mathcal{G}_b \leftarrow sample(\mathcal{G}, \cup_{v \in \mathcal{B}_b})$
  **for** $l \in \{L-1, ..., 1\}$ **do** ▷ No history for last layer
    $\mathcal{V}^l_{cache} \leftarrow \mathcal{V}^l_{\mathcal{G}_b} \cap \mathcal{T}^l$ ▷ History lookup
    $\mathcal{G}_b.remove\_subgraph(root = \mathcal{V}^l_{cache})$
      ▷ Remove nodes for the calculation of $\mathcal{V}^l_{cache}$
    $\mathcal{V}^l_{normal} \leftarrow \mathcal{V}^l_{\mathcal{G}_b} \setminus \mathcal{V}^l_{cache}$
  **end for**
  **for** $l \in \{1, ..., L\}$ **do**
    $h^l_u \leftarrow \mathcal{T}^l[u], \forall u \in \mathcal{V}^l_{history}$
    $h^l_u \leftarrow f^l_\theta(\{h^{l-1}_v : v \in \mathcal{N}_{\mathcal{G}_b}(u)\}), \forall u \in \mathcal{V}^l_{normal}$
  **end for**
  $loss = \ell(h^L_{\mathcal{B}_b}, label_{\mathcal{B}_b})$
  $loss.backward()$
  $t_{curr} \leftarrow t_{curr} + 1$
  **for** $l \in \{1, ..., L\}$ **do**
    CheckOut($\mathcal{T}^l, \mathcal{V}^l_{\mathcal{G}_b}, t, r, s$)
    CheckIn($\mathcal{T}^l, \mathcal{V}^l_{normal}, h^l, t$)
  **end for**
**end for**

However, for selective historical cache, the stored element is an estimation of the intermediate computation results. Erroneous estimation can badly hurt the model quality. So the cache policy for selective historical cache should record the mature embeddings while evicting the immature embeddings timely.

The challenge is how to judge the maturity of different embeddings. In FreshGNN, we use gradient and staleness to evaluate whether an embedding should be cached or evicted. Figure 4 shows how FreshGNN gets gradient as the indicator. In the current iteration, node 2 does not have cached embedding, so we compute its embedding $h^1_2$ using its neighbor

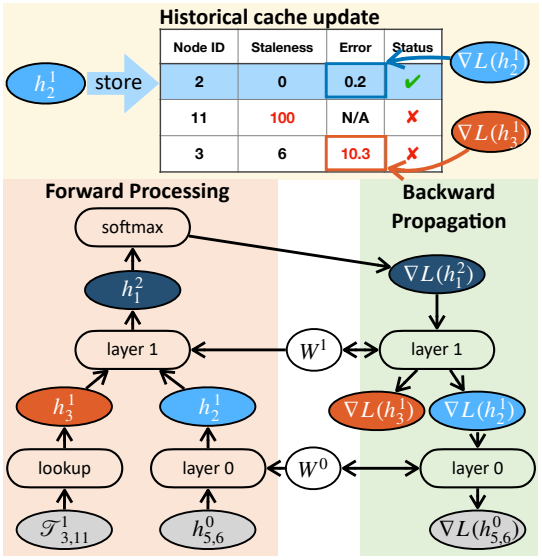



**Figure 4.** Updating mature embedding cache using gradient of embeddings

node 5 and 6's raw feature $h^0_{5,6}$, while the node 3's embedding $h^1_3$ is from the cache. After forward and backward propagation, we can get gradient of node 2 and 3 as $\nabla\mathcal{L}(h^1_2)$ and $\nabla\mathcal{L}(h^1_3)$. The gradient of the nodes is the indicator to evaluate their embeddings' maturity, which further guides the record and eviction of them.

FreshGNN calculates the norms of embeddings' gradient to the loss to quantify their maturity. Note that as the parameters are updated at each step, the comparison of gradients between iterations is meaningless. Therefore, the gradient is used to sort the maturity of embeddings that are produced or used within the same iteration. Newly-produced embeddings with small gradient contributes little to weight parameter update, which is then ignored by FreshGNN by caching and reusing these embeddings. For cached embedding used at current iteration, large gradient means that it leads to wrong predictions and contribute largely to weight parameters, therefore FreshGNN chooses to train it using neighbor nodes' raw features in upcoming iterations by evicting it from cache.

However, even within one iteration, it is unfair to directly compare the gradient of embeddings due to their different contribution to the loss. We find that the gradient contribution is influenced by the layer of the model and graph structure. For example, embeddings of the input layer are less influential to the prediction compared with embeddings of the center nodes (last layer). Also, embeddings of nodes with large out-degree contribute more to the loss because they influence more nodes. As we use gradient to compare the maturity of the embeddings, it is important to make fair comparison among them.

FreshGNN normalizes the gradient using both graph structure and model architecture. First, embeddings in different layers are managed separately due to their different distribution. Second, for embeddings in the same layer, FreshGNN takes both graph structure information (node degree) and model information (type of aggregator) into consideration. For example, when summation aggregation is used, the gradient of node $v$'s embedding at layer $l$ with outdegree of $n$ is $h_v^l.grad = \sum_{1\to n} h_u^{(l+1)}.grad, v \in \mathcal{N}(u)$. But for mean aggregator, $h_v^l.grad = \sum_{1\to n} h_u^{(l+1)}.grad/u.in\_deg, v \in \mathcal{N}(u)$.

