Scalable GNN Training via Parameter Freeze and Layer Detachment

Chang Gong[1], Boyu Yang[1], Weiguo Zheng[1](✉), Xing Huang[2], Weiyi Huang[2], Jie Wen[2], Shijie Sun[2], and Bohua Yang[2]

[1] School of Data Science, Fudan University, Shanghai, China
changgong22@m.fudan.edu.cn, {yangby19,zhengweiguo}@fudan.edu.cn
[2] Tencent, Shenzhen, China
{healyhuang,vinniehuang,welkinwen,cedricsun,bohuayang}@tencent.com

Abstract. Graph neural networks (GNNs) have been widely used in various applications. However, training large-scale GNNs with minimal cost remains an ongoing challenge. In scenarios where GPU memory is constrained, sampling a minibatch for training becomes a common practice. Nonetheless, it introduces the neighbor explosion problem. While previous sampling-based methods have made efforts to mitigate this problem, they still suffer from drawbacks such as missing information or inaccurate node representation. To address this challenge, we present MENSA, a lightweight yet powerful framework aimed to alleviate significant time and memory costs. MENSA comprises two components: EMFD and SFAA. EMFD strategically freezes trainable parameters in the first few layers to avoid unnecessary gradient computation. Additionally, it detaches these layers from the entire computational graph, effectively reducing its depth and saving GPU memory. Since fixed parameters might degrade GNN's capability, SFAA addresses this concern by sampling a considerably smaller number of neighbors, constructing a compact computational graph, and activating all parameters for message propagation. By alternately performing EMFD and SFAA in each epoch, MENSA achieves superior F_1 scores compared to other sampling-based methods. Noticeably, it exhibits 1.5x-3.5x speedup in convergence speed on most datasets and yields savings of 50%-80% in GPU memory.

Keywords: Scalable GNN · Parameter Freeze · Layer Detachment.

1 Introduction

With the development of deep learning techniques, graph neural networks (GNNs) have been a popular tool for various graph mining tasks, such as node classification [6, 11], link prediction [12, 23] and graph classification [13, 18]. The core mechanism of GNNs is message passing computational paradigm [7]. By propagating for L iterations, nodes can aggregate information within L-hop neighbors, which helps to achieve expressive representation for nodes.

1.1 The Existing Methods and Limitations

As the size of graphs grows, training large-scale GNNs at a low cost has become a critical concern. Handling graphs that surpass GPU memory limit poses a significant challenge, since it is impractical to load the entire structure into a GPU for fullbatch training. Therefore, sampling a minibatch of nodes to train GNNs has been a common practice. However, minibatch training cannot solve neighbor explosion problem. To accomplish L-step message passing, we need to load in-minibatch nodes and their L-hop neighbors into a GPU, which causes *neighbor explosion, i.e., the number of nodes to grow exponentially with* L.

Many works have concentrated on designing elaborate techniques to reduce the risk of neighbor explosion. Among them, sampling-based methods are the most widely studied, which can be divided into three categories: node-wise sampling [8], layer-wise sampling [1, 3, 25], and subgraph-wise sampling [4, 21, 22]. The first two both use recursive sampling by layer and the difference is that node-wise selects neighbors independently for each node, while layer-wise samples shared nodes for one layer. Subgraph-wise sampling extracts a large number of small-size subgraphs and restricts message passing within each one.

Node-wise sampling is more commonly employed in practice than layer-wise and subgraph-wise sampling. For layer-wise sampling, there is a high probability that nodes in one layer lack neighbors in the subsequent layer, hindering them from aggregating sufficient information from neighbors [4]. Conversely, node-wise sampling ensures adequate message passing by sampling several neighbors for each node. For subgraph-wise sampling, all nodes within the subgraph participate in loss computation, which results in inaccurate node representations, as some neighbors of the sampled nodes may not be included in the subgraph [22]. However, node-wise sampling focuses only on the loss of nodes within minibatch and treats their L-hop neighbors as auxiliary nodes. Despite these considerations, existing node-wise methods still fail to mitigate neighbor explosion [3].

1.2 Our Approach and Contributions

When training, it is necessary to retain the whole computational graph in GPU memory. However, during inference phase, we can optimize GPU memory usage by discarding the first $(k-1)$ layers when propagating to the k-th layer. Moreover, we observe that trainable parameters in the first $(L-1)$ layers change significantly slower than those in the last layer. These findings inspire a novel approach aimed at reducing the depth of computational graph, which is called **EMFD** (**E**xpanding **M**ore neighbors while **F**reezing parameters at **D**eep layers). EMFD freezes parameters in the first k layers to alleviate the cost associated with gradient computation of these layers. Furthermore, it detaches these layers from the entire computational graph, allowing parallel computation of the k-th layer's output. Specifically, the whole graph is partitioned into subgraphs and we perform forward propagation on each subgraph in parallel without saving intermediate results. With the depth of computational graph decreasing, EMFD can sample more neighbors for each node to enhance representation.

Meanwhile, we acknowledge that keeping parameters of the first k layers fixed throughout the whole training phase may diminish the expressive capability of deep GNNs. Thus, we propose an additional mechanism called **SFAA**(**S**ampling **F**ewer neighbors while **A**ctivating **A**ll parameters). SFAA adopts a simple strategy to update parameters: sampling significantly fewer neighbors than EMFD to control computational graph size. To leverage more information, neighbors are sampled based on their degrees. Then it activates all layers' parameters and performs forward and back propagation through the entire computational graph.

By combining these two strategies EMFD and SFAA, we introduce a lightweight yet powerful framework to mitigate neighbor explosion problem: **MENSA** (**M**ix of **E**MFD a**N**d **S**F**A**A). MENSA alternates between SFAA and EMFD in each epoch and aggregates information from different receptive fields: EMFD gathers neighboring nodes, while SFAA emphasizes distant ones. In experiments, we compare MENSA with six sampling-based methods across six large-scale graphs. Results demonstrate MENSA outperforms all baselines in terms of accuracy, achieving 1.5x-3.5x speedup in convergence speed and saving 50%-80% memory compared to GraphSAGE. We also compare MENSA with recently popular decoupled GNNs and find that MENSA consistently achieves higher F_1 scores.

