# RSC: Accelerate Graph Neural Networks Training via Randomized Sparse Computations

**Zirui Liu** [1] **Shengyuan Chen** [2] **Kaixiong Zhou** [1] **Daochen Zha** [1] **Xiao Huang** [2] **Xia Hu** [1]

## Abstract

Training graph neural networks (GNNs) is extremely time-consuming because sparse graph-based operations are hard to be accelerated by community hardware. Prior art successfully reduces the computation cost of dense matrix based operations (e.g., convolution and linear) via sampling-based approximation. However, unlike dense matrices, sparse matrices are stored in an irregular data format such that each row/column may have a different number of non-zero entries. Thus, compared to the dense counterpart, approximating sparse operations has two unique challenges **(1)** we cannot directly control the efficiency of approximated sparse operation since the computation is only executed on non-zero entries; **(2)** sampling sparse matrices is much more inefficient due to the irregular data format. To address the issues, our key idea is to control the accuracy-efficiency trade-off by optimizing computation resource allocation layer-wisely and epoch-wisely. For the **first** challenge, we customize the computation resource to different sparse operations, while limiting the total used resource below a certain budget. For the **second** challenge, we cache previously sampled sparse matrices to reduce the epoch-wise sampling overhead. To this end, we propose **R**andomized **S**parse **C**omputation. In practice, `RSC` can achieve up to $11.6\times$ speedup for a single sparse operation and $1.6\times$ end-to-end wall-clock time speedup with almost no accuracy drop. Codes are available at `https://github.com/warai-0toko/RSC-ICML`.

[1]Department of Computer Science, Rice University, Houston, TX, USA [2]Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Hong Kong SAR. Correspondence to: Xia Hu <xia.hu@rice.edu>.

*Proceedings of the $40^{th}$ International Conference on Machine Learning*, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

## 1. Introductions

Graph Neural Networks (GNNs) have achieved great success across different graph-related tasks (Hamilton et al., 2017; Hu et al., 2020; Ying et al., 2018; Jiang et al., 2022; Zhou et al., 2022; 2023). However, despite its effectiveness, the training of GNNs is very time-consuming. Specifically, GNNs are characterized by an interleaved execution that switches between the aggregation and update phases. Namely, in the aggregation phase, every node aggregates messages from its neighborhoods at each layer, which is implemented based on *sparse matrix-based operations* (Fey & Lenssen, 2019; Wang et al., 2019). In the update phase, each node will update its embedding based on the aggregated messages, where the update function is implemented with *dense matrix-based operations* (Fey & Lenssen, 2019; Wang et al., 2019). In Figure 1, `SpMM` and `MatMul` are the sparse and dense operations in the aggregation and update phases, respectively. Through profiling, we found that the aggregation phase may take more than $90\%$ running time for GNN training. This is because the sparse matrix operations in the aggregation phase have many random memory accesses and limited data reuse, which is hard to be accelerated by community hardwares (e.g., CPUs and GPUs) (Duan et al., 2022b; Han et al., 2016; Duan et al., 2022a). Thus, training GNNs with large graphs is often time-inefficient.

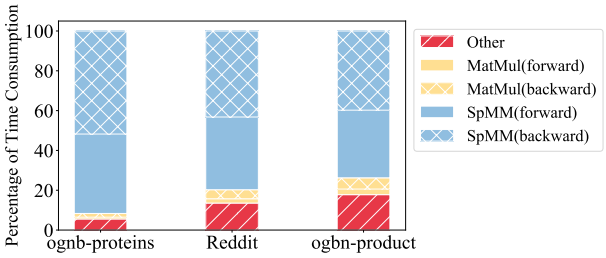


*Figure 1:* The time profiling of a two-layer GCNs on different datasets. `SpMM` may take $70\% \sim 90\%$ of the total time. We measure the time on a single NVIDIA RTX3090 (24GB). The detailed software and hardware information can be found in Appendix D.

Existing works towards this problem can be roughly divided into three categories. First, some works propose distributed GNNs training systems, which focus on minimizing the

communication cost among hardware (Zheng et al., 2020; Ramezani et al., 2022; Wan et al., 2022b; Md et al., 2021; Wan et al., 2022a). Second, another research line optimizes the memory access pattern of sparse operations via coalescing the memory access and fusing consecutive operations (Zhang et al., 2022; Huang et al., 2020a; Rahman et al., 2021; Wang et al., 2021). Third, some other works try to accelerate the training process from the optimization aspect, i.e., using fewer iterations to converge (Narayanan et al., 2022; Cong et al., 2020; Xu et al., 2021; Cai et al., 2021).

In parallel, an orthogonal direction is to replace the expensive operations with their faster-approximated versions (Adelman et al., 2021; Drineas et al., 2006b). The key idea is to sub-sample tensors onto low dimensional spaces and perform the original operations here. For example, for the linear operation between two matrices $\mathbf{A} \in \mathbb{R}^{n\times m}$ and $\mathbf{B} \in \mathbb{R}^{m\times q}$, we first obtain $\mathbf{A}' \in \mathbb{R}^{n\times k}$ and $\mathbf{B}' \in \mathbb{R}^{k\times q}$ ($k < m$) by picking $k$ representative columns of $\mathbf{A}$ and the corresponding rows of $\mathbf{B}$ (Drineas et al., 2006b). Then we approximate $\mathbf{AB} \approx \mathbf{A}'\mathbf{B}'$. With this procedure, the number of floating-point operations (FLOPs) and memory access are both reduced. Based on the idea, previous work successfully accelerates the dense matrix based operations, such as convolution and linear operations (Adelman et al., 2021). The approximated operation can plug-and-play replace the exact operation to improve per-operation efficiency, and thus is compatible with most of the efficient training methods. Despite the potential, this perspective however has not been explored for the sparse operations in GNNs.

The approximation method reduces the computational complexity at the cost of giving noisy outputs. Thus, there naturally exists an **accuracy-efficiency trade-off**. Compared to approximating dense matrix operations, there are two unique challenges to optimizing the trade-off for approximated sparse operations. *First*, unlike the previous example of approximating linear operation, $k$ cannot directly control the efficiency (FLOPs) for sparse operations. This is because, for dense matrices, each row/column has the same amount of parameters. Thus the reduction of FLOPs in approximated dense operations is determined by the dimensions of the sub-sampled matrices (i.e., $k$). However, in sparse operations, each row/column in the sparse adjacency matrix has different numbers of non-zero entries, and the computation is only executed on non-zero entries (i.e., irregular data format). Thus, the reduction of FLOPs in the sparse operations is decided by the selection of representative rows/columns. It lacks a mechanism to directly control the efficiency-accuracy trade-off for each sparse operation. *Second*, compared to the dense counterpart, sub-sampling (i.e., slicing) the sparse matrix is much more time-consuming due to its irregular data format (Han et al., 2016; Fey & Lenssen, 2019), which counteracts the acceleration from the FLOPs reduction.

To this end, we propose **R**andomized **S**parse **C**omputation, dubbed `RSC` , the first approximation framework tailored for efficient GNN training. Our core idea is to control the trade-off by optimizing the computation resource allocation at the "global" level. Specifically, to tackle the first challenge, ***at the layer-wise level***, we propose to customize the FLOPs of each sparse operation while limiting the total FLOPs under a certain budget. The rationale behind this strategy is that each operation may have a different contribution to the model accuracy. Thus, we could to assign more computational resources to "important" operations under a certain budget. More concretely, we frame it as a constraint optimization problem. Then we propose a greedy algorithm to solve it efficiently. To tackle the second challenge, ***at the epoch-wise level***, we found that the selection of representative row/columns tends to remain similar across nearby iterations. Based on this finding, we develop a caching mechanism to reuse the previously sampled sparse matrix across nearby iterations to reduce per-epoch sampling time. ***Finally***, inspired by the recent finding that the final stage of training usually needs smaller noise to help convergence (Li et al., 2019; Dao et al., 2022), we propose to use approximated sparse operation during most of the training process, while switching back to the original sparse operation at the final stage. This switching mechanism significantly reduces the accuracy drop, at the cost of slightly less speedup. We summarize our contributions as follows:

- We accelerate the training of GNNs from a new perspective, namely, replacing the expensive sparse operations with their faster-approximated versions.
- Instead of focusing on balancing the efficiency-accuracy trade-off at the operation level, we control the trade-off through optimizing resource allocation at the layer-wise and epoch-wise levels.
- We propose a caching mechanism to reduce the cost of sampling sparse matrices by reusing previous results.
- Extensive experiments have demonstrated the effectiveness of the proposed method. Particularly, `RSC` can achieve up to $11.6\times$ speedup for a single sparse operation and a $1.6\times$ end-to-end wall-clock time speedup with negligible ($\approx 0.3\%$) accuracy drop.

## 2. Background and Preliminary

### 2.1. Graph Neural Networks

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph with $\mathcal{V} = (v_1, \cdots, v_{|\mathcal{V}|})$ and $\mathcal{E} = (e_1, \cdots, e_{|\mathcal{E}|})$ being the set of nodes and edges, respectively. Let $\boldsymbol{X} \in \mathbb{R}^{|\mathcal{V}|\times d}$ be the node feature matrix. $\boldsymbol{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ is the graph adjacency matrix, where $\boldsymbol{A}_{i,j} = 1$ if $(v_i, v_j) \in \mathcal{E}$ else $\boldsymbol{A}_{i,j} = 0$.

$\tilde{\boldsymbol{A}} = \tilde{\boldsymbol{D}}^{-\frac{1}{2}}(\boldsymbol{A}+\boldsymbol{I})\tilde{\boldsymbol{D}}^{-\frac{1}{2}}$ is the normalized adjacency matrix, where $\tilde{\boldsymbol{D}}$ is the degree matrix of $\boldsymbol{A}+\boldsymbol{I}$. GNNs recursively update the embedding of a node by aggregating embeddings of its neighbors. For example, the forward pass of the $l^{\text{th}}$ Graph Convolutional Network (GCN) layer (Kipf & Welling, 2017) can be defined as:

$$\boldsymbol{H}^{(l+1)} = \text{ReLU}(\tilde{\boldsymbol{A}}\boldsymbol{H}^{(l)}\boldsymbol{\Theta}^{(l)}), \tag{1}$$

where $\boldsymbol{H}^{(l)}$ is the node embedding matrix at the $l^{\text{th}}$ layer and $\boldsymbol{H}^{(0)} = \boldsymbol{X}$. $\boldsymbol{\Theta}^{(l)}$ is the weight matrix of the $l^{\text{th}}$ layer.