By adaptively using the graph and model information, FreshGNN can make fair comparison of the embedding gradient, and prioritize cache replacement accordingly. FreshGNN also evicts the embeddings that are too stale by maintaining the check-in iteration ID for each cached embedding and comparing that with current iteration ID.

### 4.3 Remarks on FreshGNN Convergence

Not surprisingly, a 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 [30], 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* $\eta \le \frac{1}{L}$ *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 [1, 24, 28], it 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 rigorous convergence guarantees.

## 5 Pruned Subgraph Generation

With historical cache, FreshGNN is able to reduce memory access and computation workload. However, compared with original GNN training, selective historical training introduces the requirement for real-time subgraph generation. As historical cache is updated every iteration, the subgraph for training is determined in real time, on which the cached nodes together with their neighbors are pruned. Requirement for real time makes efficiency of subgraph generation critical to overall performance, and this section introduces how FreshGNN optimizes pruned subgraph generation.

### 5.1 Decoupled Sampling and Pruning

Due to FreshGNN's selective mechanism, historical cache is updated at the end of every iteration. As a result, the pruned subgraph generation, which depends on historical cache, should also be performed in real time at the beginning of each training iteration. As shown on the left of Figure 5, pruned subgraph generation is on the critical path of training, whose efficiency is important for overall performance.

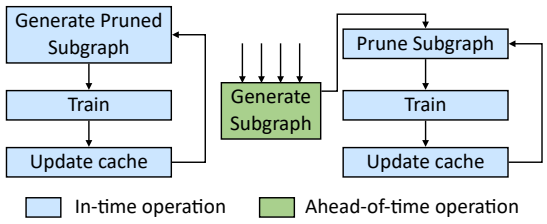



**Figure 5.** Decoupled method for pruned subgraph generation

There are two ways for real-time pruned subgraph generation. The first is to perform pruned subgraph generation on CPU at the beginning of every iteration. But as pointed out by previous work [11, 14, 35], the sequential execution on CPU makes it inefficient. The second method utilizes GPU, which is adopted by many recent works [14, 35]. With the large number of GPU threads expanding center nodes in parallel, the efficiency subgraph generation can be largely improved. However, utilizing GPU to produce subgraph requires to put the full graph structure into GPU memory, making it impractical to train large graph datasets. For example, graph structure for MAG240M-LSC takes storage of more than 20GB, leaving little space for data cache and model on GPU. The two methods are both impractical due to the limit of computation and memory resource on CPU and GPU respectively.

To solve the above problem, FreshGNN observed that pruned subgraph generation can be decoupled into two stages. The first stage generates the unpruned subgraph using neighbor sampling, with the second stage pruning the subgraph according to the historical cache. As shown on the right of Figure 5, after decoupling, FreshGNN is able to apply high-throughput ahead-of-time subgraph sampling for the first stage, and low-latency real-time subgraph pruning to generate pruned subgraph with awareness of historical cache at the second stage.

### 5.2 Ahead-of-time Subgraph Sampling

The first stage generates unpruned subgraph through sampling. It traverses both cached and uncached nodes and expands their neighbor nodes. Unpruned subgraph generation does not need information from historical cache; therefore, it can be prepared ahead-of-time before each training iteration. Though CPU thread is slow to produce one subgraph due to sequential execution, we can use multiple CPU threads to produce subgraphs in batch and achieve high throughput.

More importantly, during subgraph sampling, FreshGNN transfer the data of subgraph structure to GPU to make better use of CPU-GPU bandwidth. Data transferring is launched on different CUDA streams with the training CUDA stream, it subsequently has little influence on training efficiency.

While using more CPU threads can improve the parallelism and throughput of subgraph sampling, the subsequent data transfer will also increase the number of subgraphs residing in GPU memory. With GPU memory tightly constrained, the subsequence of sampling and transfer limits the parallelism of unpruned subgraph generation.

To break the limitation, FreshGNN designs the workflow of sampling and transfer in pipeline manner. CPU threads are instantiated as sampling workers and transfer workers. While FreshGNN can increase the number of sampling workers for high parallelism, it only maintains a moderate number of transfer workers to limit the number of subgraphs that concurrently resides in GPU memory.

Ahead-of-time subgraph sampling makes use of CPU resources that are previously ignored [14, 35], and only stores full graph structure in CPU memory, leaving GPU memory available for cache.

### 5.3 Real-time Subgraph Pruning

The second stage is subgraph pruning on GPU. For each subgraph, GPU removes the incoming edges of the nodes that exist in the historical cache. As pruning is guided by historical cache, it is in real time with training iterations and should have low latency. However, existing graph data structures are unsuitable for parallel modification, resulting in large overhead.

Compressed Sparse Row (CSR) is commonly used to represent graph structures. It has row index to represent the number of non-zeros at each row, with column index representing the columns that the non-zeros resides in. To remove the neighbors of a center node, we need to change the number of neighbors in row index, and remove the corresponding elements in column index. As the items in row index are dependent, it leads to large overhead to prune using GPU threads. Shown in Figure 6, to remove the neighbors of node 2 (index starting from 0), every entry after entry 2 in row index will be modified, of which is compute complexity is $O(N)$.