Contributions. The major contributions are summarized as follows:

(1) We observe that during the training of a deep GNN, the trainable parameters in the first few layers exhibit slower changes compared to the last layer. It motivates us to implement a strategy of parameter freezing to reduce time and memory costs.
(2) We introduce MENSA, a lightweight yet powerful framework designed to address the issue of neighbor explosion, by comprising two complementary strategies, EMFD and SFAA. EMFD reduces the depth of computational graphs, and SFAA restores the model's capability by allowing parameter updates.
(3) Extensive experimental results on real-world graphs confirm both the effectiveness and efficiency of MENSA.

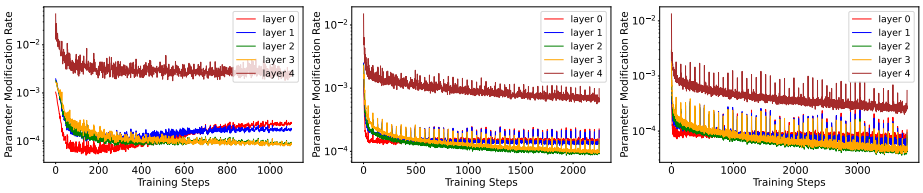
2 Preliminaries

In this section, we provide an overview of the necessary background and notations. A graph is represented as $G(V, E)$, where $V = \{v_1, v_2, ..., v_n\}$ is node set and E is edge set. Each node $v_i \in V$ is associated with an initial feature $x_i \in \mathbb{R}^d$ and a label $y_i \in [C]$. $\mathcal{N}(v_i)$ denotes neighbors of v_i. The goal of supervised node classification task is to find a function $f : V \mapsto [C]$ that maximizes prediction accuracy. Nowadays, f is often modeled as a graph neural network (GNN) owing to its robust capability in representing graphs.

2.1 GNN and Message Passing

GNN follows the two-step computation paradigm of message passing [7]:

$$m^l_{\mathcal{N}(v_i)} = \phi(\{h^l_{v_j} W^l | v_j \in \mathcal{N}(v_i)\}), \quad h^{l+1}_{v_i} = \psi(h^l_{v_i} W^l, m^l_{\mathcal{N}(v_i)}) \tag{1}$$



(a) Flickr (b) Arxiv (c) Reddit

Fig. 1: Rate of parameter modification. Vertical axis is on a logarithmic scale.

where $h^l_{v_i}$ represents the embedding of v_i after l times propagation. W^l is trainable parameter for feature transformation at step l. In the first step, node v_i collects information from $\mathcal{N}(v_i)$ and performs aggregation by function ϕ. In the second step, it combines last propagation's feature $h^l_{v_i}$ and gathered information $m^l_{\mathcal{N}(v_i)}$ to update its current feature by function ψ. Notice that all nodes carry out these operations simultaneously, leveraging efficient dense-dense and sparse-dense matrix multiplication [16].

2.2 Minibatch Training and Computational Graph

As the size of real-world graphs grows exponentially, it becomes impossible to load the whole graph into a single GPU for fullbatch training. Consequently, minibatch training has become a prevalent approach for practical applications. To obtain the output of node v, we need to consider L-hop neighbors to construct a hierarchical structure known as *computational graph (CG)*(as shown in Figure 2(b)). It is a tree with root v and from bottom to up the $(L-i)$-th layer corresponds to i-th hop neighbors of v. However, the number of L-hop neighbors grows exponentially with L, leading to a CG that exceeds the memory capacity of a GPU. To address this challenge, GraphSAGE [8] employs node sampling from $\mathcal{N}(v)$, a practice that will be the foundation for further discussion.

3 Motivation and Intuition

In this section, we conduct a thorough analysis of memory overhead of GNN training and identify cost-ineffective operations. These insights lay the groundwork for our new framework, which will be introduced in the next section.

3.1 Memory Analysis of GNN Training

Let $H^l \triangleq (h^l_{v_1}, h^l_{v_2}, ..., h^l_{v_n})^T$ represent node feature matrix at layer l. $V^l \triangleq \nabla_{H^l}\mathcal{L}$ and V^l_i denotes the i-th row of V^l. Forward computation of GNN follows Equation (1) , while backward propagation can be calculated by chain rule [14]:

$$V^l_i = \sum_{v_j \in \mathcal{N}(v_i)} (\nabla_{h^l_{v_i}} \psi(h_{v^l_j} W^l, m^l_{\mathcal{N}(v_j)})) V^{l+1}_j, \quad \nabla_{W^l}\mathcal{L} = \sum_{v_i \in V} (V^l_i)^T \nabla_{W^l} V^l_i \quad (2)$$

Equation (2) indicate that when computing the gradient of parameters at l-th layer (i.e., $\nabla_{W^l}\mathcal{L}$), not only the gradient of $(l+1)$-th layer's output (i.e., V^{l+1}) but also the output of l-th layer (i.e., $\psi(h_{v_j^l}W^l, m^l_{\mathcal{N}(v_j)})$) is required. In other words, during forward propagation, all intermediate results and the whole computational graph are saved in GPU memory for gradient computation. Suppose minibatch size is B and s neighbors are sampled for each node in a minibatch, the memory cost of storing intermediate results is $\mathcal{O}(Bds^L)$ for a L-layer GNN, which limits GNN's depth.

However, during inference with GNNs, there is no necessity to store intermediate outputs, as gradients do not flow back through computational graph. Moreover, we can reduce space complexity in the following ways: the whole computational graph is divided into two parts: the propagation of first k layers can be accomplished in parallel by multiple CPU machines with large memory (see lines 3-8 of Algorithm 1), and the remaining $(L-k)$ layers are transferred to GPU for further inference. Thus, GPU memory overhead reduces to $\mathcal{O}(Bds^{L-k})$.

In the training phase, the computational graph cannot be split since gradient depends on it for backward propagation. However, we can detach the first k layers from computational graph if the trainable parameters of these layers are fixed. Then these layers can perform inference without gradient computation, which significantly reduces time and space complexity.

3.2 Cost-ineffective Operation in GNN Training

Although freezing parameters can mitigate GPU memory overhead during training, it is still an open problem whether it will affect the final results of training. Fortunately, The following theorem confirms the plausibility of this idea.

