A Comprehensive Study on Large-Scale Graph Training: Benchmarking and Rethinking


1 National University of Singapore, 2 Rice University, 3 University of Texas at Austin

{k.duan}@u.nus.edu; {zl105, Kaixiong.Zhou, xia.hu}@rice.edu;
{peihaowang, w.zheng, tianlong.chen, atlaswang}@utexas.edu

Abstract

Large-scale graph training is a notoriously challenging problem for graph neural networks (GNNs). Due to the nature of evolving graph structures into the training process, vanilla GNNs usually fail to scale up, limited by the GPU memory space. Up to now, though numerous scalable GNN architectures have been proposed, we still lack a comprehensive survey and fair benchmark of this reservoir to find the rationale for designing scalable GNNs. To this end, we first systematically formulate the representative methods of large-scale graph training into several branches and further establish a fair and consistent benchmark for them by a greedy hyperparameter searching. In addition, regarding efficiency, we theoretically evaluate the time and space complexity of various branches and empirically compare them w.r.t GPU memory usage, throughput, and convergence. Furthermore, we analyze the pros and cons for various branches of scalable GNNs and then present a new ensembling training manner, named EnGCN, to address the existing issues. Remarkably, our proposed method has achieved new state-of-the-art (SOTA) performance on large-scale datasets. Our code is available at https://github.com/VITA-Group/Large_Scale_GCN_Benchmarking.

1 Introduction

The Graph Neural Networks (GNNs) have shown great prosperity in recent years [1–4], and have dominated a variety of applications, including recommender systems [5–7], social network analysis [8–10], scientific topological structure prediction (e.g., cellular function prediction [11, 12], molecular structure prediction [13, 14], and chemical compound retrieval [15]), and scalable point cloud segmentation [16, 17], etc. Although the message passing (MP) strategy provides GNNs’ superior performance, the nature of evolving massive topological structures prevents MP-based GNNs [18–20, 1, 2, 4, 21, 22] from scaling to industrial-grade graph applications. Specifically, as MP requires nodes aggregating information from their neighbors, the integral graph structures are inevitably preserved during forward and backward propagation, thus occupying considerable running memory and time. For example [6], training a GNN-based recommendation system over 7.5 billion items requires three days on a 16-GPU cluster (384 GB memory in total).

To facilitate understanding, a unified formulation of MP with \(K\) layers is presented as follows:

\[
X^{(K)} = A^{(K-1)} \sigma \left( A^{(K-2)} \sigma \left( \cdots \sigma (A^{(0)}X^{(0)}W^{(0)}) \cdots W^{(K-2)} \right) \right) W^{(K-1)},
\]

where \(\sigma\) is an activation function (e.g., ReLU) and \(A^{(l)}\) is the weighted adjacency matrix at the \(l\)-th layer. As in Eq. (1), the key bottleneck of vanilla MP lies on \(A^{(l)}X^{(l)}\). For the memory usage, the entire sparse adjacency matrix is supposed to be stored in one GPU. As the number of nodes grows, it is quite challenging for a single GPU to afford the message passing over the full graph.
Up to now, massive efforts have been made to mitigate the aforementioned issue and scale up GNNs \cite{23, 24, 3, 25–30}. Most of them focus on approximating the iterative full-batch MP to reduce the memory consumption for training within one single GPU. It is worth noting that we target at the algorithmic scope and do not extend to scalable infrastructure topics like distributed training with multiple GPUs \cite{31, 32} and quantization \cite{33}. Briefly, the previous works encompass two branches: Sampling-based and Decoupling-based. Namely, the former methods \cite{3, 25, 24, 23, 34–36} perform batch training that utilizes sampled subgraphs as a small batch to approximate the full-batch MP so that the memory consumption is considerably reduced. The latter follows the principle of performing propagation \((A^{(l)} X^{(l)})\) and prediction \((X^{(l)} W^{(l)})\) separately, either precomputing the propagation \cite{27, 28, 19, 37, 31} or post-processing with label propagation \cite{29, 38}. Despite the prosperity of scalable GNNs, there are still plights under-explored: we lack a systematic study of the reservoir from the perspective of effectiveness and efficiency, without which it is unachievable to tell the rationale of the designing philosophy for large-scale graph learning in practice.

