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

Kezhao Huang[*]
Tsinghua University
hkz20@mails.tsinghua.edu.cn

Haitian Jiang[*]
New York University
hj2533@nyu.edu

Minjie Wang
Amazon
minjiw@amazon.com

Guangxuan Xiao
MIT
xgx@mit.edu

David Wipf
Amazon
daviwipf@amazon.com

Xiang Song
Amazon
xiangsx@amazon.com

Quan Gan
Amazon
quagan@amazon.com

Zengfeng Huang
Fudan University
huangzf@fudan.edu.cn

Jidong Zhai
Tsinghua University
zhaijidong@tsinghua.edu.cn

Zheng Zhang
Amazon
zhaz@amazon.com

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 `Papers100M` and `MAG240M` by 2.7× 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 [1], recommender systems [2], fraud detection [3], drug discovery [4] and more. The favorable predictive 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 [5], whereby updating any particular node embedding requires collecting and aggregating the embeddings of its neighbors.

[*]Equal contribution.

K. Huang et al., FreshGNN: Reducing Memory Access via Stable Historical Embeddings for Graph Neural Network Training (Extended Abstract). Presented at the Second Learning on Graphs Conference (LoG 2023), Virtual Event, November 27–30, 2023.

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.

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 [6–9]. 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 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 [10–12]; 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 [6], layer-wise sampling [13, 14], and graph-wise sampling [7, 8]. 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 [15]. Meanwhile, the second line of work [16–18] such as GAS [17] 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. 3, its **non-selective** way of reusing historical embeddings 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.

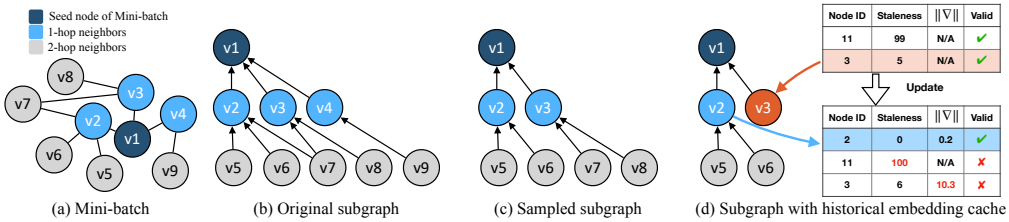
To this end, we propose a new mini-batch GNN training solution with system and algorithm co-design for efficiently handling large graphs while preserving predictive performance. As our starting point, we narrow the root cause of accuracy degradation when using historical embeddings to the non-negligible accumulation of estimation error between true and approximate representations computed using the history. As prior related work has no practical 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.

Our contribution We propose a mini-batch training algorithm for GNNs that achieves scalability without compromising model accuracy. 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 lead to large approximation errors and subsequently, poor predictive accuracy. We provide a comprehensive empirical evaluation of FreshGNN across common baseline GNN architectures, large-scale graph datasets, and hardware configurations. The results from evaluation demonstrate that FreshGNN can closely maintain the accuracy (accuracy drop controlled within 1%) of non-approximate neighbor sampling while training 2.7× faster than state-of-the-art baselines.

2 Historical Embedding Cache

The historical embedding cache design is informed by the following question: *What is the suitable cache policy for checking in and out node embeddings for accuracy?* Caching intermediate node embeddings is fundamentally different than caching raw node features. Unlike raw node features

which stay unchanged, the embeddings are constantly updated during model training, meaning the quality of cached embedding will influence the accuracy of trained model. Compared with GAS's non-selective way of embedding reusing, we propose to **selectively** cache and reuse the *stable embeddings* that are more reliable for future reuse. The following introduces the workflow of the historical embedding cache and the cache policy for being selective.



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

Figure 1: Illustration of historical embedding cache using an example mini-batch graph.