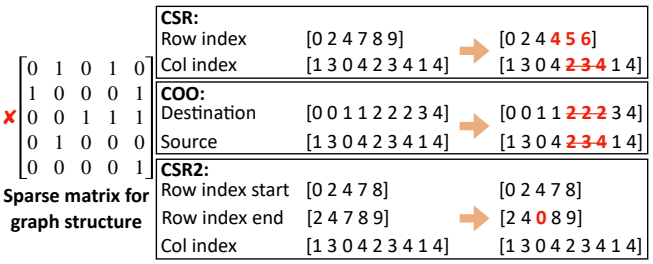



**Figure 6.** Removing a center node's neighbor using different sparse matrix formats

It is simpler to prune the subgraph structure using coordinate list (COO). It just needs to remove the edges of which the destination node is in the historical cache. In Figure 6, removing the entries that satisfies $Destination == 2$ can prune the graph. However, COO leads to substantial performance issues in training due to its edge-centric execution pattern [32] and increases the memory consumption to $O(2E)$.

To achieve efficiency in both subgraph pruning and GNN training, FreshGNN proposes a new way to represent graph data structure, named CSR2. CSR2 inherits CSR's node-centric representation, but uses two arrays for row index. The first array records the start of row index with the second array recording the end of it. And CSR2 has column index just the same as CSR's. As illustrated in Figure 6, before pruning, CSR2 has redundancy as $start[i]$ equals $end[i-1]$. However, with CSR2, to prune a center node, we just needs to modify the array of row index end using $end[i] = start[i], \forall i \in prune$, without changing the column index array. The modification will set the pruned center nodes' neighbor number to zero without influencing other center nodes. The compute complexity is $N_{prune}$, which is the number of center nodes to be pruned, and the operations are independent and can be parallelized using GPU threads.

Table 2 shows the comparison among CSR, COO, and CSR2 for subgraph pruning and GNN training. We can conclude that CSR2 has all the advantages of CSR and COO, while employing much smaller prune complexity ($O(N_{pruned})$) with little more storage overhead ($O(2N+E)$ vs. $O(N+E)$).

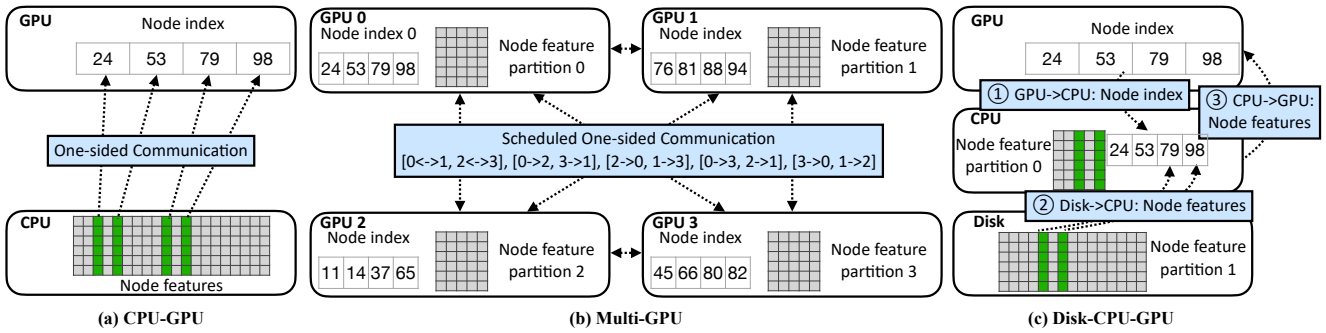



**Figure 7.** Node feature preparation under different scenarios; CPU-GPU training in (a) uses one-sided communication to directly read from CPU memory; Multi-GPU training in (b) schedules the one-sided communication order to fully utilize communication bandwidth; Disk-CPU-GPU training in (c) adopts two-sided communication to improve Disk-CPU bandwidth

**Table 2.** Prune complexity, storage complexity, and training performance of different sparse formats; $N$ is the number of nodes, $E$ is the number of edges, and $N_{pruned}$ is the number of pruned nodes of the subgraph

| | Prune Complexity | Storage Complexity | Training Performance |
|---|---|---|---|
| CSR | $O(N+E)$ | $O(N+E)$ | High |
| COO | $O(E)$ | $O(2E)$ | Low |
| CSR2 | $O(N_{pruned})$ | $O(2N+E)$ | High |

## 6 Data Loader

With subgraph structure generated, FreshGNN still needs the feature of the unpruned nodes to train the GNN model. In large graph training, node features have very high memory consumption because of the large number of nodes and long feature vector attached to each of them. As devices for computation (computation devices) usually have limited memory, the devices used for storage (storage devices) are different. To improve overall performance, FreshGNN also optimizes node feature preparation from storage device to computation device.

The challenge for node feature preparation comes from the sparsity and residence of node index. As node index is determined by the pruned subgraph structure, it is stored in computation device containing discontiguous entries. Therefore, node index is needed for indirect memory access when fetching node features from storage device.

FreshGNN has two types of communication method, which are **two-sided** communication that involves both computation and storage device, and **one-sided** communication with only computation device involved. Two-sided communication initiates data preparation on the storage device. It first moves node index from computation device to the storage device, pack the node features according to node index, and send the packed buffer back to computation device. It has at least two round-trips of communication, resulting extra overhead. One-sided communication can save the number of round-trips of node feature preparation. The core idea is to let computation device directly fetch data from storage devices using node index. With the two types of communication, FreshGNN is able to prepare node feature under different scenarios of GNN training.

**CPU-GPU Training.** In CPU-GPU training, CPU is the storage device and GPU is the computation device. Previous work has studies how to launch memory access to CPU memory from GPU [19]. Based Unified Virtual Memory (UVM), threads in GPU kernels can access the data residing in CPU memory. Internally, page fault mechanism migrates the buffer from CPU to GPU through PCIe implicitly. As shown in Figure 7(a), UVM enables one-sided memory access for GPU threads to directly fetch needed features according node index.