Theorem 1. *In a L-layer GNN, we assume that hidden, input and output dimensions are d, d_{in}, d_{out} and satisfy $d_{in} > d_{out}$ and $d \gg d_{out}$. Define the rate of parameter updates of layer k as follows:*

$$R(k) \triangleq \frac{||\nabla_{W^k}\mathcal{L}||_F}{\sqrt{\nabla_{W^k}\mathcal{L}.shape[0] \times \nabla_{W^k}\mathcal{L}.shape[1]}}$$

Then, the rate of parameter updates in the first $(L-1)$ layers is markedly lower compared to that in the last layer. In other words, $R(L) \gg R(k)(k < L)$.

We explain Theorem 1 from two different perspectives:

Real-world Examples. We utilize three real-world graphs (Flickr, Arxiv, and Reddit) to justify the above theorem. As shown in Figure 1, only the last layer has relatively drastic parameter changes compared to former layers. In other words, parameters of first $(L-1)$ layers remain essentially unchanged. Thus, we can omit the computation of these layers' gradient to save time and memory cost.

Theoretical Analysis. To simplify the problem, we consider an L-layer GCN using ReLU as an activation function σ without normalization techniques such

as batch normalization. The forward computation can be expressed as follows:

$$H^k = AZ^{k-1}W^k, \quad Z^k = \sigma(H^k), \quad \mathcal{L} = f(Z^L) \quad (k = 1, ..., L, Z^0 = X)$$

A and X are normalized adjacency matrix and input node features. f is a loss function. Now we compute $\nabla_{W^k}\mathcal{L}$ $(k = 1, ..., L-1)$ by chain rule:

$$\nabla_{W^k}\mathcal{L} = \nabla_{W^k}H^k \cdot \nabla_{H^k}Z^k \cdot \prod_{i=k+1}^{L} (\nabla_{Z^{i-1}}H^i \cdot \nabla_{H^i}Z^i) \cdot \nabla_{Z^L}\mathcal{L}$$

Here, $\nabla_{W^k}H^k = I_{d_{k+1}} \otimes (AZ^{k-1})^T$, $\nabla_{H^k}Z^k = \mathbb{1}_{>0}(H^k)$, $\nabla_{Z^{i-1}}H^i = W^i \otimes A^T$, where $\otimes$ represents Kronecker product. Through vectorization techniques (for $X \in \mathbb{R}^{m\times n}, Vec(X) \in \mathbb{R}^{mn}$), we obtain the following results:

$$Vec(\nabla_{W^k}\mathcal{L}) = (I_{d_{k+1}} \otimes (AZ^{k-1})^T)\Bigg\{Vec(\mathbb{1}_{>0}(H^k)) \odot \prod_{i=k+1}^{L-1}\Big[(W^i \otimes A^T) \odot Vec(\mathbb{1}_{>0}(H^i))\Big](W^L \otimes A^T)\Bigg\}Vec(\nabla_{Z^L}\mathcal{L})$$

For parameters at last layer, $Vec(\nabla_{W^L}\mathcal{L}) = (I_{d_{L+1}} \otimes (AZ^{L-1})^T)Vec(\nabla_{Z^L}\mathcal{L})$. By comparing gradient at W^L and W^k $(k = 1, ..., L-1)$, we identify the following two significant differences:

1. $Vec(\nabla_{W^k}\mathcal{L})$ has $Vec(\mathbb{1}_{>0}(H^i))$ while $Vec(\nabla_{W^L}\mathcal{L})$ does not. That means many negative values in $Vec(\nabla_{W^k}\mathcal{L})$ are forced to zero when applying the chain rule, resulting in a decrease of $||\nabla_{W^k}\mathcal{L}||_F$.

2. All W^i participate in computing $Vec(\nabla_{W^k}\mathcal{L})$, except W^k. When all elements in all W^i are independently and identically distributed (it is easily satisfied during parameter initialization), larger-size W^i contributes more to $||\nabla_{W^k}\mathcal{L}||_F$. Given assumptions of Theorem 1, W^L has the smallest shape (d, d_{out}), thus the Frobenius norm of $Vec(\nabla_{W^L}\mathcal{L})$ should be the largest. Specifically, when $d \gg d_{out}$, the difference will be more remarkable.

Combining the above two aspects, based on the Theorem 1 assumptions, we conclude that $R(L)$ is significantly larger than $R(k)$ $(k = 1, ..., L-1)$. □

4 Methodology

In this section, we present a new GNN training framework that hybrids multi-scale receptive fields. This framework uses two strategies alternately to get node embedding. The first is **EMFD** (**E**xpanding **M**ore neighbors while **F**reezing parameters at **D**eep layers), and the other is **SFAA** (**S**ampling **F**ewer neighbors while **A**ctivating **A**ll parameters). The overview of the framework is illustrated in Figure 2.

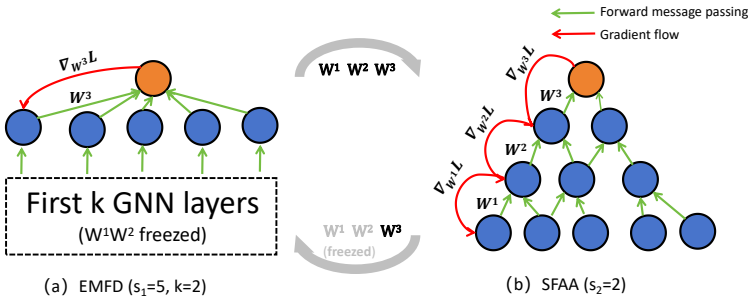


Fig. 2: MENSA framework. Orange and blue circles represent in-minibatch node and its sampled multi-hop neighbors. For SFAA, only $s_2 = 2$ neighbors are sampled for each node and sampling is processed layer by layer for $L = 3$ times. To get orange node's output, features are input from the bottom and the message passes through green arrows. Gradients are propagated backward through red arrows to each layer. For EMFD, each node samples $s_1 = 5$ neighbors and only $L - k = 1$ layer is preserved. Parameters (W^1, W^2) at first $k = 2$ layers are frozen and we perform parallel forward computation to obtain output of layer $k = 2$. Gradient flow terminates at $k+1 = 3$-rd layer. SFAA and EMFD execute alternately at a certain frequency.