To this end, we first establish a consistent benchmark and provide a systematic study for large-scale graph training for both Sampling-based methods (Sec. 2.1) and Decoupling-based methods (Sec. 2.2). For each branch, we conduct a thorough investigation of the design strategy and implementation details of typical methods. Then, we carefully examine the sensitive hyperparameters and unify them in one “sweet spot” set by a linear greedy hyperparameter (HP) search (Sec. 3), i.e., iteratively searching the optimal value for an HP while fixing the others. For all concerned methods, the performance comparison is conducted on representative datasets of different scales, varying from about 80,000 nodes to 2, 400, 000, including Flickr \cite{24}, Reddit \cite{3}, and ogbn-products \cite{11}. This step is a crucial precondition on our way to the ultimate as the configuration inconsistency significantly prohibits a fair comparison as well as the following analysis. Besides, regarding efficiency, we theoretically present the time and space complexities for the various branches, and empirically evaluate them on GPU memory usage, throughput, and convergence (Sec. 4). In addition to the benchmark, we further present a new ensembling training manner EnGCN (Sec. 5) to address the existing issues mentioned in our benchmark analysis (Sec. 5.1). Notably, via organically integrating with self-label-enhancement (SLE) \cite{29}, EnGCN achieves the new state-of-the-art (SOTA) on multiple large-scale datasets.

2 Formulations For Large-scale Graph Training Paradigms

2.1 Sampling-based Methods

Given the formulation of Equ. (1), sampling-based paradigm seeks the optimal way to perform batch training, such that each batch will meet the memory constraint of a single GPU for message passing. For completeness, we restate the unified formulation of sampling-based methods as follows:

\[
X_{B_0}^{(l)} = \tilde{A}_{B_1}^{(l-1)} \sigma \left( \tilde{A}_{B_2}^{(l-2)} \sigma \left( \ldots \sigma \left( \tilde{A}_{B_K}^{(0)} X^{(0)} W^{(0)} \right) \ldots \right) W^{(K-2)} \right) W^{(K-1)},
\]

where \(B_l\) is the set of sampled nodes for the \(l\)-th layer, and \(\tilde{A}^{(l)}\) is the adjacency matrix for the \(l\)-th layer sampled from the full graph. Given the local view of GNN — one node’s representation is only related to its neighbors — a straightforward way for unbiased batch training would be \(B_{l+1} = \mathcal{N}(B_l)\), where \(\mathcal{N}\) denotes the set of neighbors. \(B_0\) is randomly sampled according to the uniform distribution. Notably, this batch training style could achieve SOTA performance but also suffers from the “neighbor explosion” problem, where the time consumption and memory usage grow exponentially with the GNN depth, causing significant memory and time overhead. To mitigate this, a number of sampling-based methods were proposed. The key difference among them is how \(\{B_0, \ldots, B_{K-1}, B_K\}\) are sampled. Given a large-scale graph \(\mathcal{G} = (\mathcal{V}, \mathcal{E})\), there are three categories of widely-used sampling strategies:

- **Node-wise Sampling** \cite{3}: \(B_{l+1} = \bigcup_{v \in B_l} \{u \mid u \sim Q : \mathbb{P}_{\mathcal{N}(v)}\}\), where \(\mathbb{P}\) is a sampling distribution; \(\mathcal{N}(v)\) is the sampling space, i.e., the 1-hop neighbors of \(v\); and \(Q\) denotes the number of samples. The representative node-wise sampling method is:

  * **GraphSAGE** \cite{3}: In GraphSAGE, \(\mathbb{P}\) is the uniform distribution.

Compared with the aforementioned naive batch training, the node-wise sampling \cite{3} alleviates the “node explosion” problem by fixing the number of sampled neighbors \(Q\) for each node. It thus reduces the space complexity from \(D^K\) to \(Q^K\), where \(D\) is the averaged node degree. However, as \(Q\) is not far less than \(D\) in order of magnitude, such mitigation is moderate, which is empirically validated by our empirical results in Sec. 3 and Sec. 4.
Layer-wise Sampling [25, 26]. \( B_{t+1} = \{ u \mid u \sim Q \cdot P_{\mathcal{N}(B_t)} \} \), where \( \mathcal{N}(B_t) = \bigcup_{v \in B_t} \mathcal{N}(v) \) denotes the union of 1-hop neighbors of all nodes in \( B_t \). We introduce a couple of layer-wise sampling methods as follows.

* FastGCN [25]: The sampling distribution \( P \) is designed regarding the node degree, where the probability for node \( u \) of being sampled is \( p(u) \propto ||\bar{A}(u,:)||^2 \).