**Multi-GPU Training.** One-sided memory access is more complicated under multi-GPU scenario. In multi-GPU training, GPUs are both the computation device and storage device with node features partitioned. As shown in Figure 7(b), node features are evenly distributed to GPUs. GPUs separately maintain historical cache and prune subgraphs. After that, they fetch the needed node features from other GPUs according to pruned subgraph structure. UVM can still be used to perform one-sided memory access among GPUs. However, as GPUs are connected asymmetrically, link congestion badly hurts the overall bandwidth. FreshGNN schedules one-sided communication with awareness of topology for multi-GPU. The basic idea is to reorder the communications among GPUs to maximize PCIe utilization, with awareness of PCIe connections. Example in Figure 7(b) shows the communication schedule for four GPUs connected with tree-style PCIe. The schedule is to first fully utilize the bandwidth between GPUs under the same PCIe switch (0<->1, 2<->3), then it avoids link congestion by initiating bidirectional communication through PCIe host bridge, e.g., 0->2 and 3->1 at the same time.

**Disk-CPU-GPU data loading.** FreshGNN also supports disk-CPU-GPU training, which is useful when CPU memory

**Table 3.** Graph datasets and their number of vertices (**#V**), number of edges (**#E**), input vertex feature dimension (**Dim.**), and number of classification results (**#Class**) .

| Dataset | #V | #E | Dim. | #Class |
|---|---|---|---|---|
| **arxiv** [13] | 2.9M | 30.4M | 128 | 64 |
| **products** [13] | 2.4M | 123M | 100 | 47 |
| **papers100M** [13] | 111M | 1.6B | 128 | 172 |
| **MAG240M** [13] | 121.8M | 1.3B | 768[1] | 153 |
| **RMAG240M** [13] | 244.2M | 1.7B | 768[2] | 153 |
| **Twitter** [33][3] | 41.7M | 1.5B | 768 | 64 |
| **Friendster** [34][4] | 65.6M | 1.8B | 768 | 64 |

is also tightly constrained. In such setting, only part of the node features can be stored in CPU memory, with the rest of them stored in disk. As GPU cannot directly visit disk, FreshGNN adopts two-sided communication, passing node index to CPU and initiates memory access to disk and memory using CPU threads. As shown in Figure 7, in two-sided communication, CPU first copies node index from GPU, then uses it to fetch node features on disk, and sends them back to GPU. The data movement between CPU and GPU introduced by two-sided communication can be negligible, as GPU-CPU bandwidth is much higher than disk-CPU bandwidth (12GB/s from PCIe 3.0 vs. 3Gb/s from SSD). Moreover, memory access initiated by CPU threads is configurable. By reducing the read-ahead memory access to disk, disk-CPU bandwidth can be largely improved.

# 7 Evaluation

## 7.1 Methodology

**Table 4.** Overall results of test accuracy (%) and train epoch time (s) for large datasets (papers100M and mag240M).

| | Methods | arxiv | products | papers100M | mag240M[5] |
|---|---|---|---|---|---|
| SAGE | Vanilla | 70.91 | 78.66 | 66.43/351.7* | 66.14/258.0* |
| | GAS | 71.35 | 77.47 | 58.26/1098 | OOM |
| | ClusterGCN | 67.81 | 78.66 | 58.86/1039 | OOM |
| | GraphFM | 71.53 | 70.76 | 48.03/990.2 | OOM |
| | FreshGNN | 71.51 | 79.04 | 66.28/37.82 | 65.63/46.80 |
| GAT | Vanilla | 70.93 | 79.41 | 66.13/410.5* | 65.16/? |
| | GAS | 70.89 | 77.18 | 57.46/886.7 | OOM |
| | ClusterGCN | 67.02 | 76.49 | 58.05/1230.2 | OOM |
| | GraphFM | 48.26 | 62.87 | OOM | OOM |
| | FreshGNN | 70.43 | 78.87 | 65.42/85.01 | 64.80/83.22 |
| GCN | Vanilla | 71.24 | 78.57 | 65.78/351.5* | 65.24/260.3* |
| | GAS | 71.68 | 76.66 | 53.49/1081 | OOM |
| | ClusterGCN | 68.11 | 78.97 | 53.35/1049 | OOM |
| | GraphFM | 71.71 | 62.87 | 47.08/1011.45 | OOM |
| | FreshGNN | 70.53 | 78.26 | 65.62/36.23 | 64.95/44.33 |

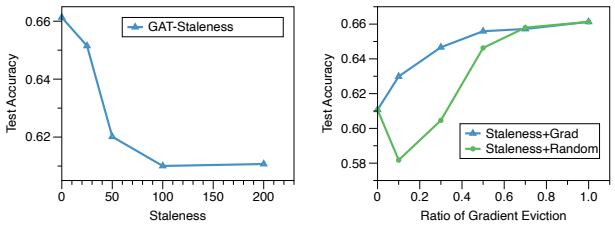
**Hardware and environment.** We evaluate FreshGNN on a server with NVIDIA Tesla A100 GPU and two AMD EPYC 7742 64-Core processors. All experiments run with CUDA 11.3, NCCL [21] v2.11.4, PyTorch [22] 1.10.2.