4.1 EMFD

Theorem 1 shows that for a deep GNN, frequent parameter updates across all layers may incur additional costs with marginal benefits. Given the relatively slow evolution of parameters in deeper layers, a strategy of freezing them can be employed to mitigate unnecessary time and space costs.

The whole process of EMFD for one epoch is summarized in Algorithm 1. Here, superscript j denotes j-th layer while subscript j means the j-th part. Firstly, we utilize METIS [10] algorithm to partition the whole graph, enabling each subgraph to fit within a single GPU memory (line 2). Then, the first k layers' parameters are fixed, so we detach these layers from the original GNN and perform parallel computation to get each node's embedding at k-th layer (lines 3-8). During this phase, there is no need to retain the complete computational graph; only the final output of the k-th layer needs to be preserved. Since memory consumption is only related to the size of partitioned subgraphs and is independent of layer number, we can increase each partition's size based on GPU memory limit and reduce information loss. Furthermore, for moderate-size graphs, it is even possible to load the whole structure into a single GPU memory for inference of first k layers. After that, given a minibatch $\mathcal{B}_i$, we recursively sample s_1 neighbors for each node and build a computational graph $\mathcal{G}_i$, which can be used for message propagation through the following $(L - k)$ layers (lines 9-11). Notice that gradient flow terminates at the $(k + 1)$-th layer and no gradient is computed for the first k layers. Thus, only the left parameters (i.e., $W^{k+1}, \ldots, W^L$) will be updated in subsequent training (lines 12-13).

Algorithm 1 EMFD (for one epoch)

Input: graph G, feature X, label Y, EMFD fanout s_1, GNN layers L, fixed layers k.

1: $H^0 = X$
2: $\mathcal{B}_1, ..., \mathcal{B}_q = \text{Split}(G)$
3: **for** $i = 1 : q$ **do**
4: // Execute in parallel, without gradient computation.
5: $G_i = \text{InducedSubgraph}(G, \mathcal{B}_i)$
6: $H^j|_{\mathcal{B}_i} = \text{GNN}^j(H^{j-1}|_{\mathcal{B}_i}, G_i, W^j)$ $(j = 1, ..., k)$
7: **end for**
8: $H^k = \text{Concat}(\{H^k|_{\mathcal{B}_i} | i = 1, ..., q\})$
9: **for** $i = 1 : q$ **do**
10: $\mathcal{G}_i = \text{BuildCG}(G, L - k, \mathcal{B}_i, s_1)$
11: $H^j = \text{GNN}^j(H^{j-1}, \mathcal{G}_i^{j-k}, W^j)$ $(j = k + 1, ..., L)$
12: $\mathcal{L} = \text{loss function}(H^L, Y|_{\mathcal{B}_i})$
13: $W^j = W^j - \eta \nabla_{W^j} \mathcal{L}$ $(j = k + 1, ..., L)$
14: **end for**

Line 10 shows that the computational graph in the training process has only $(L - k)$ layers, which decreases the GPU memory overhead from $\mathcal{O}(Bds_1^L)$ to $\mathcal{O}(Bds_1^{L-k})$ (the memory cost for first k layers' inference is just $\mathcal{O}(Bd)$, which is relatively small). Thus, we can appropriately increase the value of s_1 to gather more neighbor information and reduce sampling bias.

4.2 SFAA

While the parameters of the first k layers exhibit slow changes, keeping them fixed throughout the whole training phase may yield notable disparities between intermediate results of EMFD and those derived from full-parameter training, which negatively impacts final accuracy. Meanwhile, a deep GNN degrades to a shallow one since the first k layers' parameters are fixed, thereby diminishing the model's expressive capability.

To compensate for the deficiencies of EMFD, we use a simple approach named SFAA to update parameters of the first k layers. The detailed procedure of SFAA is described in Algorithm 2. Given a minibatch $\mathcal{B}_i$, we recursively sample s_2 neighbors for each node and build a computational graph $\mathcal{G}_i$ (line 4). Specifically, we follow three principles:

(1) s_2 is much smaller than s_1 in EMFD to ensure the size of $\mathcal{G}_i$ falls into an acceptable range.
(2) Neighbors are sampled based on their degrees (line 4), which helps to focus more on high-degree ones that can provide more information. It can also avoid the structure sparsity of computational graphs.
(3) When layer size exceeds the upper bound, we discard several low-degree nodes. Since batch size B and fanout s_2 are small, this situation occurs infrequently. However, it still ensures computational graph's size is controllable.

After that, all layers' parameters are activated, and normal forward and back propagation are performed through $\mathcal{G}_i$ (lines 5-7).

4.3 Put It All Together: MENSA

EMFD can be thought of as using more information from neighboring nodes, while SFAA focuses more on the features of distant nodes. A straightforward combination is to execute EMFD and SFAA sequentially within the same minibatch. In other words, it computes node embeddings in the same minibatch in two completely different ways during two consecutive iterations, which easily leads to unstable parameter updates. Furthermore, before invoking EMFD, it needs to perform an additional forward propagation to obtain the k-th layer's output. If the frequency of EMFD invocation is too high, it can result in an untenable increase in time costs.

To address these problems, we present an efficient and powerful framework by performing SFAA for all minibatches and then executing EMFD in one epoch. This framework is called **MENSA**, short for **M**ix of **E**MFD a**N**d **S**F**A**A. As outlined in Algorithm 3, SFAA and EMFD are alternatively invoked in the same epoch to help nodes gather information from receptive fields of varying scales.

Algorithm 2 SFAA (for one epoch)

Input: graph G, feature X, label Y, SFAA fanout s_2, GNN layers L, total node degree d.