* LADIES [26]: More recently, based on FastGCN, Zou et al. [26] propose LADIES that extends the sampling space from \( \mathcal{N}(B_t) \) to \( \mathcal{N}(B_t) \cup B_t \) by adding the self-loops.

Notably, Compared with the node-wise sampling, the layer-wise sampling essentially solves the "neighbor explosion" problem by fixing the number of overall sampled nodes in a layer to \( Q \). However, the layer-wisely induced adjacency matrix is usually sparser than the others, which accounts for its sub-optimal performance in practice.

Subgraph-wise Sampling [23, 24]. \( B_K = B_{K-1} = \cdots = B_0 = \{ u \mid u \sim Q \cdot P_{\mathcal{G}} \} \). In the subgraph-wise sampling, all layers share the same subgraph induced from the entire graph \( G \) based on a specific sampling strategy \( P_{\mathcal{G}} \), such that the sampled nodes are confined in the subgraph. Typically, this sampling strategy has two representative works:

* ClusterGCN [23]: ClusterGCN first partitions the entire graph into clusters based on some graph partition algorithms, e.g. METIS [39], and then select several clusters to form a batch.

* GraphSAINT [24]: GraphSAINT samples a subset of nodes based on sampling strategy \( P_{\mathcal{G}} \) and then induces the corresponding subgraph as a batch. The commonly-used sampling strategies include: (i) node sampler: \( P(u) = ||\bar{A}_{:,u}||^2 \), (ii) edge sampler: \( P(u,v) = \frac{1}{\deg(u)} + \frac{1}{\deg(v)} \), and (iii) random walk sampler. They are illustrated in Appendix A1.1.

2.2 Decoupling-based Methods

Training GNNs with full-batch message passing at each epoch is not plausible. In this section, we summarize another line of scalable GNNs which decouple the message passing from GPU training to CPUs. Specifically, the message passing is conducted only once at CPUs accompanied by large accessible memory. Depending on the processing order, there are two typical ways to decouple these two operations: (i) pre-processing and (ii) post-processing.

Pre-processing: MP precomputing [27–29]. Recalling Equ. (1), without loss of generalization, we assume that \( A^{(k-1)} = A^{(k-2)} = \cdots = A^{(0)} = A \), i.e. the topological structure for the entire graph remains the same during forward propagation, meeting most of the cases. To decouple the two operations, message passing (\( AX \)) and feature transformation (\( XW \)), we can first pre-compute the propagated node representations and then train a neural network for the downstream task based on these fused representations:

\[
X^l = A^lX, \quad \bar{X} = \rho(X, X^1, \cdots, X^K), \quad Y = f_\theta(\bar{X}),
\]

where \( X^l \) can be regarded as the node representation aggregating \( l \)-hop neighborhood information; \( K \) is the largest propagation hop; \( \rho(\cdot) \) is a function that combines the aggregated features from different hops; and \( f_\theta(\cdot) \) is a feature mapping function parameterized by \( \theta \). We summarize three existing pre-computing schemes as follows.

* SGC [27]: SGC leverages the node representations aggregated with \( k \) hops and feeds the resultant features to a fully-connected layer. We can formulate this scheme by letting \( \rho(\cdot) \) select the last element \( X^K \) and \( f_\theta(\cdot) \) be a linear layer with readout activation: \( Y = \sigma(X^K \Omega) \).

* SIGN [28]: SIGN concatenates features from different hops and then fuse them as the final node representation via a linear layer. To be more specific, \( \rho(\cdot) \) is defined as \( \bar{X} = [X \ X^1 \ \cdots \ X^K] \Omega \), where \( \Omega \) is a transformation matrix, and \( f_\theta(\cdot) \) is defined as a linear readout layer \( Y = \sigma(\bar{X} \Omega) \).