In practice, $\tilde{\boldsymbol{A}}$ is often stored in the sparse matrix format, e.g., compressed sparse row (CSR) (Fey & Lenssen, 2019). From the implementation aspect, the computation of Equation (1) can be described as:

$$\boldsymbol{H}^{(l+1)} = \text{ReLU}\left(\texttt{SpMM}\left(\tilde{\boldsymbol{A}}, \texttt{MatMul}(\boldsymbol{H}^{(l)}, \boldsymbol{\Theta}^{(l)})\right)\right),$$

where $\texttt{SpMM}(\cdot,\cdot)$ is the Sparse-Dense Matrix Multiplication and $\texttt{MatMul}(\cdot,\cdot)$ is the Dense Matrix Multiplication. Sparse operations, such as `SpMM` , have many random memory accesses and limited data reuse. Thus they are much slower than the dense counterpart (Han et al., 2016; Duan et al., 2022b). To get a sense of the scale, we show in Figure 1 that for GCNs, `SpMM` may take roughly $70\% \sim 90\%$ of the total training time.

### 2.2. Fast Approximated `MatMul` with Sampling

Let $\boldsymbol{X} \in \mathbb{R}^{n\times m}$, $\boldsymbol{Y} \in \mathbb{R}^{m\times q}$. The goal is to efficiently estimate the matrix production $\boldsymbol{XY}$. Truncated Singular Value Decomposition (SVD) outputs provably optimal low-rank estimation of $\boldsymbol{XY}$ (Adelman et al., 2021). However, SVD is almost as expensive as matrix production itself. Instead, the sampling algorithm is proposed to approximate the matrix product $\boldsymbol{XY}$ by sampling $k$ columns of $\boldsymbol{X}$ and corresponding rows of $\boldsymbol{Y}$ to form smaller matrices, which are then multiplied as usual (Drineas et al., 2006b). This algorithm reduces the computational complexity from $\mathcal{O}(mnq)$ to $\mathcal{O}(knq)$. Specifically,

$$\begin{aligned}\boldsymbol{XY} = \sum_{i=1}^{m} \boldsymbol{X}_{:,i}\boldsymbol{Y}_{i,:} &\approx \sum_{t=1}^{k} \frac{1}{s_t}\boldsymbol{X}_{:,i_t}\boldsymbol{Y}_{i_t,:} \\ &= \texttt{approx}(\boldsymbol{XY}),\end{aligned} \tag{2}$$

where $\boldsymbol{X}_{:,i} \in \mathbb{R}^{n\times 1}$ and $\boldsymbol{Y}_{i,:} \in \mathbb{R}^{1\times q}$ are the $i^{\text{th}}$ column and row of $\boldsymbol{X}$ and $\boldsymbol{Y}$, respectively. *In this paper, we call* $(\boldsymbol{X}_{:,i}, \boldsymbol{Y}_{i,:})$ *the* $i^{\text{th}}$ *column-row pair.* $k$ is the number of samples ($1 \le k \le m$). $\{p_i\}_{i=1}^m$ is a probability distribution over the column-row pairs. $i_t \in \{1, \cdots m\}$ is the index of the sampled column-row pair at the $t^{\text{th}}$ trial. $s_t$ is the scale factor. Theoretically, (Drineas et al., 2006b) shows that if we set $s_t = \frac{1}{kp_{i_t}}$, then we have $\mathbb{E}[\texttt{approx}(\boldsymbol{XY})] = \boldsymbol{XY}$. Further, the approximation error $\mathbb{E}[||\boldsymbol{XY} - \texttt{approx}(\boldsymbol{XY})||_F]$ is minimized when the sampling probabilities $\{p_i\}_{i=1}^m$ are proportional to the product of the column-row Euclidean norms (Drineas et al., 2006b):

$$p_i = \frac{||\boldsymbol{X}_{:,i}||_2\, ||\boldsymbol{Y}_{i,:}||_2}{\sum_{j=1}^{m} ||\boldsymbol{X}_{:,j}||_2\, ||\boldsymbol{Y}_{j,:}||_2}. \tag{3}$$

#### 2.2.1. Top-$k$ sampling

The above sampling-based method is originally developed for accelerating the general application of `MatMul` (Drineas et al., 2006b). Directly applying it to neural networks may be sub-optimal since it does not consider the characteristic of neural network weights. Based on the empirical observation that the distribution of weights remains centered around zero during training (Glorot & Bengio, 2010; Han et al., 2015), (Adelman et al., 2021) proposes a **top-$k$ sampling** algorithm: *Picking $k$ column-row pairs with the largest* $\frac{||\boldsymbol{X}_{:,i}||_2\, ||\boldsymbol{Y}_{i,:}||_2}{\sum_{j=1}^{m} ||\boldsymbol{X}_{:,j}||_2\, ||\boldsymbol{Y}_{j,:}||_2}$ *deterministically without scaling.*

Equivalently, it means $p_i$ of column-row pairs with the $k$-largest value in Equation (3) equals 1, otherwise it equals 0. And $s_{i_t}$ is a constant 1. Albeit without the scaling while sampling column-row pairs deterministically, under on the assumption of zero-centered weight distribution, (Adelman et al., 2021) theoretically show that *top-$k$ sampling still yields an unbiased estimation of $\boldsymbol{XY}$ with minimal approximation error.* Consequently, the top-$k$ sampling algorithm empirically shows a significantly lower accuracy drop when approximating the convolution and linear operations in the neural networks (Adelman et al., 2021).

In the next section, we explore how to approximate the expensive sparse operation via the top-$k$ sampling.

## 3. The Proposed Framework

The overview of `RSC` is shown in Figure 2, where we use the computation graph of GCN as an example. We first explore which `SpMM` in the computation graph can be replaced with its approximated version (Section 3.1). Then since GNNs have multiple `SpMM` and each of them may have different importance to the model performance, we then automatically allocate computation resources to different `SpMM` (Section 3.2). Finally, we explore two simple and effective tricks for improving `RSC` , including a caching mechanism to reduce the overhead of sampling sparse matrices (Section 3.3.1) and a switching mechanism to reduce the accuracy drop (Section 3.3.2).

### 3.1. Where to Apply the Approximation

#### 3.1.1. Experimental analysis

Each sparse operation is executed twice at each training step, i.e., one in the forward pass and the other one

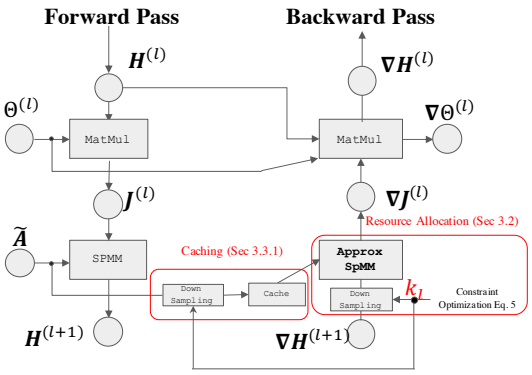


*Figure 2:* Overview of RSC . For convenience, ReLU is ignored. RSC only replace the SpMM in the backward pass with its approximated version using top-$k$ sampling (Section 3.1). $k_l$ is the number of samples for top-$k$ sampling at the $l^{\text{th}}$ layer, which is automatically allocated (Section 3.2). To reduce the overhead of sampling, we also cache the sampled graph and reuse it across nearby iterations (Section 3.3).

in the backward pass. As shown in Figure 2, here we take SpMM in the $l^{\text{th}}$ GCN layer as an example, the forward one is $\boldsymbol{H}^{(l+1)} = \text{ReLU}(\texttt{SpMM}(\tilde{\boldsymbol{A}}, \boldsymbol{J}^{(l)}))$, where $\boldsymbol{J}^{(l)} = \texttt{MatMul}(\boldsymbol{H}^{(l)}, \boldsymbol{\Theta}^{(l)})$ is the intermediate node representations. And the backward one is $\nabla\boldsymbol{J}^{(l)} = \texttt{SpMM}(\tilde{\boldsymbol{A}}^\top, \nabla\boldsymbol{H}^{(l+1)})$. $\nabla\boldsymbol{J}^{(l)}$ and $\nabla\boldsymbol{H}^{(l)}$ are the gradient with respect to $\boldsymbol{J}^{(l)}$ and $\boldsymbol{H}^{(l)}$, respectively.

Even though the approximation method itself is statistically unbiased, replacing the exact sparse operation with their faster-approximated versions still injects noise to the computation graph. As we analyzed above, each SpMM is executed twice in the training step. Below we first experimentally analyze the impact of the injected noise in the forward pass and the backward pass. As shown in Table 1, we apply top-$k$ sampling to approximate the SpMM in the forward pass, backward pass, or both, respectively.

*Table 1:* Preliminary results on approximating SpMM via top-$k$ sampling. The model is a two-layer GCN, and the dataset is Reddit. Here we set the $k$ as $0.1|\mathcal{V}|$ across different layers.

| Method | Reddit |
|---|---|
| without approximation | 95.39±0.04 |
| only forward | 16.45±0.39 |
| **only backward** | **95.25±0.03** |
| forward and backward | 80.74±1.00 |

From Table 1, the accuracy drop is negligible if we only replace SpMM in the backward pass. Notably, if we apply approximation in both the forward and backward pass, the result is significantly better than only applying top-$k$ sampling in the forward pass. The reason is that when only applying approximation in the forward pass, some row/columns are not included in the computation graph, so intuitively these row/columns should be excluded in the backward pass. "forward and backward" result in Table 1 is built based on this intuition such that in the backward pass, we use the column-row pairs sampled in the forward pass to compute the gradient (Adelman et al., 2021). However, it is still not comparable to the result of applying approximation only in the backward pass. Below we mathematically analyze the reason behind the results in Table 1.

#### 3.1.2. Theoretical analysis

We first analyze the case of approximating the sparse operations in the forward pass. Namely, replacing $\texttt{SpMM}(\tilde{\boldsymbol{A}}, \boldsymbol{J}^{(l)})$ with $\texttt{approx}(\tilde{\boldsymbol{A}}\boldsymbol{J}^{(l)})$. We note that we have $\mathbb{E}[f(x)] \neq f(\mathbb{E}[x])$ for any non-linear function $f(\cdot)$, e.g., $\mathbb{E}[x^2] \neq \mathbb{E}^2[x]$. Thus, even when the approximation method gives an unbiased estimation, i.e., $\mathbb{E}[\texttt{approx}(\tilde{\boldsymbol{A}}\boldsymbol{J}^{(l)})] = \tilde{\boldsymbol{A}}\boldsymbol{J}^{(l)}$, the node embeddings $\boldsymbol{H}^{(l+1)}$ are still biased since the activation function is non-linear. To see this,

$$\begin{aligned}\mathbb{E}[\boldsymbol{H}^{(l+1)}] &= \mathbb{E}[\text{ReLU}(\texttt{approx}(\tilde{\boldsymbol{A}}\boldsymbol{J}^{(l)})]) \\ &\neq \text{ReLU}(\mathbb{E}[\texttt{approx}(\tilde{\boldsymbol{A}}\boldsymbol{J}^{(l)})]) = \boldsymbol{H}^{(l+1)}.\end{aligned}$$

Thus, if we apply the approximation for the SpMM in the forward pass, the bias will be propagated layer-by-layer and cause significantly worse results. For the case of only approximating the sparse operation in the backward pass, we have the following proposition:

**Proposition 3.1** (Proof in Appendix A)**.** *If the approximation method is itself unbiased, and we only replace the* SpMM *in the backward pass with its approximated version, while leaving the forward one unchanged, then the calculated gradient is provably unbiased.*

The high-level idea is that the gradient of the activation function in the backward pass is only related to the pre-activations in the forward pass, and thus is independent of the approximation error introduced in the backward pass. Due to the page limit, we also discuss why sampling-based approximation is suitable for accelerating GNNs in Appendix A. As suggested by our theoretical and empirical analysis, as shown in Figure 2, *we only approximate the sparse operations in the backward pass, while leaving all other operations unchanged.*

### 3.2. How to Apply the Approximation

As we mentioned, for sparse operations, the acceleration is decided by the selection of sampled column-row pairs. To see this, as shown in Figure 3, suppose we use top-$k$ sampling to approximate $\texttt{SpMM}(\tilde{\boldsymbol{A}}^\top, \nabla\boldsymbol{H})$. Since the computations are only executed on the non-zero entries, so selecting the orange pairs (i.e., pair 1 and 3) will result in $\frac{3}{7}\times$ less computational cost (FLOPs) compared to selecting

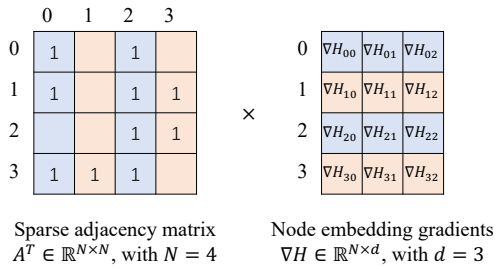


*Figure 3:* For approximated sparse operations, the acceleration is decided by the selection of column-row pairs.

the blue pair (i.e., pair 0 and 2). For both the orange and blue cases, we have $k = 2$. Thus, the number of samples $k$ cannot directly constrain the FLOPs for each individual operation. Moreover, a GNN has multiple operations (or layers), and the model accuracy has a different sensitivity to the approximation error at different layers. To optimize the accuracy-efficiency trade-off, our key idea is to customize the computation resources (i.e., FLOPs) for each layer by adjusting the number of samples $k_l$ in the $l$-th layer. In this way, we minimize the impact of approximation, while limiting the overall FLOPs under a certain budget. Based on the idea, we frame the resource allocation problem as the following constrained optimization problem:

$$\min_{\{k_l\}} -\sum_{l=1}^{L} \sum_{i \in \mathrm{Top}_{k_l}} \frac{\|\tilde{\boldsymbol{A}}_{:,i}^{\top}\|_2 \|\nabla \boldsymbol{H}_{i,:}^{(l+1)}\|_2}{\|\tilde{\boldsymbol{A}}\|_F \|\nabla \boldsymbol{H}^{(l+1)}\|_F}, \tag{4a}$$

$$s.t. \sum_{l=1}^{L} \sum_{i \in \mathrm{Top}_{k_l}} \#nnz_i * d_l \leq C \sum_{i=1}^{L} |\mathcal{E}| d_l, \tag{4b}$$

where $C$ is the budget ($0 < C < 1$) that controls the overall reduced FLOPs. $k_l$ is the number of samples for the top-$k$ sampling at the $l$-th layer. $d_l$ is the hidden dimensions of $l$-th layer, and $\#nnz_i$ is the number of non-zero entries at the $i$-th column of $\tilde{\boldsymbol{A}}^{\top}$. $\mathrm{Top}_{k_l}$ is the set of indices associated with the $k_l$ largest $\|\tilde{\boldsymbol{A}}_{:,i}^{\top}\|_2 \|\nabla \boldsymbol{H}_{i,:}^{(l+1)}\|_2$.

Equation (4a) is equivalent to minimizing the relative approximation error $\mathbb{E}[\frac{\|\tilde{\boldsymbol{A}}^{\top}\nabla \boldsymbol{H}^{(l+1)} - \mathtt{approx}(\tilde{\boldsymbol{A}}^{\top}\nabla \boldsymbol{H}^{(l+1)})\|_F}{\|\tilde{\boldsymbol{A}}\|_F \|\nabla \boldsymbol{H}^{(l+1)}\|_F}]$ summarized over all layers (Adelman et al., 2021). Also, different sparse operations are weighted summation by the magnitude of gradient $\|\nabla \boldsymbol{H}^{(l+1)}\|_2$, which implicitly encodes the importance of different operations.

Equation (4b) is the constraint that controls the overall FLOPs. Specifically, **the FLOPs of `SpMM`** between $\tilde{\boldsymbol{A}}$ and the gradient $\nabla \boldsymbol{H} \in \mathbb{R}^{N \times d}$ is $\mathcal{O}(|\mathcal{E}|d)$ and $\sum_{j \in \mathcal{V}} \#nnz_j = |\mathcal{E}|$. We note that Equation (4b) also bounds the number of memory access of `SpMM` .

#### 3.2.1. Greedy Solution

The above combination optimization objective is NP-hard, albeit it can be solved by dynamic programming. However, dynamic programming is very slow, which somehow contradicts our purpose of being efficient. Thus, *we propose to use a greedy algorithm to solve it.* Specifically, it starts with the highest $k_l = |\mathcal{V}|$ for all layers. In each move, it chooses a $k_l$ among $\{k_l\}_{l=1}^{L}$ to reduce by a step size (e.g., $0.02|\mathcal{V}|$), such that the increment of errors in Equation (4a) is minimal. The greedy algorithm will stop when the current total FLOPs fits in the budget in Equation (4b). This algorithm runs super fast, and we found that it has minimal impact on efficiency. We provide the pseudo-code of our greedy algorithm in Algorithm 1 of Appendix B.

### 3.3. When to Apply the Approximation

#### 3.3.1. Cache the Sampled Sparse Matrices

We first give the details about the Compressed Sparse Row (CSR) format for representing the sparse matrix here. CSR stores nonzero values in a matrix and their position in three arrays: index array `Rowptr`, column array `Col`, and value array `Val`. The elements in `Rowptr` act as the starting indices of the elements in `Col` and `Val` that correspond to each row. Specifically, the elements of row $i$ are stored in indices `Rowptr`[i] to `Rowptr[i+ 1]` $-1$ of `Col` and `Val` . The elements in `Col` and `Val` are the column index and value in that column, respectively. Figure 5 shows the CSR format of the matrix shown in Figure 3. We ignore the `Val` array here for illustration convenience.

Executing the top-$k$ sampling contains two steps: First, it decides the indices corresponding to the top-$k$ largest column row norms in Equation (3). Second, slicing the matrices according to the indices. In practice, the overhead of the first step can be ignored. However, unlike dense matrices, slicing the adjacency matrix is much slower due to its irregular data format. To see this, suppose the top-$k$ indices of the sparse matrix in Figure 3 correspond to the orange column-row pairs. Figure 5 shows the process of slicing the adjacency matrix in CSR format by reserving only the orange columns. Slicing sparse matrices requires to re-process the graph to build the new `Rowptr` and `Col` (Fey & Lenssen, 2019), which introduces significant time overhead, especially for large graphs.

For the full graph training, we use the same adjacency matrix across different epochs[1]. We made a crucial observation that the top-$k$ indices in the adjacency matrix tend to be the same across iterations. In Figure 4, we plot the AUC score of top-$k$ indices between every iteration $t$ and iteration $t + 10$ for

[1]For sub-graph based training, we can first sample all of the sub-graphs offline. Then during the training, we apply the caching mechanism to each sampled graph.

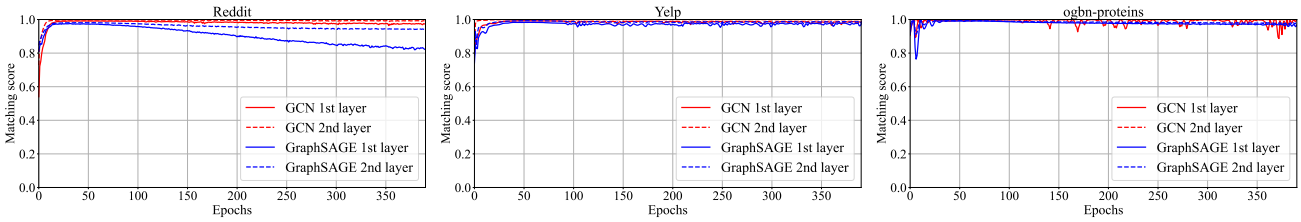


*Figure 4:* For each layer, the selected column-row pairs tend to be very similar across iterations. Models here are two-layer GCN and GraphSAGE. Here we show the matching scores (AUC) of top-$k$ indices between every 10 steps.

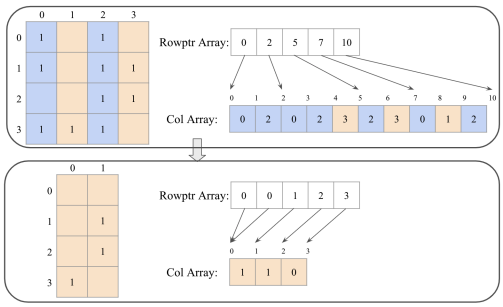


*Figure 5:* The process of slicing the sparse matrix in Figure 3 by only reserving orange columns (in CSR format).

each layer throughout the whole training process. Here we note that AUC score is a commonly used ranking measure and a 1.0 AUC score means the ranking of column-row pairs is identical across iterations. The results in Figure 4 indicate that the top-$k$ indices won't change significantly within a few iterations. Thus, as shown in Figure 2, we propose to reuse the sampled adjacency matrix for each layer across nearby iterations.

**Discussion.** The rationale behind the success of caching is the slow rate of change in the learned embeddings within GNNs (Fey et al., 2021; Wan et al., 2022a). Prior research has leveraged this "staleness" of embeddings to enhance the efficiency of GNN training [1, 2]. The success of caching can also be explained by the staleness: if embeddings (and their gradients) across consecutive steps remain nearly identical, the sampled sparse matrix will also exhibit minimal variation. Later we experimentally show that the caching mechanism does not impact the model performance a lot, but leads to a significant speedup.

#### 3.3.2. Switch back at the end

When training neural networks, the common practice is to use a large learning rate for exploration and anneal to a small one for final convergence (Li et al., 2019). The rationale behind this strategy is that, at the end of the training process, we need to fine-tune our model with small noise for convergence. Since our approximation sparse operations will bring extra noise to the gradient, intuitively, we can switch back to the original sparse operations to help convergence. More formally, we propose to use approximated sparse operation during most of the training process, while switching back to the original sparse operation at the final stage. We experimentally show that this switching mechanism significantly reduces the accuracy drop at the cost of slightly less acceleration effect.

**We note that the switching mechanism is not proposed in this paper.** The switching mechanism takes inspiration from previous work Dao et al. (2022), and both our work and Dao et al. (2022) utilize the switching mechanism to minimize the impact of approximation.

## 4. Related work and Discussion

Due to the page limit, we first discuss the related work on approximated matrix multiplication. Other related topics, i.e., subgraph-based training, randomized GNN training, and non-approximated GNN acceleration, can be found in Appendix C.