1: $H^0 = X$
2: $\mathcal{B}_1, ..., \mathcal{B}_q = \text{Split}(G)$
3: **for** $i = 1 : q$ **do**
4: $\quad \mathcal{G}_i = \text{BuildCG}(G, L, \mathcal{B}_i, s_2, d)$
5: $\quad H^j = \text{GNN}^j(H^{j-1}, \mathcal{G}_i^j, W^j)$ $(j = 1, ..., L)$
6: $\quad \mathcal{L} = \text{loss function}(H^L, Y|_{\mathcal{B}_i})$
7: $\quad W^j = W^j - \eta \nabla_{W^j} \mathcal{L}$ $(j = 1, ..., L)$
8: **end for**

Algorithm 3 MENSA

Input: graph G, EMFD fanout s_1, SFAA fanout s_2, GNN layers L, fixed layers k.

1: **for** $e = 1 : \text{total\_epoch}$ **do**
2: $\quad \text{SFAA}(G, L, s_2)$
3: $\quad \text{EMFD}(G, L, k, s_1)$
4: **end for**
5: **return** $\{W^1, ..., W^L\}$

4.4 Complexity Analysis

Theorem 2. *Given a computational graph $\mathcal{G}$, suppose input and hidden dimension are both d, then the time complexity of propagating forward through $\mathcal{G}$ is $\mathcal{O}(N_{all}d^2 + E_{all}d)$, where N_{all} and E_{all} represent node and edge number.*

Proof. Message passing between l-th layer and $(l+1)$-th layer of $\mathcal{G}$ consists of two steps: feature transformation and propagation, which can be formulated as a dense-dense and sparse-dense matrix multiplication(i.e., short as SpMM)

$$\hat{H}^l = H^l W^l, \quad H^{l+1} = SpMM(A^l, \hat{H}^l)$$

where A^l represents the 0-1 adjacency matrix of these two layers. Suppose the number of nodes at l-th layer is N_l, then the first and second equation cost $\mathcal{O}(N_l d^2)$ and $\mathcal{O}(\|A^l\|_0 d))$, respectively. Thus, the overall time complexity is: $\sum_{i=1}^{L}(\mathcal{O}(N_l d^2 + \|A^l\|_0 d)) = \mathcal{O}(N_{all} d^2 + E_{all} d)$. □

Time Complexity. For EMFD, it includes two parts: detached first k layers and the left $(L-k)$ layers. The former is not included in GPU time complexity since we can leverage multiple CPU machines for parallel processing of these forward computations. For the latter one, according to Theorem 2, the time complexity is $\mathcal{O}((Bd^2+Bd)s_1^{L-k})$. Similarly, the time complexity of SFAA is estimated to be $\mathcal{O}((Bd^2+Bd)s_2^L)$. Thus, the overall time complexity is $\mathcal{O}((Bd^2+Bd)(s_1^{L-k}+s_2^L))$.
Space Complexity. The space complexity of MENSA encompasses three components: trainable parameters, intermediate outcomes, and computational graph structure. The first costs $\mathcal{O}(Ld^2)$, shared by EMFD and SFAA. For EMFD, we only need to consider the intermediate output of the latter $(L-k)$ layers, which is $\mathcal{O}(Bds_1^{L-k})$; while for SFAA, the intermediate results cost $\mathcal{O}(Bds_2^L)$. Lastly, the computational graphs of EMFD and SFAA consume $\mathcal{O}(Bs_1^{L-k})$ and $\mathcal{O}(Bs_2^L)$. Thus, the overall space complexity is $\mathcal{O}(Ld^2 + max(Bds_1^{L-k} + Bs_1^{L-k}, Bds_2^L + Bs_2^L))$. Compared to GraphSAGE with $\mathcal{O}(Ld^2 + Bds_1^L + Bs_1^L)$, MENSA exhibits noticeably lower memory consumption, since $s_2 \ll s_1$.

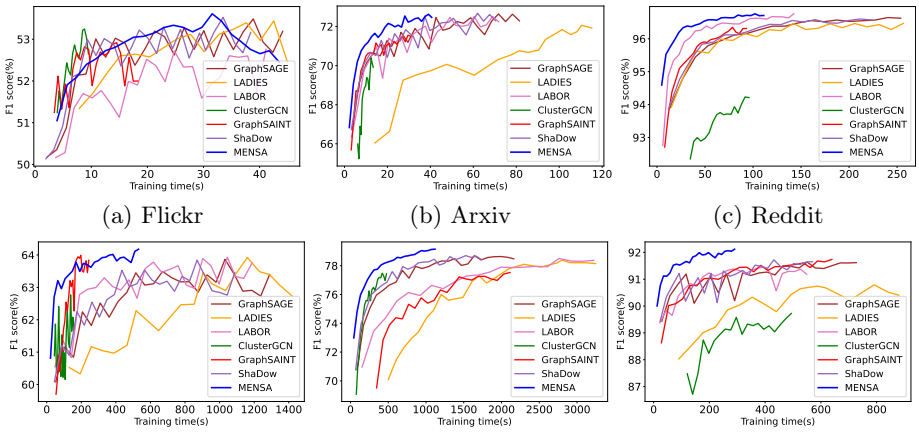
5 Experiments

5.1 Experimental Settings

Datasets. We use six real-world graphs for evaluation: Flickr [8], Arxiv [9], Reddit [22], Yelp [22], Amazon [22], and Products [9]. These graphs contain thousands or millions of nodes/edges and are widely studied in previous works. Detailed information of these datasets can be found in Table 1.
Baselines. We compare our method with six classic sampling-based algorithms, including GraphSAGE [8], LADIES [25], LABOR [1], ClusterGCN [4], GraphSAINT [22] and ShaDow [21]. To further illustrate advantages of MENSA in terms of accuracy, we also compare it with several decoupled GNNs, including SGC [17], SIGN [6], SAGN [15], GAMLP [24], HopGNN [2] and LazyGNN [19].
Implementation Details. All models are implemented by DGL [16]. Experiments are conducted on a single machine equipped with two 12-core CPUs and a GeForce RTX 2080Ti. For the sake of fairness, the same GNN backbone is applied to all algorithms: five SAGEConv layers with 256 hidden size for Flickr, Arxiv, and Reddit, and four SAGEConv layers with 512 hidden size for other datasets. Batch size is fixed as 2048 and the total epoch number is 100. For GraphSAGE and LABOR, we select fanout $s \in \{5, 10\}$. For LADIES, we set the maximum layer size to $20k$. For ClusterGCN, the average size of subgraph is selected from $\{5k, 8k\}$. For GraphSAINT and ShaDow, the number of roots and random walk length are chosen from $\{2k, 5k\}$ and $\{2, 3\}$. For MENSA, we set $s_1 \in \{5, 10, 15\}$ and $s_2 \in \{1, 2, 3\}$.