2.1 Historical Cache Workflow

Figure 1 elucidates the workflow of the historical embedding cache using a toy example. Here node $v1$ is selected as the seed node of the current mini-batch as in Figure 1(a). Computing $v1$'s embedding requires recursively collecting information from multi-hop neighbors as illustrated by the subgraph in Figure 1(b). And as mentioned previously, neighbor sampling can reduce this subgraph size as shown in Figure 1(c). Figure 1(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.

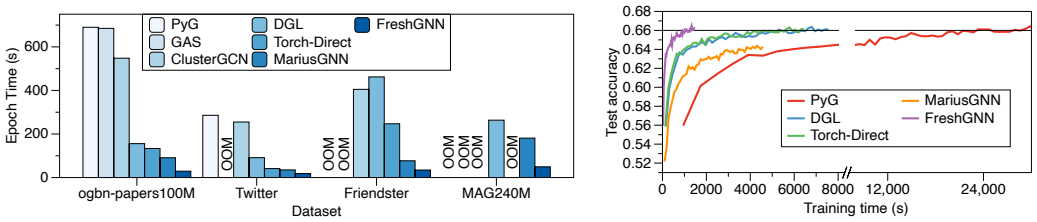
2.2 Historical Embedding Cache Policy

To achieve caching embeddings selectively, it is important to identify the stable embeddings (i.e., embeddings that are safe to be reused during training). However, it is challenging to quantify the stability of embeddings, where in the context of GNN training, the notion of stability is calibrated by the difference between a true node embedding and the corresponding cached one. The 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 data loading and computation we are trying to avoid.

FreshGNN adopts a *gradient-based criteria* to identify stable embeddings in a light-weight way. The gradient of node embeddings are naturally calculated to update the weight parameters during training, therefore fetching the embedding gradients is zero-cost. More importantly, the gradients are the feedbacks from model training that can reflect the stability of the embeddings. A near-zero gradient magnitude indicates that the embedding leads to the correct predictions and only needs little change at this iteration of training. Compared to the other embeddings with gradient of large magnitude, this embedding is more stable and can be cached for further reuse. Based on the absolute value of embedding gradients, FreshGNN is able to compare the stability of embeddings at each iteration in order to store stable embeddings newly produced and invalidate unstable ones in the cache.

Based on the intuition of using gradient as indicator of embedding stability, we formulate the embedding cache policy for accuracy as below. Given a mini-batch graph, denote the set of nodes at layer l as $\mathcal{V}^{(l)}$ and the set of cached nodes as $\mathcal{V}_{cache}$. 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. FreshGNN admits nodes with small absolute values of 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, if any of these happen to already be in the cache, they will now be evicted.

With the weight parameters updated at every iteration, the embeddings produced using parameters from previous iterations can be stale. To maintain the accuracy of the model, besides the gradient



(a) Epoch time for training a GraphSAGE model using a single GPU

(b) Test accuracy curve for GraphSAGE on `Papers100M`.

Table 1: Test accuracy of algorithms compared with target accuracy obtained by neighbor sampling.

Model	SAGE		GAT		GCN	
Dataset	`Papers100M`	`MAG240M`	`Papers100M`	`MAG240M`	`Papers100M`	`MAG240M`
Target	66.4	66.1	66.1	65.2	65.8	65.2
GAS	-8.17	OOM	-8.67	OOM	-12.3	OOM
ClusterGCN	-7.57	OOM	-8.08	OOM	-12.4	OOM
GraphFM	-18.4	OOM	OOM	OOM	-18.7	OOM
MariusGNN	-3.43	-2.97	OOM	OOM	-2.04	-2.37
Ours	**-0.15**	**-0.51**	**-0.71**	**-0.36**	**-0.16**	**-0.29**

policy, FreshGNN also bounds the *staleness* of the embeddings. The staleness is set to zero when an embedding is admitted to cache and it will increase by one at each iteration. FreshGNN treats any embedding with staleness larger than a threshold t_{stale} as being out-dated and evicts it from cache.