***Approximated Matrix Multiplication.*** The approximated matrix production can be roughly divided into three categories. However, only a few of them can be used for accelerating GNN training. Specifically, (1) Random walk-based methods (Cohen & Lewis, 1999) performs random walks on a graph representation of the dense matrices, but is only applicable to non-negative matrices; (2) Butterfly-based methods (Chen et al., 2021; Dao et al., 2022) replace dense matrices with butterfly matrices. It is not applicable to `SpMM` in GNNs because the adjacency matrix often cannot be reduced to a butterfly matrix. (3) Column-row sampling methods(Drineas et al., 2006a; Drineas & Kannan, 2001) sample the input matrices with important rows and columns, then perform the production on the sampled matrix as usual.

## 5. Limitations

First, to guarantee the model accuracy, we only replace the sparse operation in the backward pass. Thus the upper bound of `RSC` 's speedup is limited. However, we note that the backward pass usually is more time-consuming than the forward pass, which is also empirically shown in Table 2. Second, some GNNs rely on the scatter-and-gather instead of `SpMM` (and its variant) to perform the aggregation, such as GAT (Veličković et al., 2017). They are not covered in this paper. However, scatter-and-gather based GNNs can also

be accelerated by RSC because the column-row sampling is also applicable to scatter and gather operation. Similarly, the caching and switching mechanisms are also applicable to them. However, for the resource allocation Algorithm 1, the scatter and gather operations require tailored error bound and the computation cost modeling in Equation (4). We leave it as future work.

# 6. Experiments

We verify the effectiveness of our proposed framework via answering the following research questions: **Q1**: How effective is RSC in terms of accuracy with reduced training time? **Q2**: How effective is our proposed allocation strategy compared to the uniform allocation strategy? **Q3**: What is the layer-wise ratio assigned by RSC ? **Q4**: How effective is the caching and switching mechanism in terms of the trade-off between efficiency and accuracy? If without explicitly mentioned, all reported results are averaged over ten random trials

## 6.1. Experimental Settings

**Datasets and Baselines.** To evaluate RSC , we adopt four common large-scale graph benchmarks from different domains, i.e., Reddit (Hamilton et al., 2017), Yelp (Zeng et al., 2020), *ogbn-proteins* (Hu et al., 2020), and *ogbn-products* (Hu et al., 2020). We evaluate RSC under both the mini-batch training and full-batch training settings. For the mini-batch training setting, we integrate RSC with one of the state-of-the-art sampling methods, GraphSAINT (Zeng et al., 2020). For the full-batch training setting, we integrate RSC with three popular models: two commonly used shallow models, namely, GCN (Kipf & Welling, 2017) and GraphSAGE (Hamilton et al., 2017), and one deep model GCNII (Chen et al., 2020). To avoid creating confusion, GCN, GraphSAGE, and GCNII are all trained with the whole graph at each step. For a fair comparison, we use the MEAN aggregator for GraphSAGE and GraphSAINT throughout the paper. Details about the hyperparameters and datasets are in Appendix D.

**Hyperparameter settings.** RSC contains three parts. First, the allocation strategy. We choose the overall budget $C$ in Equation (4b) from $\{0.1, 0.3, 0.5\}$. We run the resource allocation strategy every **ten** steps. The step size $\alpha$ in Algorithm 1 is set as $0.02|\mathcal{V}|$. Second, the caching mechanism. According to Figure 4, we sample the adjacency matrix every **ten** steps and reuse the sampled matrices for nearby steps. Third, the switching mechanism, where we apply RSC for **80%** of the total epochs, while switching back to the original operations for the rest of the **20%** epochs. Due to the page limit, We present a detailed hyperparameter study in Appendix E Figure 11 and Figure 12.

**Evaluation metrics.** To evaluate the practical usage of RSC , we report the wall clock time speedup measured on GPUs. Specifically, the speedup equals $T_{\text{baseline}}/T_{rsc}$, where $T_{\text{baseline}}$ and $T_{rsc}$ are the wall clock training time of baseline and RSC , respectively. We note that *the $T_{rsc}$ includes the running time of the greedy algorithm, and the effects of caching and switching.*

## 6.2. Performance Analysis

### 6.2.1. Accuracy-efficiency trade-off

To answer **Q1**, we summarize the speedup and the test accuracy/F1-micro/AUC of different methods in Table 3. Since RSC accelerates the sparse operation in the backward pass, we also provide the detailed efficiency analysis in Table 2. In summary, we observe:

❶ *At the operation level, RSC can accelerate the sparse operation in the backward pass by up to* $11.6\times$. *For end-to-end training, the accuracy drop of applying RSC over baselines is negligible (0.3%) across different models and datasets, while achieving up to* $1.6\times$ *end-to-end wall clock time speedup.* The gap between the operation speedup and the end-to-end speedup is due to the following two reasons. *First*, we focus on accelerating the sparse computations in GNNs, which is the unique bottleneck to GNNs. The other dense computations can certainly be accelerated by approximation methods, but this is beyond the scope of this paper. *Second*, we only accelerate the sparse computation in the backward pass instead of the forward one to guarantee performance. We note that for approximation methods that accelerate the training process at operation level, a $1.2 \approx 1.3\times$ wall-clock speedup with negligible accuracy drop can be regarded as non-trivial (for details, please see Table 1 in (Adelman et al., 2021)), especially considering that these approximation methods are orthogonal to most of the existing efficient training methods. For GraphSAINT, the speedup of RSC is around $1.1\times$, which is smaller than the full graph training. This is because for subgraph-based training, the equivalent "batch size" is much smaller than the full graph counterparts. As a result, the GPU utility is low since it does not assign each processor a sufficient amount of work and the bottleneck is the mini-batch transfer time (Kaler et al., 2022). We note that the mini-batch sampling and transfer time can be optimized from the system perspective (Kaler et al., 2022), which is orthogonal to our work. The speedup is expected to be larger when the mini-batch sampling time is optimized.

### 6.2.2. Ablation on resource allocation.

Due to the page limit, we first show the running time of the greedy algorithm in Appendix E Table 11. We conclude that the overhead of the greedy algorithm is negligible compared to the acceleration effect of RSC . To answer **Q2**, we

*Table 2:* Comparison on the efficiency at the operation level. fwd/bwd is the wall-clock time for a single forward/backward pass (ms). `SpMM_MEAN` corresponds to the MEAN aggregator used in GraphSAGE (Appendix A.3).

| | | Reddit | | Yelp | | *ogbn-proteins* | | *ogbn-products* | |
|---|---|---|---|---|---|---|---|---|---|
| | | fwd | bwd | fwd | bwd | fwd | bwd | fwd | bwd |
| `SpMM` | Baseline | 36.28 | 44.23 | 26.88 | 34.38 | 31.72 | 42.99 | 261.03 | 316.80 |
| | +`RSC` | - | 3.81 (11.6×) | - | 9.86 (3.49×) | - | 14.87 (2.89×) | - | 35.28 (8.98×) |
| `SpMM_MEAN` | Baseline | 36.21 | 44.27 | 26.78 | 34.38 | 31.80 | 43.11 | 261.03 | 316.84 |
| (Appendix A.3) | +`RSC` | - | 7.47 (5.92×) | - | 19.62 (1.75×) | - | 5.22 (8.26×) | - | 71.59 (4.43×) |

*Table 3:* Comparison on the test accuracy/F1-micro/AUC and speedup on four datasets. **Bold faces** indicate the accuracy drop is negligible ($\approx 0.3\%$) or the result is better compared to the baseline.The hardware here is a RTX3090 (24GB).

| # nodes | | 230K | | | 717K | | | 132K | | | 2.4M | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # edges | | 11.6M | | | 7.9M | | | 39.5M | | | 61.9M | | |
| Model | Methods | Reddit | | | Yelp | | | *ogbn-proteins* | | | *ogbn-products* | | |
| | | Acc. | Budget $C$ | Speedup | F1-micro | Budget $C$ | Speedup | AUC | Budget $C$ | Speedup | Acc. | Budget $C$ | Speedup |
| Graph-SAINT | Baseline | 96.40±0.03 | 1 | 1× | 63.30±0.14 | 1 | 1× | — | — | — | 79.01±0.21 | 1 | 1× |
| | +`RSC` | **96.24±0.03** | 0.1 | 1.11× | **63.34±0.18** | 0.1 | 1.09× | — | — | — | **78.99±0.32** | 0.3 | 1.04× |
| GCN | Baseline | 95.33±0.03 | 1 | 1× | 44.28±1.04 | 1 | 1× | 71.99±0.66 | 1 | 1× | 75.74±0.11 | 1 | 1× |
| | +`RSC` | **95.13±0.05** | 0.1 | 1.47× | **46.09±0.54** | 0.1 | 1.17× | **71.60±0.45** | 0.3 | 1.51× | **75.44±0.21** | 0.3 | 1.35× |
| GraphSAGE (full batch) | Baseline | 96.61±0.05 | 1 | 1× | 63.06±0.18 | 1 | 1× | 76.09±0.77 | 1 | 1× | 78.73 ± 0.12 | 1 | 1× |
| | +`RSC` | **96.52±0.04** | 0.1 | 1.32× | **62.89±0.19** | 0.1 | 1.13× | **76.30±0.42** | 0.3 | 1.60× | **78.50± 0.09** | 0.1 | 1.53× |
| GCNII | Baseline | 96.71±0.07 | 1 | 1× | 63.45±0.17 | 1 | 1× | 73.79±1.32 | 1 | 1× | — | — | — |
| | +`RSC` | **96.50±0.12** | 0.3 | 1.45× | **63.57±0.21** | 0.1 | 1.19× | **75.20±0.54** | 0.5 | 1.41× | — | — | — |

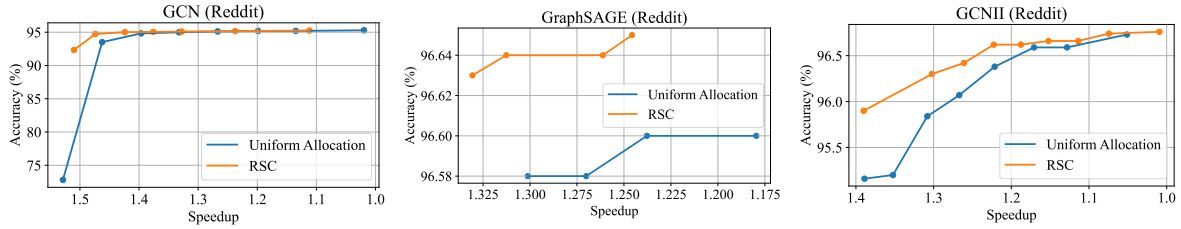


*Figure 6:* The Pareto frontier of the accuracy-efficiency trade-off for `RSC` and the uniform allocation. Here we disabled the caching and switch mechanism for a fair comparison. More results can be found in Appendix E

*Table 4:* Ablation on the caching and switching mechanism. Experiments are conducted on *ogbn-proteins*. All results are averaged over five random trials.