Table 1: F_1 score (%) comparison on six datasets. **Bold** font and underline indicate the best and the second best result.

	Flickr	**Arxiv**	**Reddit**	**Yelp**	**Amazon**	**Products**
(#N,#E)	(89.3K,0.9M)	(0.2M,1.2M)	(0.2M, 23.2M)	(0.7M,14.0M)	(1.6M,264.3M)	(2.4M,123.7M)
GraphSAGE	53.32±0.17	71.52±0.52	96.62±0.05	64.25±0.10	78.10±0.04	79.41±0.21
LADIES	53.10±0.21	70.47±0.37	96.56±0.01	63.20±0.16	77.02±0.08	78.22±0.13
LABOR	52.90±0.30	71.37±0.35	96.66±0.02	64.26±0.13	77.99±0.08	79.19±0.44
ClusterGCN	51.95±0.16	69.81±0.44	94.20±0.38	63.13±0.30	76.71±0.19	78.01±0.25
GraphSAINT	53.05±0.07	71.09±0.17	96.32±0.06	63.98±0.22	77.16±0.11	78.91±0.31
ShaDow	53.20±0.14	71.23±0.20	96.45±0.03	64.11±0.12	77.44±0.20	78.70±0.18
MENSA	**53.48±0.16**	**71.71±0.26**	**96.71±0.03**	**64.49±0.18**	**78.50±0.07**	**80.37±0.06**



(a) Flickr (b) Arxiv (c) Reddit

(d) Yelp (e) Amazon (f) Products

Fig. 3: Convergence speed of various algorithms on six datasets.

5.2 Comparison with Sampling-based Methods

Accuracy Comparison. For node classification tasks, accuracy serves as the most important metric. As an improvement of GraphSAGE in terms of time and memory consumption, we also expect MENSA to achieve similar or better accuracy to GraphSAGE with the same parameter settings. All experiments run five times and average results are reported in Table 1. Clearly, MENSA demonstrates superior performance across a wide range of datasets, achieving the best results among these sampling-based methods in the average sense.

We find that GraphSAGE achieves lower accuracy than MENSA, even with the same hyperparameters. That is because deep GraphSAGE with large fanout may encounter an over-smoothing problem as too many nodes are aggregated, weakening the difference between nodes. However, at each iteration, MENSA only focuses on a subset of nodes in the receptive field of GraphSAGE, which reduces the risk of over-smoothing. What is more, shallow and long-range neigh-

Table 2: GPU memory consumption (MB) comparison on six datasets.

	Flickr	**Arxiv**	**Reddit**	**Yelp**	**Amazon**	**Products**
GraphSAGE	927	971	2140	1971	1306	3261
LADIES	714(-23%)	683(-30%)	1143(-47%)	838(-57%)	838(-36%)	2058(-37%)
LABOR	810(-13%)	771(-21%)	879(-59%)	1357(-31%)	510(-61%)	1435(-56%)
ClusterGCN	494(-47%)	957(-1%)	1197(-44%)	782(-60%)	993(-24%)	763(-77%)
GraphSAINT	**338(-64%)**	790(-19%)	1060(-50%)	757(-62%)	970(-26%)	831(-75%)
ShaDow	467(-50%)	781(-20%)	1123(-48%)	789(-60%)	957(-27%)	801(-75%)
MENSA	428(-54%)	**391(-60%)**	**783(-63%)**	**530(-73%)**	**373(-71%)**	**624(-81%)**

bors play different roles in embedding nodes. Alternatively exploring shallow and deep neighbors' information helps to capture their different importance.

Time and Memory Comparison. Next, we investigate the training efficiency and GPU memory of MENSA. Figure 3 depicts the variation of F_1 score on the validation set over training time, with results beyond runtime limit excluded. Clearly, MENSA outperforms node-wise (GraphSAGE) and layer-wise (LADIES and LABOR) approaches, achieving a convergence speedup of approximately 1.5x-3.5x on most datasets. Additionally, MENSA surpasses these methods in convergence accuracy. When compared to subgraph-wise methods (ClusterGCN, GraphSAINT, and ShaDow), MENSA, despite having a longer overall runtime, reaches higher accuracy more swiftly. Furthermore, subgraph-wise methods that sample smaller graphs exhibit a more volatile accuracy trend (like ClusterGCN with a high partition number), which indicates a more challenging convergence process. In contrast, MENSA updates parameters using only a small batch of nodes each iteration, and as the number of updates accumulates, it progressively mitigates sampling bias and stabilizes convergence.

Table 2 displays GPU memory consumption of various algorithms, with the numbers in parentheses indicating the percentage of memory reduction relative to GraphSAGE. MENSA, on average, achieves a GPU memory reduction of 67.00%, surpassing the second-best one by 17.67 percentage points. This improvement is credited to EMFD, which eliminates the need to sample all L-hop neighbors, requiring only the first $L-k$ layers. This adjustment reduces the space complexity from $\mathcal{O}(Bds_1^L)$ to $\mathcal{O}(Bds_1^{L-k})$. When s_1 is large, even setting $k=2$ can lead to significant reductions in GPU memory.

5.3 Comparison with Decoupled GNNs

Decoupled GNNs have emerged as a popular approach for training scalable GNNs. The core idea is decoupling message propagation from feature transformation, which alleviates the significant memory overhead incurred by storing computational graphs. However, the debate on whether these decoupled models can maintain an expressive power comparable to that of traditional GNNs is still ongoing. Table 3 shows an accuracy comparison between MENSA and decou-

Table 3: F_1 score comparison between MENSA and decoupled GNNs.