**Tested models.** FreshGNN can work transparently with different of GNN models that employ multi-layer message passing. In the evaluation, we tested on GCN [16], GraphSAGE [11], and GAT [25] for accuracy. We compare the FreshGNN's accuracy with vanilla training, GAS [8], ClusterGCN [4], and GraphFM [38] to show the advantage of FreshGNN's selective mechanism in Section 7.2. And we test FreshGNN's performance and compare with other GNN systems including DGL [29], PyG [7], and Torch-Direct [19], as well as training methods such as GAS [8] and ClusterGCN [4] in Section 7.4.

**Dataset.** Table 3 shows the dataset used in evaluation. We use the ogbn-MAG240M and ogbn-Papers100M as the large-scale graph dataset to show FreshGNN's capability. ogbn-MAG240M is a citation graph with papers as nodes and citations as edges. This graph has 121M nodes and 1.6B edges. ogbn-MAG240M also contains the information of authors and institutes. By including them together with the papers, the graph is much larger, with 244M nodes and 1.7B edges, named RMAG240M. To compare with And we also carry out algorithm analysis on small datasets such as ogbn-Arxiv and ogbn-Products. We also use the graph structure of FriendSter and Twitter to test the efficiency of FreshGNN.

## 7.2 Overall Results

Table 4 shows the overall results. It contains test accuracy for three models on four dataset, and the train epoch time for web-scale datasets including ogbn-papers100M and ogbn-MAG240M. We compare FreshGNN with vanilla neighbor-sampling-based graph training, which is the same as Algorithm 1 when cache size is set as zero. and other methods for training GNN models on web-scale graphs, including GNN-Auto-Scale (GAS), ClusterGCN, and GraphFM. For test accuracy, on small datasets, different implementations have comparable test accuracy for SAGE and GCN. For GAT, FreshGNN's performance is still high but GraphFM gets much lower. For large datasets, FreshGNN can always get similar results ($\leq$ 1%) with the vanilla method. But other historical based methods show very bad performance, with $5\% - 18\%$ accuracy drop. For efficiency, the vanilla method is slow due to the memory access and sampling overhead during data preparation stage. The methods based on non-selective historical embedding suffers from problem of Out-Of-Memory (OOM) due to their storage of historical embedding for all nodes in the graph. Moreover, their active reads and writes to the historical buffer introduce large overhead. After FreshGNN's optimization, the epoch time for SAGE and GCN is kept under 40 seconds and 50 seconds respectively for ogbn-papers100M and ogbn-MAG240M, which is xx $\times$ and xx $\times$ faster than the best of other methods. The result demonstrate that the selective cache mechanism is both




**(a)** Staleness cache strategy **(b)** Staleness and gradient cache strategy, staleness=200

**Figure 8.** Test accuracy for staleness and gradient cache strategy

important for efficiently running GNN models and keeping the accuracy on large graphs.

### 7.3 Effectiveness of Selective Historical Cache

Figure 8 shows the effect of different cache strategies for training GAT model on ogbn-papers100M. Figure 8a shows staleness strategy, in which the selective cache only evicts the stale embeddings. Staleness indicates the upper-bound of the staleness of historical embedding. If staleness is $s$ and current iteration is $t$, then historical embeddings that are store into cache before iteration $t - s$ will be evicted. Staleness at 0 means the vanilla training that does not use historical cache. We can observe a clear trend that the test accuracy decreases with staleness threshold. Which shows that with historical embeddings getting more stale, there is much larger influence on the model quality. Figure 8b is for gradient strategy, in which the selective cache evicts embeddings according to both staleness and gradient. Ratio of gradient eviction indicates the percentage that the touched embeddings get evicted. 0.2 indiciates to evict 20% of them. $Ratio = 0$ is equivalent to staleness cache strategy and $Ratio = 1$ is training without historical cache. We can observe a rapid increase of accuracy when the ratio grows from 0, and the accuracy increases slower when reaching 1. The trend indicates that gradient cache strategy is able to identify the embeddings that most harmful to model quality and evict them in time.

**Table 5.** Number of clusters and staleness for historical embeddings

| #Clusters | 250 | 500 | 1000 | 2000 | 4000 | 8000 |
|---|---|---|---|---|---|---|
| Staleness | 104.14 | 208.30 | 416.30 | 830.92 | 1658.06 | 3309.79 |
| Data move (GB) | 31.65 | 17.30 | 7.50 | 3.93 | 1.37 | 0.80 |

With Figure 8a showing the relationship between staleness and accuracy, Table 5 can further explain why the non-selective methods such as GAS work badly. It shows the relationship between the number of clusters and average staleness for non-selective historical cache on ogbn-papers100M. We set the number of clusters as 4000 to make the model fit into GPU memory, which results in average staleness at 1658. As from Figure 8a, staleness over 100 leads to accuracy drop of 5%, staleness over 1000 has worse effect on model accuracy. However, in order to limit the staleness under 100, the non-selective methods should fetch more than 30 giga bytes for one iteration. It is slow and may exceed GPU memory capacity. It shows that it is difficult for non-selective methods to work well on large datasets.

**Table 6.** Comparing different graph data structures for pruning

| | CSR | COO | CSR2 |
|---|---|---|---|
| Time per iteration (s) | 20.5 | 3.8E-03 | 2.6E-05 |
| Time proportion (%) | 99.6 | 4.5 | 0.032 |

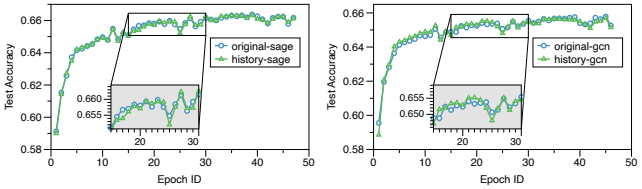



**Figure 9.** Convergence comparison w/ and w/o selective historical cache

Figure 9 shows the test accuracy curve for training GraphSAGE and GCN using vanilla and historical cache respectively. We can observe a very similar curve, which empirically proves the little influence that the cached high-quality historical embeddings have on the model quality.