| Ablation on | Caching | Switching | AUC | Speedup |
|---|---|---|---|---|
| GCN | ✗ | ✗ | 71.60 ± 0.66 | 1.19× |
| | ✗ | ✓ | 72.19 ± 0.79 | 1.14× |
| | ✓ | ✗ | 69.80 ± 0.60 | 1.60× |
| | ✓ | ✓ | 71.60 ± 0.45 | 1.51× |
| GraphSAGE | ✗ | ✗ | 75.23 ± 0.79 | 1.37× |
| | ✗ | ✓ | 76.39 ± 0.39 | 1.32× |
| | ✓ | ✗ | 75.53 ± 0.60 | 1.78× |
| | ✓ | ✓ | 76.30 ± 0.42 | 1.60× |
| GCNII | ✗ | ✗ | 74.07 ± 0.83 | 1.10× |
| | ✗ | ✓ | 74.50 ± 0.52 | 1.04× |
| | ✓ | ✗ | 72.47 ± 0.75 | 1.46× |
| | ✓ | ✓ | 75.20 ± 0.54 | 1.41× |

compare `RSC` with the uniform allocation strategy, i.e., setting $k_l = C|\mathcal{V}|$ for all sparse operations in the backward pass. As shown in Figure 6, we plot the Pareto frontier of the accuracy-efficiency trade-off on the Reddit dataset for `RSC` and the uniform strategy with different $C$. For a fair comparison, we disabled the caching and switching mechanism. Due to page limit, more results are shown in Appendix E. We observe that: ❷ *`RSC` exhibits a superior trade-off between accuracy and efficiency compared to the uniform allocation, especially under high speedup regime.* Namely, compared to the uniform allocation, `RSC` can achieve higher model accuracy under the same speedup. This can be explained by the fact that each operation has a different importance to the model performance. `RSC` can automatically allocate more resources to important operations under a given total budget.

To answer **Q3**, due to the page limit, we visualize the allocated $k_l$ for each layer across iterations in Appendix E Figure 7, and the degree of picked nodes in Appendix E Figure 8. We observe: ❸ *The $k_l$ assigned by `RSC` evolves along with the training.*

#### 6.2.3. Ablation on Caching and Switching.

In section 6.2.2, we have shown the superior results of the proposed resource allocation strategy. As we mentioned in Section 3.3, we also introduce two simple tricks to for improving `RSC` , i.e., the caching and switching mechanism. To verify the effect of each of them (**Q4**), we conduct incremental evaluations on GCN, GraphSAGE and GCNII with

*ogbn-proteins*, which are summarized in Table 4. The row without caching and switching in Table 4 corresponds to the results with the proposed resource allocation strategy. We observe: ❹ *Switching mechanism significantly improves the model performance, at the cost of slightly less acceleration effect.* As we analyzed in Section 3.3.2, the improvement can be explained by the fact that the final training stage requires smaller gradient noise to help convergence. ❺ *Caching mechanism significantly improves the wall-clock time speedup, at the cost of worse model performance.* Although caching mechanism can reduce the overhead of sampling, the performance drop is too large ($> 1\%$). *Intuitively, the accuracy drop of caching also implies that we could not use a "static" down-sampled graph throughout the training process.* ❻ *Surprisingly, jointly applying the caching and switching, the performance drop can be minimized.*

## 7. Acknowledgements

The authors thank the anonymous reviewers for their helpful comments. The work is in part supported by NSF grants NSF IIS-2224843. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing any funding agencies.

## 8. Conclusions and Future work

We propose `RSC` , which replaces the sparse computations in GNNs with their fast approximated versions. `RSC` can be plugged into most of the existing training frameworks to improve their efficiency. Future work includes exploring `RSC` for GNNs that rely on scatter-and-gather operations.

## References

Adelman, M., Levy, K., Hakimi, I., and Silberstein, M. Faster neural network training with approximate tensor operations. *Advances in Neural Information Processing Systems*, 34:27877–27889, 2021.

Cai, T., Luo, S., Xu, K., He, D., Liu, T.-y., and Wang, L. Graphnorm: A principled approach to accelerating graph neural network training. In *International Conference on Machine Learning*, pp. 1204–1215. PMLR, 2021.

Chen, B., Dao, T., Liang, K., Yang, J., Song, Z., Rudra, A., and Re, C. Pixelated butterfly: Simple and efficient sparse training for neural network models. *arXiv preprint arXiv:2112.00029*, 2021.

Chen, J., Zhu, J., and Song, L. Stochastic training of graph convolutional networks with variance reduction. In *International conference on machine learning*. PMLR, 2017.

Chen, M., Wei, Z., Huang, Z., Ding, B., and Li, Y. Simple and deep graph convolutional networks. In *International Conference on Machine Learning*, pp. 1725–1735. PMLR, 2020.

Chiang, W.-L., Liu, X., Si, S., Li, Y., Bengio, S., and Hsieh, C.-J. 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*, pp. 257–266, 2019.

Cohen, E. and Lewis, D. D. Approximating matrix multiplication for pattern recognition tasks. *Journal of Algorithms*, 30(2):211–252, 1999.

Cong, W., Forsati, R., Kandemir, M., and Mahdavi, M. Minimal variance sampling with provable guarantees for fast training of graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1393–1403, 2020.

Dao, T., Chen, B., Sohoni, N. S., Desai, A., Poli, M., Grogan, J., Liu, A., Rao, A., Rudra, A., and Ré, C. Monarch: Expressive structured matrices for efficient and accurate training. In *International Conference on Machine Learning*, pp. 4690–4721. PMLR, 2022.

Drineas, P. and Kannan, R. Fast monte-carlo algorithms for approximate matrix multiplication. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pp. 452–459. IEEE, 2001.

Drineas, P., Kannan, R., and Mahoney, M. W. Fast monte carlo algorithms for matrices i: Approximating matrix multiplication. *SIAM Journal on Computing*, 36(1):132–157, 2006a.

Drineas, P., Kannan, R., and Mahoney, M. W. Fast monte carlo algorithms for matrices i: Approximating matrix multiplication. *SIAM Journal on Computing*, 36(1):132–157, 2006b.

Duan, K., Liu, Z., Wang, P., Zheng, W., Zhou, K., Chen, T., Hu, X., and Wang, Z. A comprehensive study on large-scale graph training: Benchmarking and rethinking. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022a. URL https://openreview.net/forum?id=2QrFr_U782Z.

Duan, K., Liu, Z., Wang, P., Zheng, W., Zhou, K., Chen, T., Hu, X., and Wang, Z. A comprehensive study on large-scale graph training: Benchmarking and rethinking. 2022b.

Feng, W., Zhang, J., Dong, Y., Han, Y., Luan, H., Xu, Q., Yang, Q., Kharlamov, E., and Tang, J. Graph random neural networks for semi-supervised learning on graphs.

*Advances in neural information processing systems*, 33: 22092–22103, 2020.

Feng, W., Dong, Y., Huang, T., Yin, Z., Cheng, X., Kharlamov, E., and Tang, J. Grand+: Scalable graph random neural networks. In *Proceedings of the ACM Web Conference 2022*, pp. 3248–3258, 2022.

Fey, M. and Lenssen, J. E. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

Fey, M., Lenssen, J. E., Weichert, F., and Leskovec, J. Gnnautoscale: Scalable and expressive graph neural networks via historical embeddings. In *International conference on machine learning*, 2021.

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256. JMLR Workshop and Conference Proceedings, 2010.

Hamilton, W. L., Ying, R., and Leskovec, J. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 1025–1035, 2017.

Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J. Eie: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

Han, X., Jiang, Z., Liu, N., and Hu, X. G-mixup: Graph data augmentation for graph classification. *arXiv preprint arXiv:2202.07179*, 2022.

Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

Huang, G., Dai, G., Wang, Y., and Yang, H. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12. IEEE, 2020a.

Huang, Q., He, H., Singh, A., Lim, S.-N., and Benson, A. R. Combining label propagation and simple models out-performs graph neural networks. In *International Conference on Learning Representations*, 2020b.

Huang, W., Zhang, T., Rong, Y., and Huang, J. Adaptive sampling towards fast graph representation learning. In *Advances in Neural Information Processing Systems*, 2018.

Jiang, Z., Han, X., Fan, C., Liu, Z., Zou, N., Mostafavi, A., and Hu, X. Fmp: Toward fair graph message passing against topology bias. *arXiv preprint arXiv:2202.04187*, 2022.

Jin, W., Ma, Y., Liu, X., Tang, X., Wang, S., and Tang, J. Graph structure learning for robust graph neural networks. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 66–74, 2020.

Kaler, T., Stathas, N., Ouyang, A., Iliopoulos, A.-S., Schardl, T., Leiserson, C. E., and Chen, J. Accelerating training and inference of graph neural networks with fast sampling and pipelining. *Proceedings of Machine Learning and Systems*, 4:172–189, 2022.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=SJU4ayYgl.

Klicpera, J., Bojchevski, A., and Günnemann, S. Predict then propagate: Graph neural networks meet personalized pagerank. In *International Conference on Learning Representations*, 2018.

Li, Y., Wei, C., and Ma, T. Towards explaining the regularization effect of initial large learning rate in training neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

Liu, Z., Jin, H., Wang, T.-H., Zhou, K., and Hu, X. Divaug: Plug-in automated data augmentation with explicit diversity maximization. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 4762–4770, 2021.

Martinsson, P.-G. and Tropp, J. Randomized numerical linear algebra: foundations & algorithms (2020). *arXiv preprint arXiv:2002.01387*, 2020.

Md, V., Misra, S., Ma, G., Mohanty, R., Georganas, E., Heinecke, A., Kalamkar, D., Ahmed, N. K., and Avancha, S. Distgnn: Scalable distributed training for large-scale graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2021.

Narayanan, S. D., Sinha, A., Jain, P., Kar, P., and SELLAMANICKAM, S. Iglu: Efficient GCN training via lazy updates. In *International Conference on Learning*

*Representations*, 2022. URL https://openreview.net/forum?id=5kq11Tl1z4.

Qiu, J., Dhulipala, L., Tang, J., Peng, R., and Wang, C. Lightne: A lightweight graph processing system for network embedding. In *Proceedings of the 2021 international conference on management of data*, pp. 2281–2289, 2021.

Rahman, M. K., Sujon, M. H., and Azad, A. Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 256–266. IEEE, 2021.

Ramezani, M., Cong, W., Mahdavi, M., Kandemir, M., and Sivasubramaniam, A. Learn locally, correct globally: A distributed algorithm for training graph neural networks. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=FndDxSz3LxQ.

Rong, Y., Huang, W., Xu, T., and Huang, J. Dropedge: Towards deep graph convolutional networks on node classification. *arXiv preprint arXiv:1907.10903*, 2019.

Savas, B. and Dhillon, I. S. Clustered low rank approximation of graphs in information science applications. In *Proceedings of the 2011 SIAM International Conference on Data Mining*, pp. 164–175. SIAM, 2011.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. Graph attention networks. In *International Conference on Learning Representations*, 2017.