	Flickr	**Arxiv**	**Reddit**	**Yelp**	**Amazon**	**Products**
SGC	50.70±0.16	67.62±0.33	94.56±0.03	64.01±0.20	76.32±0.05	74.87±0.19
SIGN	50.94±0.08	69.84±0.17	95.96±0.10	63.12±0.08	77.00±0.18	77.60±0.06
SAGN	50.82±0.25	70.51±0.28	96.48±0.07	63.59±0.16	77.65±0.29	77.46±0.25
GAMLP	52.89±0.13	71.01±0.15	96.66±0.03	64.35±0.10	77.98±0.11	79.35±0.09
HopGNN	53.06±0.09	71.21±0.06	96.53±0.04	64.37±0.14	77.90±0.05	79.64±0.15
LazyGNN	53.04±0.10	71.10±0.30	96.20±0.06	64.25±0.18	78.25±0.07	79.72±0.14
MENSA	**53.48±0.16**	**71.71±0.26**	**96.71±0.03**	**64.49±0.18**	**78.50±0.07**	**80.37±0.06**

Table 4: F_1 score when using only one part of MENSA

Method	**Flickr**	**Arxiv**	**Reddit**	**Yelp**	**Amazon**	**Products**
EMFD	52.74(-0.74)	70.32(-1.39)	95.39(-1.32)	62.99(-1.50)	75.23(-3.27)	77.42(-2.95)
SFAA	52.26(-1.22)	71.30(-0.41)	96.52(-0.19)	64.23(-0.26)	77.87(-0.63)	80.01(-0.36)
MENSA	**53.48**	**71.71**	**96.71**	**64.49**	**78.50**	**80.37**

pled methods, demonstrating that MENSA surpasses all decoupled GNNs and achieves an improvement of 0.33 percentage points over the second-best one.

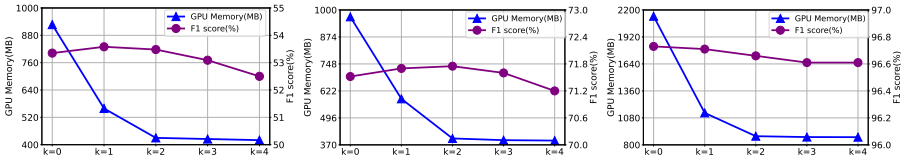
5.4 Ablation and Hyperparameter Study

Impact of EMFD and SFAA. EMFD and SFAA play different roles in MENSA: EMFD expands the neighborhood in shallow layers to pay more attention to close nodes, while SFAA prioritizes the exploration of distant nodes. We explore the effect of two components on accuracy when used independently.

From Table 4, EMFD or SFAA alone achieves lower accuracy than their combination, which is as expected: on one hand, EMFD freezes some layers' parameters, leading to the decrease of model expressive power; on the other hand, SFAA samples very few neighbors for message aggregation, resulting in an insufficient representation of node embedding. MENSA complements both shortcomings by alternately executing them, ultimately achieving better performance.

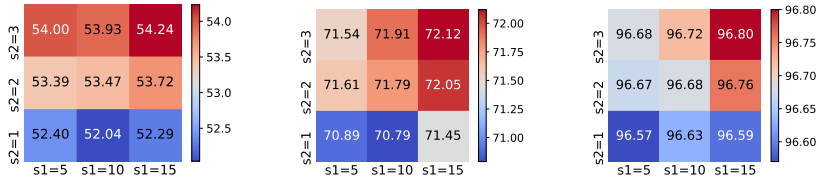
Diverse k Configurations for MENSA. Hyperparameter k determines model's expressive capability and GPU memory consumption. As shown in Figure 4, both F_1 score and memory consumption decrease with k increasing. This is intuitive since more fixed layers reduce the number of trainable parameters, resulting in a decrease in accuracy. Meanwhile, the size of the computational graph retained in the GPU is also reduced, alleviating the burden on GPU memory.

Compared to the reduction in memory, the decrease in F_1 score caused by increasing k is rather trivial. This is consistent with Theorem 1 that for a GNN with L layers, the changes in the parameters of first $(L-1)$ layers are negligible. Freezing these parameters after training several epochs has a minimal impact on the final accuracy. Comparatively, merely increasing k from 0 to 2 can reduce memory by 50%-60% and speed up the whole training process.



(a) Flickr (b) Arxiv (c) Reddit

Fig. 4: F_1 score and memory with different k.



(a) Flickr (b) Arxiv (c) Reddit

Fig. 5: F_1 score under different setting of (s_1, s_2).

Different Settings of s_1 and s_2. s_1 and s_2 are two critical hyperparameters: s_1 determines the width of close neighbors, while s_2 constrains the cost of exploring distant neighbors. Figure 5 shows F_1 scores with different values of (s_1, s_2). Overall, increasing one parameter while keeping the other fixed not only improves accuracy, but also results in increased memory overhead. The improvement in the model's expressive capability is more noticeable when increasing s_2. This phenomenon is intuitive since SFAA benefits from larger s_2 by enabling more accurate gradient estimations. In contrast, solely increasing s_1 fails to update first k layers' parameters, potentially leading to prediction error.

6 Related Work

Many pioneer works trying to address neighbor explosions when training large-scale GNNs, which can be divided into three categories: sampling-based methods, decoupled GNNs, and historical-embedding-based methods.

Sampling-based methods. These methods simulate fullbatch training by sampling a minibatch. They can be classified into three groups: 1) *Node-wise sampling*. GraphSAGE [8] samples only a small number of neighbors for message passing. 2) *Layer-wise sampling*. FastGCN [3] samples nodes independently for each layer and reduces variance by importance sampling. LADIES [25] introduces a layer-specific importance sampler, ensuring the connectivity of selected nodes. LABOR [1] improves LADIES by replacing the original sampler with Poisson sampler. 3) *Subgraph-wise sampling*. ClusterGCN [4] partitions the large graph by METIS algorithm [10], and then applies fullbatch GCN to each small batch. GraphSAINT [22] extracts a large number of small-size induced subgraphs by random walks, while ShaDow [21] samples nodes within K-hop neighbors of root nodes to form subgraphs.