### 7.4 System Improvement

There are also other types of cache that can save memory access. **??** shows how different types of cache works. In **??**, we can find that employing both raw feature cache and historical cache can largely reduce the data movement for every iteration. On average, raw feature cache can reduce data movement by 44.5%, historical cache can reduce it by 31.2%. FreshGNN uses both of them to reduce the memory access to both hot and cold nodes, it can save the data movement by 63.2%. **??** also proved the effectiveness of applying both raw feature and historical cache. The raw feature cache hit rate is similar on both pruned and unpruned nodes, indicating that the nodes pruned by historical cache are irrelevant with the ones in raw feature cache. Therefore, though raw feature cache has already stored the frequently used nodes, historical cache can still further prune the nodes in the subgraph.

**??** shows the execution time for graph sampling per training iteration. When running graph sampling with single thread and in synchronized manner, it is the performance bottleneck for training that reaches up to 277 seconds per

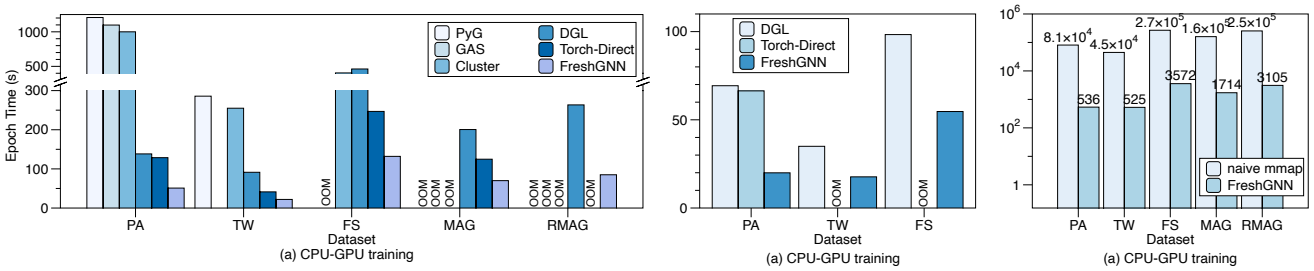



**Figure 10.** Overall speed

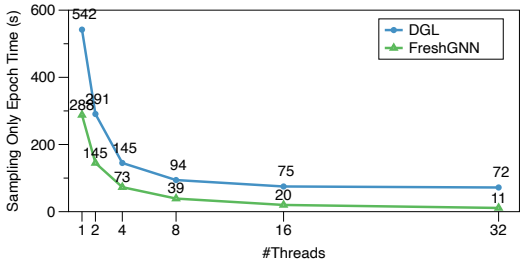



**Figure 11.** Scalibility of graph sampling

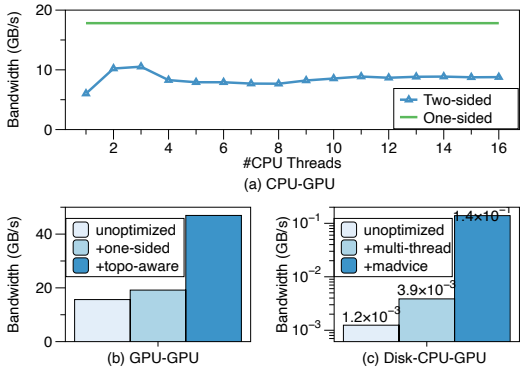



**Figure 12.** Optimizations in data loader

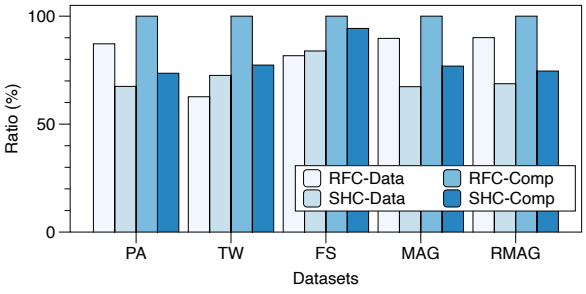



**Figure 13.** Ratio of unpruned data movement and computation

epoch. After putting the graph sampling workload on multiple threads, the graph sampling time is reduced to about ten seconds, which can be fully overlapped by feature fetching and training.

?? shows the memory usage of our system, compared with the size of nodes' raw features, there is only a slight increase in memory consumption, on average 16.9%. But for GAS and GraphFM, the memory usage increase by 2.66×. The short execution time and low memory usage also make it effective and practical to train web-scale graphs on a single machine.

?? shows the efficiency on memory-constraint machines. We limit the usage of memory for node features and historical cache, and compare FreshGNN with the naive implementation that utilizes mmap. The mmap method utilizes linux mmap for disk read, which automatically cache some of the node features in memory according to memory access pattern. However, as the mmap caches node features in naive way without awareness of graph structure, the cache miss rate is high and the memory access overhead is huge. It takes more than 194 minutes for an epoch when memory is 32GB, while FreshGNN only need 6 minutes.