Wan, C., Li, Y., Kim, N. S., and Lin, Y. {BDS}-{gcn}: Efficient full-graph training of graph convolutional nets with partition-parallelism and boundary sampling, 2021. URL https://openreview.net/forum?id=uFA24r7v4wL.

Wan, C., Li, Y., Li, A., Kim, N. S., and Lin, Y. 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, 2022a.

Wan, C., Li, Y., Wolfe, C. R., Kyrillidis, A., Kim, N. S., and Lin, Y. Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication. *arXiv preprint arXiv:2203.10428*, 2022b.

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., and Zhang, Z. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

Wang, Y., Feng, B., and Ding, Y. Tc-gnn: Accelerating sparse graph neural network computation via dense tensor core on gpus. *arXiv preprint arXiv:2112.02052*, 2021.

Wang, Z., Wu, X. C., Xu, Z., and Ng, T. E. Cupcake: Acompression optimizer for scalable communication-efficient distributed training.

Wang, Z., Xu, Z., Wu, X., Shrivastava, A., and Ng, T. E. Dragonn: Distributed randomized approximate gradients of neural networks. In *International Conference on Machine Learning*, pp. 23274–23291. PMLR, 2022.

Wu, F., Souza, A., Zhang, T., Fifty, C., Yu, T., and Weinberger, K. Simplifying graph convolutional networks. In *International conference on machine learning*, pp. 6861–6871. PMLR, 2019.

Xu, K., Zhang, M., Jegelka, S., and Kawaguchi, K. Optimization of graph neural networks: Implicit acceleration by skip connections and more depth. In *International Conference on Machine Learning*, pp. 11592–11602. PMLR, 2021.

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., and Leskovec, J. 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.

Yu, L., Shen, J., Li, J., and Lerer, A. Scalable graph neural networks for heterogeneous graphs. *arXiv preprint arXiv:2011.09679*, 2020.

Yuan, B., Wolfe, C. R., Dun, C., Tang, Y., Kyrillidis, A., and Jermaine, C. Distributed learning of fully connected neural networks using independent subnet training. *Proc. VLDB Endow.*, 15(8):1581–1590, 2022. URL https://www.vldb.org/pvldb/vol15/p1581-wolfe.pdf.

Zeng, H., Zhou, H., Srivastava, A., Kannan, R., and Prasanna, V. Graphsaint: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=BJe8pkHFwS.

Zha, D., Feng, L., Tan, Q., Liu, Z., Lai, K.-H., Bhushanam, B., Tian, Y., Kejariwal, A., and Hu, X. Dreamshard: Generalizable embedding table placement for recommender systems. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=_atSgd9Np52.

Zha, D., Feng, L., Luo, L., Bhushanam, B., Liu, Z., Hu, Y., Nie, J., Huang, Y., Tian, Y., Kejariwal, A., and Hu, X. Pre-train and search: Efficient embedding table sharding with pre-trained neural cost models. *CoRR*, abs/2305.01868, 2023. doi: 10.48550/arXiv.2305.01868. URL https://doi.org/10.48550/arXiv.2305.01868.

Zhang, H., Yu, Z., Dai, G., Huang, G., Ding, Y., Xie, Y., and Wang, Y. Understanding gnn computational graph: A coordinated computation, io, and memory perspective. *Proceedings of Machine Learning and Systems*, 4:467–484, 2022.

Zheng, D., Ma, C., Wang, M., Zhou, J., Su, Q., Song, X., Gan, Q., Zhang, Z., and Karypis, G. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pp. 36–44. IEEE, 2020.

Zhong, S., Zhang, G., Huang, N., and Xu, S. Revisit kernel pruning with lottery regulated grouped convolutions. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=LdEhiMG9WLO.

Zhou, K., Liu, Z., Chen, R., Li, L., Choi, S., and Hu, X. Table2graph: Transforming tabular data to unified weighted graph. In Raedt, L. D. (ed.), *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pp. 2420–2426. ijcai.org, 2022. doi: 10.24963/ijcai.2022/336. URL https://doi.org/10.24963/ijcai.2022/336.

Zhou, K., Choi, S.-H., Liu, Z., Liu, N., Yang, F., Chen, R., Li, L., and Hu, X. Adaptive label smoothing to regularize large-scale graph training. In *Proceedings of the 2023 SIAM International Conference on Data Mining (SDM)*, pp. 55–63. SIAM, 2023.

Zou, D., Hu, Z., Wang, Y., Jiang, S., Sun, Y., and Gu, Q. Layer-dependent importance sampling for training deep and large graph convolutional networks. *arXiv preprint arXiv:1911.07323*, 2019.

# A. Mathematical Analysis

## A.1. Why Sampling-based Approximation for GNN?

In the main text, we mentioned `SpMM` is the main speed bottleneck for GNNs. Below we illustrate why the column-row sampling is suitable for accelerating `SpMM` in GNNs, from the approximation error perspective. Here we analyze $\tilde{\boldsymbol{A}}\boldsymbol{J}^{(l)} = \texttt{SpMM}(\tilde{\boldsymbol{A}}, \boldsymbol{J}^{(l)})$ for illustration convenience. For the backward pass of `SpMM` , the analysis is similar, except that we are approximating $\nabla \boldsymbol{J}^{(l)} = \texttt{SpMM}(\tilde{\boldsymbol{A}}^\top, \nabla \boldsymbol{H}^{(l+1)})$.

Column-row sampling approximates the matrix production by excluding some "unimportant" columns and rows in the original matrix. So intuitively, the approximation error $\mathbb{E}[||\tilde{\boldsymbol{A}}\boldsymbol{J}^{(l)} - \texttt{approx}(\tilde{\boldsymbol{A}}\boldsymbol{J}^{(l)})||_F]$ is low if the "unimportant" columns/rows are correlated in the selected one. Namely, $\tilde{\boldsymbol{A}}$ and $\boldsymbol{J}^{(l)}$ are low-rank. Formally, we have the following theorem:

**Theorem A.1** ((Martinsson & Tropp, 2020))**.** *Suppose we approximate $\tilde{\boldsymbol{A}}\boldsymbol{J}^{(l)}$ using column-row sampling, and $p_i$ is obtained by Equation (3). Then for any positive number $\epsilon$, if the number of samples $k$ satisfies $k \geq \epsilon^{-2}(\texttt{srank}(\tilde{\boldsymbol{A}}) + \texttt{srank}(\boldsymbol{J}^{(l)}))\log(|\mathcal{V}| + d)$, we have $\mathbb{E}[||\tilde{\boldsymbol{A}}\boldsymbol{J}^{(l)} - \texttt{approx}(\tilde{\boldsymbol{A}}\boldsymbol{J}^{(l)})||_F] \leq 2\epsilon$,*

where `srank` in Theorem A.1 is called the stable rank, which is the continuous surrogate measure for the rank that is largely unaffected by tiny singular values. Formally for any matrix $\boldsymbol{Y}$, $\texttt{srank}(\boldsymbol{Y}) = \frac{||\boldsymbol{Y}||_F^2}{||\boldsymbol{Y}||_2} \leq \texttt{rank}(\boldsymbol{Y})$.

Fortunately, most real-world graphs are cluster-structured, which means the adjacency matrix $\tilde{\boldsymbol{A}}$ is low-rank (Qiu et al., 2021; Savas & Dhillon, 2011). The low-rank property of real-world graphs is also wildly reported in previous work (Jin et al., 2020; Qiu et al., 2021). Moreover, the intermediate activations $\boldsymbol{J}^{(l)}$ and the activation gradients are also low-rank, due to the aggregation. Namely, low-rank means "correlation" in the row/column space. The embedding (i.e., rows in the activation matrix) of connected nodes tend to close due to the graph propagation, which resulting in the low-rank property of the activation matrix. Thus for GNNs, the approximation error is low with a relatively small number of sample $k$. This perspective is also experimentally verified in the experiment section.

## A.2. Proof of Proposition 1

**Proposition A.2** (Proof in Appendix A)**.** *If the approximation method is itself unbiased, and we only replace the `SpMM` in the backward pass with its approximated version, while leaving the forward one unchanged, then the calculated gradient is provably unbiased.*

Here we note that in the main text, for the notation convenience, we ignore the backward pass of ReLU. However, the proof here will consider the non-linear activation function to prove the unbiasedness. Let $\boldsymbol{H}_{pre}^{(l+1)} = \texttt{SpMM}(\tilde{\boldsymbol{A}}, \boldsymbol{J}^{(l)})$ be the pre-activation. The backward pass of ReLU is:

$$\begin{aligned}\mathbb{E}[\nabla \boldsymbol{H}_{pre}^{(l+1)}] &= \mathbb{E}[\mathbb{1}_{\boldsymbol{H}_{pre}^{(l+1)}>0} \odot \nabla \boldsymbol{H}^{(l+1)}] \\ &= \mathbb{1}_{\boldsymbol{H}_{pre}^{(l+1)}>0} \odot \mathbb{E}[\nabla \boldsymbol{H}^{(l+1)}],\end{aligned} \tag{5}$$

where $\odot$ is the element-wise product and $\mathbb{1}$ is the indicator function. The element-wise product is linear operation and $\mathbb{1}_{\boldsymbol{H}_{pre}^{(l+1)}>0}$ is only related to the pre-activation in the forward pass, we only apply the approximation during the backward pass so $\mathbb{1}_{\boldsymbol{H}_{pre}^{(l+1)}>0}$ can be extracted from the expectation. We know that for the last layer, we have $\mathbb{E}[\nabla \boldsymbol{H}^{(L)}] = \boldsymbol{H}^{(L)}$ since we do not apply ReLU at the output layer. We then can prove by induction that $\mathbb{E}[\nabla \boldsymbol{H}^{(l+1)}] = \boldsymbol{H}^{(l+1)}$ and $\mathbb{E}[\nabla \boldsymbol{J}^{(l)}] = \mathbb{E}[\texttt{approx}(\tilde{\boldsymbol{A}}^\top \nabla \boldsymbol{H}_{pre}^{(l+1)})] = \nabla \boldsymbol{J}^{(l)}$ for any layer $l$.