Decoupled GNNs. These approaches decouple feature transformation from message passing in GNN and model them separately. SGC [17] initiates message propagation for K iterations and subsequently employs an MLP for prediction. SIGN [6] transforms each hop's features independently and concatenates them together for the final MLP. SAGN [15] further adopts attention mechanisms to aggregate multi-hop information, while GAMLP [24] utilizes recursive attention and jumping knowledge attention to adaptively capture connections between multi-hop neighbors. HopGNN [2] applies Transformer to learn the association between any two-hop neighbor features. LazyGNN [19] mixes last iteration's output into current iteration's embedding to capture long-distance dependency.

Historical-embedding-based methods. They reuse node embedding from previous epochs to approximate fullbatch training. GNNAutoScale [5] only loads in-minibatch nodes into GPU and retrieves features of out-of-minibatch nodes from a historical embedding cache. After each epoch, it updates cache with new embeddings. GraphFM [20] updates embedding in the cache, not only for in-minibatch nodes but also for their 1-hop neighbors. LMC [14] additionally maintains a historical gradient cache to compensate for lost messages during forward and backward computation.

7 Conclusion

In this paper, we propose a lightweight yet powerful framework MENSA to mitigate neighbor explosion problem. MENSA consists of two strategies EMFD and SFAA. The former freezes the first few layers' parameters and detaches these layers from computational graph. The latter samples much fewer neighbors to construct a small-size computational graph and compensates for EMFD's shortcomings by activating all parameters for message passing. Experiments on real-world graphs show that MENSA significantly reduces time and memory cost, achieving 1.5x-3.5x speedup in convergence speed on most graphs and saving 50%-80% memory compared to GraphSAGE, while achieving a higher F_1 score.

Acknowledgments. This work was substantially supported by Key Projects of the National Natural Science Foundation of China (Grant No. U23A20496).

References

1. Balin, M.F., Catalyurek, U.: Layer-neighbor sampling — defusing neighborhood explosion in gnns. In: Neural Information Processing Systems (2023)
2. Chen, J., Li, Z., Zhu, Y., Zhang, J., Pu, J.: From node interaction to hop interaction: New effective and scalable graph learning paradigm. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (June 2023)
3. Chen, J., Ma, T., Xiao, C.: FastGCN: Fast learning with graph convolutional networks via importance sampling. In: ICLR (2018)
4. Chiang, W.L., Liu, X., Si, S., Li, Y., Bengio, S., Hsieh, C.J.: Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In: Proceedings of the 25th ACM SIGKDD (2019)

5. Fey, M., Lenssen, J.E., Weichert, F., Leskovec, J.: GNNAutoScale: Scalable and expressive graph neural networks via historical embeddings. In: International Conference on Machine Learning (ICML) (2021)
6. Frasca, F., Rossi, E., Eynard, D., Chamberlain, B., Bronstein, M., Monti, F.: Sign: Scalable inception graph neural networks. In: ICML 2020 Workshop on Graph Representation Learning and Beyond (2020)
7. Gilmer, J., Schoenholz, S.S., Riley, P.F., Vinyals, O., Dahl, G.E.: Neural message passing for quantum chemistry. In: ICML (2017)
8. Hamilton, W.L., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. In: Neural Information Processing Systems (2017)
9. Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., Leskovec, J.: Open graph benchmark: Datasets for machine learning on graphs. arXiv preprint arXiv:2005.00687 (2020)
10. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
11. Li, G., Müller, M., Thabet, A., Ghanem, B.: Deepgcns: Can gcns go as deep as cnns? In: The IEEE International Conference on Computer Vision (ICCV) (2019)
12. Lu, K., Yu, Y., Fei, H., Li, X., Yang, Z., Guo, Z., Liang, M., Yin, M., Chua, T.S.: Improving expressive power of spectral graph neural networks with eigenvalue correction. In: AAAI (2024)
13. Lu, K., Yu, Y., Huang, Z., Li, J., Wang, Y., Liang, M., Qin, X., Ren, Y., Chua, T.S., Wang, X.: Addressing heterogeneity and heterophily in graphs: A heterogeneous heterophilic spectral graph neural network. arXiv preprint arXiv:2410.13373 (2024)
14. Shi, Z., Liang, X., Wang, J.: LMC: Fast training of GNNs via subgraph sampling with provable convergence. In: The Eleventh ICLR (2023)
15. Sun, C., Wu, G.: Scalable and adaptive graph neural networks with self-label-enhanced training. arXiv preprint arXiv:2104.09376 (2021)
16. Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., Li, M., Zhou, J., Huang, Q., Ma, C., Huang, Z., Guo, Q., Zhang, H., Lin, H., Zhao, J.J., Li, J., Smola, A., Zhang, Z.: Deep graph library: Towards efficient and scalable deep learning on graphs. arXiv preprint arXiv:1909.01315 (2019)
17. Wu, F., Souza, A., Zhang, T., Fifty, C., Yu, T., Weinberger, K.: Simplifying graph convolutional networks. In: Proceedings of the 36th ICML (2019)
18. Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? In: International Conference on Learning Representations (2019)
19. Xue, R., Han, H., Torkamani, M., Pei, J., Liu, X.: Lazygnn: large-scale graph neural networks via lazy propagation. In: Proceedings of the 40th ICML (2023)
20. Yu, H., Wang, L., Wang, B., Liu, M., Yang, T., Ji, S.: Graphfm: Improving large-scale gnn training via feature momentum. In: ICML (2022)
21. Zeng, H., Zhang, M., Xia, Y., Srivastava, A., Malevich, A., Kannan, R., Prasanna, V., Jin, L., Chen, R.: Decoupling the depth and scope of graph neural networks. In: Neural Information Processing Systems (2021)
22. Zeng, H., Zhou, H., Srivastava, A., Kannan, R., Prasanna, V.: GraphSAINT: Graph sampling based inductive learning method. In: ICLR (2020)
23. Zhang, M., Chen, Y.: Link prediction based on graph neural networks. In: Neural Information Processing Systems (2018)
24. Zhang, W., Yin, Z., Sheng, Z., Li, Y., Ouyang, W., Li, X., Tao, Y., Yang, Z., Cui, B.: Graph attention multi-layer perceptron. In: SIGKDD (2022)
25. Zou, D., Hu, Z., Wang, Y., Jiang, S., Sun, Y., Gu, Q.: Layer-dependent importance sampling for training deep and large graph convolutional networks. In: Neural Information Processing Systems (2019)