## 8 Related Work

**Sampling methods**. Some methods have been proposed to alleviate the scalability issue using sampling techniques, including *neighbor sampling* that can maintain a fixed number of direct neighbors for each node [1, 5, 11? ], *layer sampling* that samples nodes independently for each layer [2, 42], and *subgraph sampling* that produces a bunch of mini-batches containing a sampled subset of nodes and all the pertaining edges [4, 40]. The usage of sampling techniques have been widely accepted in scalable GNN systems [1, 4, 5, 8, 11, 15, 26, 35, 38, 39], in that it can reduce the problem of neighbor explosion [39] and allow training on GPUs.

**Historical embedding.** Historical embedding is firstly proposed in VR-GCN[1], which aims to reduce both the variance in neighbor sampling (reception field) and the needed raw node features. GAS [8] extends it and applies it to different GNN models including GCN [16], GAT [25], APPNP [17]. However, GAS needs heavy workload in hyperparameter tuning. Moreover, it leads to large accuracy drop on web-scale graph datasets due to staleness and inaccuracy. GraphFM [38] tries to solve the problem using feature momentum, though it can gain better accuracy over GAS, it still cannot work on

large graph dataset. FreshGNN also uses historical embedding, but proposes cache mechanism for it to find out the recent and well learned embeddings and save computation workload from them.

Another way of scaling GNN training is to use the stale historical embeddings. Staleness has long been introduced in asynchronous distributed training of neural networks to update the models by stale gradients in works like SSP [12], MXNet [18], etc. The historical embeddings is slightly different in that its staleness comes from the feature rather than the parameters [28]. Historical embeddings can be used for different purposes. The initial usage of historical embeddings in GNN was VR-GCN [1], aiming at reducing the variance of sampling. GAS [8] stores historical embeddings for every node so as to replace the feature for out-of-batch nodes in sub-graph mini-batch training. Other systems like Dorylus [24], PipeGCN [28], and Sancus [23] use historical embeddings to alleviate the synchronization in distributed full-graph training of GNNs.

**GNN on web-scale-graph**. To train on large graph, previous work studies the optimization from algorithm and system. For algorithm, they find ways to limit the memory access while keeping the similar accuracy. ClusterGCN [4] partitions the graph into several clusters, and each iteration it picks a number of clusters for training. Clustering the nodes limits the number of nodes when finding neighbor nodes during graph sampling. GNS [5] also limits the number of nodes accessed at each iteration using global neighbor. It prioritize the memory access to the global neighbors and reduce the visit to other nodes. For system optimization, PyTorch-Direct [19] enables a GPU-centric data accessing paradigm for GNN training to improve the bandwidth and latency CPU-GPU data transfer. Data-Tiering [20] improves feature cache hit rate by optimizing data placement using the structure of input graph as well as the training process. PaSca [41] proposes a new scalable paradigm to enable neural archtecture search on web-scale graphs. It separates the data pre-processing and post-processing stage from the training stage, and tunes them independently. However, the scalability-friendly paradigm has not been widely accepted by model designers and users.

## 9 Conclusion

In this paper, we propose FreshGNN, a GNN training system powered by a selective historical cache. It combines historical embedding with a cache mechanism by selectively caching the recently trained or well trained nodes' embeddings, and reusing the historical embeddings to replace neighbor expansion and computation. FreshGNN's cache replacement includes staleness, gradient, and hybrid strategy, which stores to the cache the node embeddings that has least influence on the model quality. Evaluation shows that FreshGNN can improve the performance (speed) by 2.23× while getting similar test accuracy on giant graph datasets. And FreshGNN is able to be applied on resource-constraint scenarios for training web-scale graphs.

## References


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

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

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

[4] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-GCN. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery amp Data Mining*. ACM, jul 2019. doi: 10.1145/3292500.3330925. URL https://doi.org/10.1145%2F3292500.3330925.

[5] Jialin Dong, Da Zheng, Lin F. Yang, and Geroge Karypis. Global neighbor sampling for mixed cpu-gpu training on giant graphs, 2021. URL https://arxiv.org/abs/2106.06150.

[6] 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*, pp. 315–324, 2020.

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

[8] 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*, pp. 3294–3304. PMLR, 2021.

[9] 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)*, pp. 551–568, 2021.

[10] Thomas Gaudelet, Ben Day, Arian R. Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy B. R. Hayter, Richard Vickers, Charles Roberts, Jian Tang, David Roblin, Tom L. Blundell, Michael M. Bronstein, and Jake P. Taylor-King. Utilising graph machine learning within drug discovery and development, 2020.

[11] 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, pp. 1025–1035, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

[12] 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.

[13] 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.

[14] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. Accelerating graph sampling for graph machine learning using gpus. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 311–326, 2021.

[15] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E Leiserson, and Jie Chen. Accelerating training and inference of graph neural networks with fast sampling

and pipelining. *Proceedings of Machine Learning and Systems*, 4:172–189, 2022.

[16] 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. URL https://openreview.net/forum?id=SJU4ayYgl.

[17] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. Predict then propagate: Graph neural networks meet personalized pagerank. *arXiv preprint arXiv:1810.05997*, 2018.

[18] 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.

[19] 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 preprint arXiv:2101.07956*, 2021.

[20] 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, pp. 3555–3565, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393850. doi: 10.1145/3534678.3539038. URL https://doi.org/10.1145/3534678.3539038.

[21] NVIDIA. Nccl: Optimized primitives for collective multi-gpu communication, 2022. URL https://github.com/NVIDIA/nccl.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32, pp. 8026–8037. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[23] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. Sancus: sta le n ess-aware c omm u nication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proceedings of the VLDB Endowment*, 15(9):1937–1950, 2022.