## A.3. Analysis of MEAN aggregator

For GraphSAGE, one commonly used aggregator is the MEAN aggregator, which can be expressed as follows:

$$\boldsymbol{H}^{(l+1)} = \boldsymbol{W}_1 \boldsymbol{H}^{(l)} + \boldsymbol{W}_2 \texttt{SpMM_MEAN}(\boldsymbol{A}, \boldsymbol{H}^{(l)}), \tag{6}$$

where `SpMM_MEAN` is one variant of the vanilla `SpMM`, which replace the reducer function from `sum(·)` to `mean(·)`. We note that in popular GNN packages, the MEAN aggregator usually is implemented based on `SpMM_MEAN` (Fey & Lenssen, 2019; Wang et al., 2019) to reduce the memory usage. Here we give an example of `SpMM_MEAN` to illustrate how it works:

$$\texttt{SpMM_MEAN}(\begin{bmatrix}1 & 0\\ 0 & 4\\ 5 & 6\end{bmatrix}, \begin{bmatrix}7 & 8\\ 9 & 10\end{bmatrix}) = \begin{bmatrix}\frac{1}{2}(1\times 7+0\times 9) & \frac{1}{2}(1\times 8+0\times 10)\\ \frac{1}{2}(0\times 7+4\times 9) & \frac{1}{2}(0\times 8+4\times 10)\\ \frac{1}{2}(5\times 7+6\times 9) & \frac{1}{2}(5\times 8+6\times 10)\end{bmatrix},$$

Equivalently, the `SpMM_MEAN` can also be expressed as:

$$\texttt{SpMM_MEAN}(\boldsymbol{A}, \boldsymbol{H}^{(l)}) = \boldsymbol{D}^{-1}\boldsymbol{A}\boldsymbol{H}^{(l)},$$

where $\boldsymbol{D}$ is the degree matrix of $\boldsymbol{A}$. Thus, although we did not normalize the adjacency matrix in GraphSAGE, when applying the top-$k$ sampling to approximate `SpMM_MEAN`, the column norm of $\boldsymbol{A}_{:,j_i}$ is actually $\frac{1}{\sqrt{\text{Deg}_{j_i}}}$ due to the normalization.

Also, for GraphSAGE, the inputs to the first `SpMM_MEAN` operation are $\boldsymbol{A}$ and $\boldsymbol{X}$. They do not require gradient since they are not trainable. Thus, the first SAGE layer is not presented in Figure 8 and Figure 7.

## B. Pseudo code of the greedy algorithm

**Algorithm 1** The greedy algorithm

**Inputs:** Gradients of node embeddings $\{\nabla \boldsymbol{H}^{(1)}, \cdots \nabla \boldsymbol{H}^{(L)}\}$, adjacency matrix $\boldsymbol{A}$, graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, hidden dimensions $\{d_1, \cdots d_L\}$.
**Parameters:** The step size $\alpha$, the overall budget $C$.
**Outputs:** The layer-wise $\{k_1, \cdots k_L\}$ associated with the top-$k$ sampling.
$B \leftarrow \sum_{l=1}^{L} |\mathcal{E}| d_l$.
$\forall i, k_l \leftarrow |\mathcal{V}|, \text{Top}_{k_l} \leftarrow \{1, \cdots |\mathcal{V}|\}$.
**while** $B \geq C \sum_{l=1}^{L} |\mathcal{E}| d_l$ **do**
  $m \leftarrow \arg\min_{l\in\{1,\cdots L\}}(\sum_{i\in \text{Top}_{k_l}} \frac{\|\boldsymbol{A}_{:,i}^{\top}\|_2 \|\nabla \boldsymbol{H}_{i,:}^{(l+1)}\|_2}{\|\boldsymbol{A}\|_F \|\nabla \boldsymbol{H}^{(l+1)}\|_F} - \sum_{i\in \text{Top}_{k_l-\alpha|\mathcal{V}|}} \frac{\|\boldsymbol{A}_{:,i}^{\top}\|_2 \|\nabla \boldsymbol{H}_{i,:}^{(l+1)}\|_2)}{\|\boldsymbol{A}\|_F \|\nabla \boldsymbol{H}^{(l+1)}\|_F}$ /⋆ Choose the layer $m$ to reduce by a step size $\alpha|\mathcal{V}|$, such that the increment of errors is minimal. ⋆/
  $B \leftarrow B - d_m \sum_{i\in \text{Top}_{k_m} \cap i \notin \text{Top}_{k_m-\alpha|\mathcal{V}|}} \#nnz_i$ /⋆Since we exclude some column-row pairs for layer $m$, here we reduce the budget $B$ accordingly. ⋆/
  $k_m \leftarrow k_m - \alpha|\mathcal{V}|$ /⋆ Update $k_m$ accordingly. ⋆/
  $\text{Top}_{k_m} \leftarrow$ the set of indices $i$ associated with $k_m$ largest $\frac{\|\boldsymbol{A}_{:,i}^{\top}\|_2 \|\nabla \boldsymbol{H}_{i,:}^{(l+1)}\|_2}{\|\boldsymbol{A}\|_F \|\nabla \boldsymbol{H}^{(l+1)}\|_F}$ /⋆ Update $\text{Top}_{k_m}$ accordingly. ⋆/
**end while**
**Return** $\{k_1, \cdots, k_L\}$

In algorithm 1, here we provide the pseudo code of our greedy algorithm for solving the constrained optimization problem. In Table 11, we show the run time of the greedy algorithm, which is negligible compared to the acceleration effect.

## C. Extended Related works

**Connections to Graph Data Augmentation** Data augmentation (Liu et al., 2021; Han et al., 2022) is wildly adopted in the graph learning for improving model generalization, including dropping nodes (Feng et al., 2020), dropping edges (Rong et al., 2019), and graph mixup (Han et al., 2022). As shown in Figure 5, the top-$k$ sampling drops the entire columns in the adjacency matrix, while keeping the number of rows unchanged. **That means `RSC` drops all of the out edges for a set of nodes.** This can be viewed as the "structural dropedge" for improving the efficiency. Since we only apply the top-$k$ sampling in the backward pass and top-$k$ indices are different for each operation, **`RSC` essentially forward pass with the whole graph, backward pass with different subgraphs at each layer.** This structural dropedge and heterogeneous backward propagation introduce the regularization effect. Thus as shown in the experiment section, `RSC` may also improve the model accuracy over the baseline.

**Subgraph-based GNN training.** The key idea of this line of work is to improve the scalability of GNNs by separating the graph into overlapped small batches, then training models with sampled subgraphs (Hamilton et al., 2017; Huang et al., 2018; Zou et al., 2019; Chiang et al., 2019; Zeng et al., 2020). Based on this idea, various sampling techniques have been proposed, including the node-wise sampling (Hamilton et al., 2017; Chen et al., 2017), layer-wise sampling (Huang et al., 2018; Zou et al., 2019), and subgraph sampling (Chiang et al., 2019; Zeng et al., 2020). However, this approach reduces the memory footprint but results in extra time cost to compute the overlapping nodes between batches. Generally, methods in this category are orthogonal to `RSC` , and they can be combined.

**Graph precomputation.** The graph precomputation methods decouple the message passing from the model training, either as a preprocessing step (Wu et al., 2019; Klicpera et al., 2018; Yu et al., 2020) or post-processing step (Huang et al., 2020b), where the model is simplified as the Multi-Layer Perceptron (MLP). We did consider this line of work in this paper since the backbone model is not GNN anymore.

**Distributed GNN training.** The distributed training leverages extra hardwares to increase the memory capacity and training efficiency (Zha et al., 2023; 2022; Yuan et al., 2022; Wang et al., 2022; Wang et al.). However, the graph data cannot be trivially divided into independent partitions due to the node connectivity. Thus, the graph distributed training frameworks propose to split graph into related partitions and minimize the communication overhead (Wan et al., 2021; 2022b; Ramezani et al., 2022). Our methods are orthogonal to this line of work.

**Other randomized GNN training.** Dropedge (Rong et al., 2019) randomly drops edges to avoid the over-smoothing problem. Graph Random Neural Networks (Grand) (Feng et al., 2020) randomly drop nodes to generate data augmentation for improving model generalization. Grand+ improves the scalability over Grand by **pre-computing** a general propagation matrix and employ it to perform data augmentation (Feng et al., 2022). As shown in Section C, the key difference between GRAND(+) and `RSC` is that *`RSC` does not drop any node. Instead `RSC` drops all of the out edges for a set of nodes only during backward pass. Moreover, the drop pattern are evolving during the training process.* This can be viewed as the "structural dropedge". However, unlike Dropedge (Rong et al., 2019), `RSC` drop the column-row pairs according to the euclidean norm instead of uniformly dropping.

## D. Experimental Settings

### D.1. Software and Hardware Descriptions

All experiments are conducted on a server with four NVIDIA 3090 GPUs, four AMD EPYC 7282 CPUs, and 252GB host memory. We implement all models based on Pytorch and Pytorch Geometric. During our experiments, we found that the version of Pytorch, Pytorch Sparse, and Pytorch Scatter can significantly impact the running speed of the baseline. Here we list the details of our used packages in all experiments in Table 5.

*Table 5:* Package configurations of our experiments.

| Package | Version |
|---|---|
| CUDA | 11.1 |
| pytorch_sparse | 0.6.12 |
| pytorch_scatter | 2.0.8 |
| pytorch_geometric | 1.7.2 |
| pytorch | 1.9.0 |
| OGB | 1.3.2 |

### D.2. Statistics of benchmark datasets

The statistics for all used datasets are shown in Table 6. We follow the standard data splits and all datasets are directly downloaded from Pytorch Geometric or the protocol of OGB (Hu et al., 2020).

### D.3. Hyperparameter Settings

Regarding Reddit and Yelp dataset, we follow the hyperparameter reported in the respective papers as closely as possible. Regarding *ogbn-proteins* and *ogbn-products* dataset, we follow the hyperparameter configurations and codebases provided

*Table 6:* Dataset Statistics.

| Dataset | Task | Nodes | Edges | Classes | Label Rates |
|---|---|---|---|---|---|
| Reddit | multi-class | 232,965 | 11,606,919 | 41 | 65.86% |
| Yelp | multi-label | 716,847 | 6,977,409 | 100 | 75.00% |
| *ogbn-proteins* | binary-Class | 132,534 | 39,561,252 | 2 | 65.00% |
| *ogbn-products* | multi-class | 2,449,029 | 61,859,076 | 47 | 8.03% |

on the OGB (Hu et al., 2020) leader-board. Please refer to the OGB website for more details. The optimizer is Adam for all these models. All methods terminate after a fixed number of epochs. We report the test accuracy associated with the highest validation score. Table 10 summarize the hyperparameter configuration of GraphSAINT. Table 7, Table 8, and Table 9 summarize the hyperparameter configuration of full-Batch GCN, GraphSAGE, and GCNII, respectively.