3 Evaluation

We employed three widely-used GNN architectures for our experiments: GraphSAGE [6], GAT [19], and GCN [20], . To measure their baseline model performance, we train them using mini-batch neighbor sampling in DGL [21]. All datasets are from OGB benchmark [22, 23], including `Papers100M` and `MAG240M` with over 100M nodes. We also use `Twitter` [24] and `FriendSter` [25] to evaluate system speed. See Appendix A for detailed dataset settings. The compared systems include state-of-the-art alternatives for GNN mini-batch training, including DGL [21], PyG [26], PyTorch-Direct [10], MariusGNN [27], and representative mini-batch training algorithms such as ClusterGCN [7], GAS [17], and GraphFM [18].

Figure 2a compares the time for training a GraphSAGE model for one epoch on the four large-scale graph datasets using a single GPU (NVIDIA-A100-40GB). FreshGNN significantly outperforms all the other baselines across all the datasets by 2.73× over the best among others. FreshGNN is not only fast but also reliably converges to the desired target accuracy. Figure 2b plots the time-to-accuracy curve of different training systems. FreshGNN can reach the same accuracy in 25 minutes while the slowest baseline (PyG) takes more than 6 hours.

Table 1 compares the test accuracy of FreshGNN with other mini-batch training algorithms. When scaling to larger graphs such as `Papers100M`, most of the baselines experience a substantial accuracy drop (from 7% to 18%) while running out of memory on `MAG240M`. Only MariusGNN can run on all datasets, but with lower accuracy (over 2% drop) and longer epoch time (Figure 2a). By contrast, FreshGNN only experiences a less than 1% accuracy difference across all datasets compared with base models.

4 Conclusion

In this paper, we propose FreshGNN, a mini-batch training algorithm for GNNs on large, real-world graphs. At the core of our design is a selective historical cache that stores and reuses the stable 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. FreshGNN is able to accelerate the training speed of GNNs on large graphs by 2.7× over state-of-the-art systems on GPU, with less than 1% influence on model accuracy.

References

[1] Zhengdao Chen, Xiang Li, and Joan Bruna. Supervised community detection with line graph neural networks. *arXiv preprint arXiv:1705.08415*, 2017. 1

[2] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. Graph neural networks in recommender systems: a survey. *ACM Computing Surveys (CSUR)*, 2020. 1

[3] 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. 1

[4] 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. 1

[5] 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. 1

[6] 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. ISBN 9781510860964. 2, 4

[7] 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. 2, 4

[8] 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. 2

[9] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. GraphSAINT: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019. 2

[10] 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. 2, 4

[11] 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. ISBN 9781450393850. doi: 10.1145/3534678.3539038. URL https://doi.org/10.1145/3534678.3539038.

[12] 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. 2

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

[14] 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. 2

[15] 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. 2

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

[17] 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. 2, 4

[18] Zekun Li, Shu Wu, Zeyu Cui, and Xiaoyu Zhang. GraphFM: Graph factorization machines for feature interaction modeling. *arXiv:2105.11866*, 2021. 2, 4

[19] 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. 4

[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. URL https://openreview.net/forum?id=SJU4ayYgl. 4

[21] 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. 4

[22] 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. 4, 6

[23] 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. 4, 6

[24] 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. ISBN 9781450304931. doi: 10.1145/1935826.1935863. URL https://doi.org/10.1145/1935826.1935863. 4, 6

[25] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015. 4, 6

[26] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. 4

[27] 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. 4

A Appendix

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.[2]	#Class
`Papers100M` [22]	111M	1.6B	128	172
`MAG240M` [23]	244.2M	1.7B	768	153
`Twitter` [24]	41.7M	1.5B	768	64
`Friendster` [25]	65.6M	1.8B	768	64

[2]The first three datasets use float32 while the latter three use float16, which follows common practice [22, 23].