[24] 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)*, pp. 495–514, 2021.

[25] 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. URL https://openreview.net/forum?id=rJXMpikCZ.

[26] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius++: Large-scale training of graph neural networks on a single machine. *arXiv preprint arXiv:2202.02365*, 2022.

[27] 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.

[28] 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.

[29] 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.

[30] Felix Wu, Tianyi Zhang, Amauri Holanda de Souza, Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. Simplifying graph convolutional networks, 2019. URL https://arxiv.org/abs/1902.07153.

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

[32] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. Seastar: vertex-centric programming for graph neural networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 359–375, 2021.

[33] 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, pp. 177–186, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304931. doi: 10.1145/1935826.1935863. URL https://doi.org/10.1145/1935826.1935863.

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

[35] 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*, EuroSys '22, pp. 417–434, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391627. doi: 10.1145/3492321.3519557. URL https://doi.org/10.1145/3492321.3519557.

[36] 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*, pp. 974–983, 2018.

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

[38] Haiyang Yu, Limei Wang, Bokun Wang, Meng Liu, Tianbao Yang, and Shuiwang Ji. Graphfm: Improving large-scale gnn training via feature momentum, 2022.

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

[40] 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.

[41] Wentao Zhang, Yu Shen, Zheyu Lin, Yang Li, Xiaosen Li, Wen Ouyang, Yangyu Tao, Zhi Yang, and Bin Cui. Pasca: A graph neural architecture search system under the scalable paradigm. In *Proceedings of the ACM Web Conference 2022*, WWW '22, pp. 1817–1828, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450390965. doi: 10.1145/3485447.3511986. URL https://doi.org/10.1145/3485447.3511986.

[42] 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 Convergence Analysis of the Historical Cache Applied to SGC

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

## A.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 [30] can be formulated as

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

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 $S_\tau, \tau = 1, \cdots, n$ 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 $S_\tau$, $S_{\tau,ii} = \mathbb{I}_{\{\xi_i=\tau\}}$, where $\mathbb{I}$ is the indicator function. So $\mathbb{E}\left[S_{\tau,ii}\right] = \mathbb{E}\left[\mathbb{I}_{\{\xi_i=\tau\}}\right] = p_\tau$ and $\mathbb{E}\left[S_\tau\right] = p_\tau I$. It can be easily shown that $S_i S_j = S_i \cdot \mathbb{I}_{\{i=j\}}$

## A.2 Loss Functions

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

Note that here the $Z$ in the loss function $\ell$ 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 A.1.** *The loss function $\ell(W)$ is $L$-smooth w.r.t. $W$, i.e. $\|\nabla\ell(W^+) - \nabla\ell(W)\|_2 \le L\|W^+ - W\|_2$*

## A.3 Proof of the Main Proposition

**Proposition A.1.** *Under assumption A.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 local minima in $\ell(W)$*

*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

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

Therefore,

$$S_0^{(t)} \nabla_{Z^{(t)}} Loss\left(Z^{(t)}, Y\right) = S_0^{(t)} \nabla_{\tilde{Z}^{(t)}} Loss\left(\tilde{Z}^{(t)}, Y\right)$$

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

$$\nabla\ell(W^{(t)}) = \hat{X}^\top \nabla_{Z^{(t)}} Loss\left(Z^{(t)}, Y\right)$$

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

$$\nabla\tilde{\ell}(W^{(t)}) = \hat{X}^\top S_0^{(t)} \nabla_{\tilde{Z}^{(t)}} Loss\left(\tilde{Z}^{(t)}, Y\right) \tag{7}$$

$$= \hat{X}^\top S_0^{(t)} \nabla_{Z^{(t)}} Loss\left(Z^{(t)}, Y\right) \tag{8}$$

So the update rule for $W$ is

$$W^{(t+1)} \leftarrow W^{(t)} - \eta \hat{X}^\top S_0^{(t)} \nabla_{\tilde{Z}^{(t)}} Loss\left(\tilde{Z}^{(t)}, Y\right) \tag{9}$$

$$= W^{(t)} - \eta \hat{X}^\top S_0^{(t)} \nabla_{Z^{(t)}} Loss\left(Z^{(t)}, Y\right) \tag{10}$$

Since $\ell$ is Lipschitz smooth, it holds true that

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

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

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

Then by the law of total expectation,

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

Therefore,

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

$$\sum_{t=0}^{T}\left\|E\left[\nabla l\left(W^{(t)}\right)\right]\right\|_F^2 \leq \frac{\ell\left(W^{(0)}\right) - \ell\left(W^*\right)}{\eta p_0\left(1 - \frac{L\eta}{2}\right)}$$

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

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

# B Appendix

The used datasets are all public available, below are the links of them.

1. arxiv: https://ogb.stanford.edu/docs/nodeprop/#ogbn-arxiv
2. reddit: https://paperswithcode.com/dataset/reddit
3. products: https://ogb.stanford.edu/docs/nodeprop/#ogbn-products
4. papers100M: https://ogb.stanford.edu/docs/nodeprop/#ogbn-papers100M
5. MAG240M: https://ogb.stanford.edu/docs/lsc/mag240m/
6. citation: https://ogb.stanford.edu/docs/linkprop/#ogbl-citation2
7. WikiKG: https://ogb.stanford.edu/docs/lsc/wikikg90mv2/
8. Twitter: https://snap.stanford.edu/data/twitter-2010.html
9. FriendSter: http://snap.stanford.edu/data/com-Friendster.html