*Table 7:* Configuration of Full-Batch GCN.

| Dataset | Training | | | Archtecture | | |
|---|---|---|---|---|---|---|
| | Learning Rates | Epochs | Dropout | BatchNorm | Layers | Hidden Dimension |
| Reddit | 0.01 | 400 | 0.5 | Yes | 3 | 256 |
| Yelp | 0.01 | 500 | 0.1 | Yes | 3 | 512 |
| *ogbn-proteins* | 0.01 | 1000 | 0.5 | No | 3 | 256 |
| *ogbn-products* | 0.001 | 500 | 0.5 | No | 3 | 256 |

*Table 8:* Configuration of Full-Batch GraphSAGE.

| Dataset | Training | | | Archtecture | | |
|---|---|---|---|---|---|---|
| | Learning Rates | Epochs | Dropout | BatchNorm | Layers | Hidden Dimension |
| Reddit | 0.01 | 400 | 0.5 | Yes | 3 | 256 |
| Yelp | 0.01 | 500 | 0.1 | Yes | 3 | 512 |
| *ogbn-proteins* | 0.01 | 1000 | 0.5 | No | 3 | 256 |
| *ogbn-products* | 0.001 | 500 | 0.5 | No | 3 | 256 |

*Table 9:* Configuration of Full-Batch GCNII.

| Dataset | Training | | | Archtecture | | |
|---|---|---|---|---|---|---|
| | Learning Rates | Epochs | Dropout | BatchNorm | Layers | Hidden Dimension |
| Reddit | 0.01 | 400 | 0.5 | Yes | 4 | 256 |
| Yelp | 0.01 | 500 | 0.1 | Yes | 4 | 256 |
| *ogbn-proteins* | 0.01 | 1000 | 0.5 | No | 4 | 256 |

## E. More experiment results

The running time of the greedy algorithm is shown in 11. We also visualize the allocated $k_l$ for each layer across iterations in Figure 7, and the degree of picked nodes in Figure 8. Here we use Reddit dataset for the case study. We observe that the $k_l$ assigned by `RSC` evolves along with the training.

*Table 10:* Training configuration of GraphSAINT.

| Dataset | RandomWalk Sampler | | Training | | | Archtecture | | |
|---|---|---|---|---|---|---|---|---|
| | Walk length | Roots | Learning Rates | Epochs | Dropout | BatchNorm | Layers | Hidden Dimension |
| Reddit | 4 | 8000 | 0.01 | 40 | 0.1 | Yes | 3 | 128 |
| Yelp | 2 | 8000 | 0.01 | 75 | 0.1 | Yes | 3 | 512 |
| *ogbn-products* | 3 | 60000 | 0.01 | 20 | 0.5 | No | 3 | 256 |

*Table 11:* The running time (second) of the greedy algorithm.

| | Reddit | Yelp | *ogbn-proteins* | *ogbn-products* |
|---|---|---|---|---|
| GCN | 0.03 | 0.03 | 0.03 | 0.03 |
| GraphSAGE | 0.02 | 0.02 | 0.03 | 0.03 |
| GCNII | 0.05 | 0.05 | 0.06 | - |

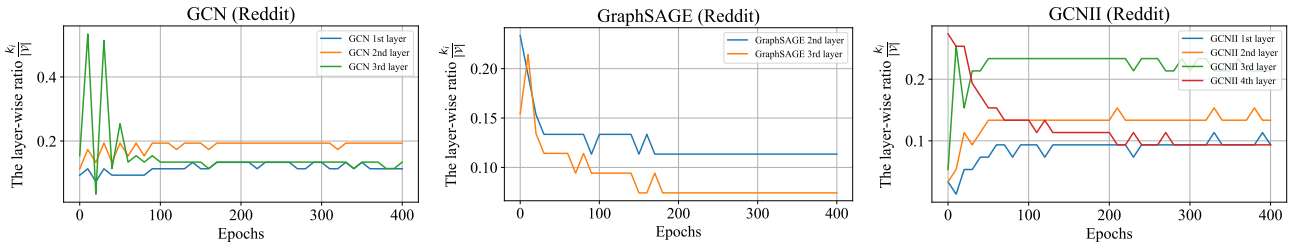


*Figure 7:* The allocated layer-wise $k_l$ for GCN, GraphSAGE and GCNII on Reddit, where budget $C$ is set as 0.1. The input of the `SpMM` in the first GraphSAGE layer does not require gradient and thus absent in the Figure (Appendix A.3).

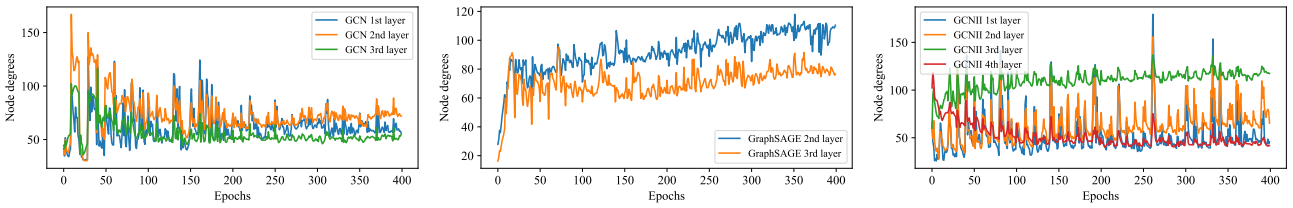


*Figure 8:* The averaged degrees of nodes picked by top-$k$ sampling along the whole training process, where the applied dataset is Reddit and overall budget $C$ is set as 0.1.

### E.1. Additional Ablation Results to the Resource Allocation Algorithm (Figure 6)

Due to the page limit, we present more ablation study on the resource allocation algorithm here. Specifically, in Figure 9, we compare `RSC` to the uniform allocation on ogbn-proteins dataset with GCN, GraphSAGE, and GCNII, respectively. In Figure 10, we compare `RSC` to the uniform allocation on Yelp dataset with GCN, GraphSAGE, and GCNII, respectively. We conclude that `RSC` generally outperforms the uniform allocation strategy.

### E.2. Hyperparameter Sensitivity Analysis

Here we analyze the impacts of the main hyperparameters of `RSC` : **(1)** the budget $C$, which controls the efficiency-accuracy trade-off; **(2)** the step size $\alpha$ in the greedy Algorithm 1; **(3)** when switching back to the original sparse operations. In Figure 12, we vary only one of them with the others fixed. We conclude **(1)** larger budget $C$ leads to better accuracy with smaller speedup, since we are using more computational resources to approximate the full operation. **(2)** larger step size $\alpha$ leads

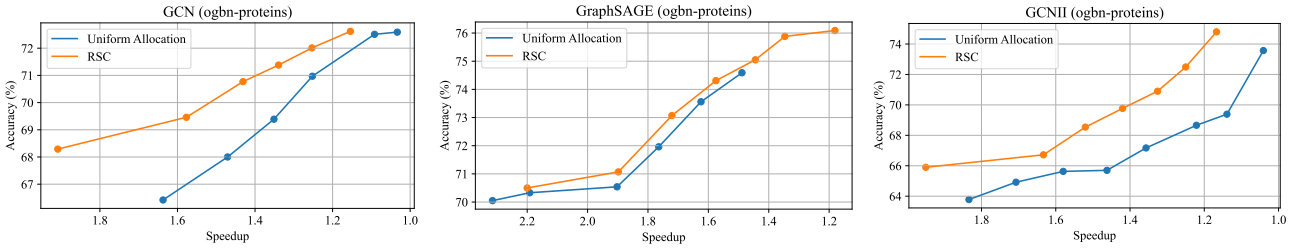


*Figure 9:* The Pareto frontier of the accuracy-efficiency trade-off for RSC and the uniform allocation. The dataset is ogbn-proteins. Here we disabled the caching and switch mechanism for a fair comparison.

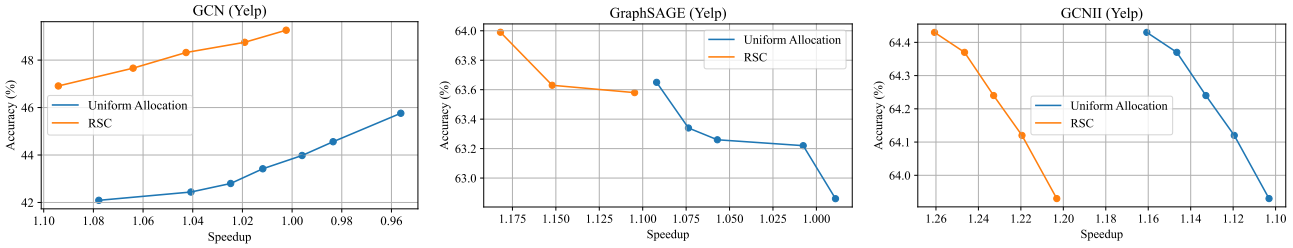


*Figure 10:* The Pareto frontier of the accuracy-efficiency trade-off for RSC and the uniform allocation. The dataset is Yelp. Here we disabled the caching and switch mechanism for a fair comparison.

to marginally larger speedup since the greedy algorithm will terminate earlier. Also the step size $\alpha$ does not affect the model accuracy a lot. In practice, we set $\alpha = 0.02|\mathcal{V}|$. **(3)** The later we switch back to the original operation, the larger the accuracy drop and the smaller the speedup, it is equivalent to using less resources to approximate the full operation epoch-wisely. Thus, we apply RSC for $80\%$ of the total epochs to balance the trade-off.

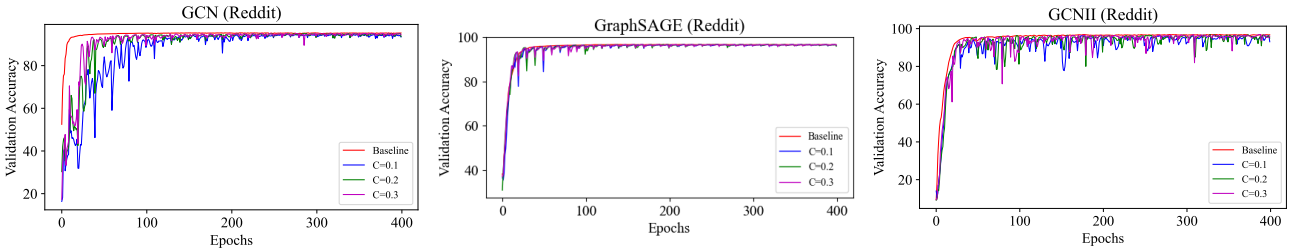


*Figure 11:* Learning curves for validation accuracy under different overall budget $C$ on Reddit dataset. Here we disabled the caching and switching mechanism for ablating the effect of $C$.

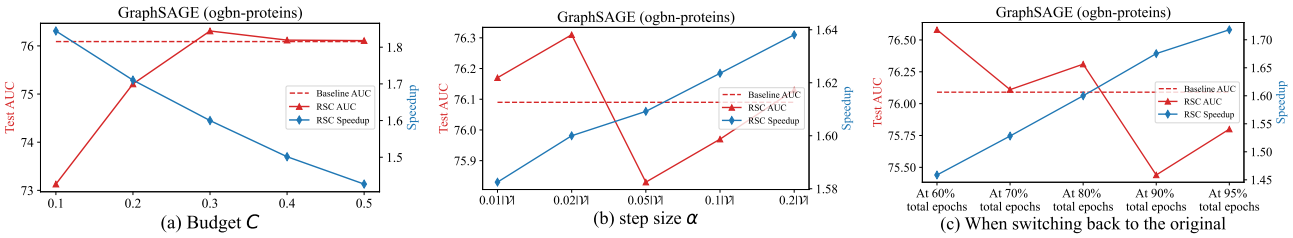


*Figure 12:* Hyperparameter analysis w.r.t. the budget $C$, the step size $\alpha$ in Algorithm 1, and when switching back to the original operations. The model is GraphSAGE and the applied dataset is *ogbn-proteins*.