* SAGN [29]: SAGN adopts attention mechanism to combine feature representations from \( K \) hops: \( \bar{X} = \sum_{k=1}^{K} T^kX^k \), where \( T^k \) is a diagonal matrix whose diagonal corresponds to the attention weight for each node of \( k \)-hop information. The attention weight for the \( i \)-th node is calculated by \( T^k_i = \text{softmax}_K(\text{LeakyReLU}(u^T X_i + v^T X^K_i)) \), where the subscripts slice the data matrices along the row. The feature mapping function is implemented by an MLP block with a skip connection to initial features: \( Y = \text{MLP}_\theta(\bar{X} + X \Omega \bar{r}) \).
Post-processing: Label Propagation. The label propagation algorithms [40–44, 38, 45] diffuse labels in the graph and make predictions based on the diffused labels. It is a classical family of graph algorithms for transductive learning, where the nodes for testing are used in the training procedure. The label propagation can be written in a unified form as follows:

\[ Y^{(l)} = \alpha A Y^{(l-1)} + (1-\alpha) G. \]  

(4)

The diffusion procedure iterates the formula above with \( l \) for multiple times to guarantee convergence. It requires two sets of inputs: (i) the stack of the label embeddings of all nodes, denoted as \( Y^{(0)} \in \mathbb{R}^{N \times c} \), where \( c \) is the number of classes. In our implementation, the \( Y^{(0)} \) is the output of a trained MLP model [38]. (ii) the diffusion embedding, denoted as \( G \in \mathbb{R}^{N \times c} \) that propagate themselves across the edges in the graph. Depending on how the diffusion embeddings of unlabeled nodes are computed, two types of \( G \) are summarized as follows:

* **Zeros** [40]: \( G_{i,:} = \begin{cases} \hat{Y}_{i,:} - \alpha A Y_{i,:}, & i \in \mathcal{T}_{train} \\ 0, & \text{otherwise} \end{cases} \), where \( \mathcal{T}_{train} \) denotes the training set and \( \hat{Y} \) is the stack of true labels. For zeros, \( Y^{(0)} = G \).

* **Residual** [38]: \( G_{i,:} = \begin{cases} \hat{Y}_{i,:}, & e_i \in \mathcal{T}_{train} \\ \hat{Z}_{i,:}, & \text{otherwise} \end{cases} \), where \( \hat{Z} = Z + \hat{E} \). \( Z \) is the predictions of a trained simple neural network, e.g. MLP, and \( \hat{E} \) is an residual error matrix, which is optimized iteratively for multiple times by \( E^{t+1} = (1 - \alpha) E + \alpha A E^{(t)} \), where \( E = Z - \hat{Y} \) and \( E^{(0)} = E \).

2.3 More Related Works

Model-agnostic Tricks. Besides the training methods as introduced above, there are some model-agnostic tricks that have been empirically confirmed to be effective for boosting large-scale graph training. Although those add-ons cannot be included into our benchmarking analysis, it is of equal importance to introduce them for completeness. Here we briefly introduce two representative ones:

* **Self-Label-Enhanced (SLE)** [29]: SLE includes two individual tricks, self training and label augmentation. Here we use \( \mathcal{S} \) denoting the training set. For self training, the unlabeled nodes with high confidence (larger than a pre-defined threshold) are added to \( \mathcal{S} \) after a certain number of training epochs. For label augmentation, it trains an additional model \( \Phi(\cdot) \). The forward propagation can be formulated as \( out = \Phi(\hat{A}^k Y_{\mathcal{S}}) \). \( out \) is added to the main model to make the final prediction.

* **GIAN T** [46]: In general, node features are usually pre-embed with graph-agnostic language models, such as word2vec [47] and BERT [38]. Recently, Chien et.al propose a graph-related node feature extraction framework (GIANT), which embeds the raw texts to numerical features by taking advantage of graph structures, to help boost the performance of GNNs for the downstream tasks.

Memory-based GNN Training. Focus on mitigating the “Neighbor Explosion” problem of full-batch training as introduced, memory-based GNNs [49, 50] try to save the GPU memory with different techniques while including all neighbor nodes into computing during the message passing. GAS [49] incorporates historical embeddings [34] to provably maintain the expressive power of full-batch GNN. VQ-GNN [50] utilizes vector quantization to scale convolutional-based GNN and resemble the performance of full-batch message passing by learning an additional quantized feature matrix and a corresponding low-rank adjacent matrix.

3 Benchmarking Over Effectiveness

3.1 Implementation Details

We test numerous large-scale graph training methods with a greedy hyperparameter (HP) search to find their sweet spot and the best performance for a fair comparison. The search space is defined in Table 1. The access and statistics of all used datasets are introduced in Appendix A3.1. Particularly, for label propagation, we select two representative algorithms: Huang et.al. [38], the residual diffusion type, and Zhu et.al. [40], the zeros type. The number of propagation is the maximum iteration \( k \). The aggregation ratio is \( \alpha \) as in Equ. (4), and the number of MLP layers is the number of MLP layers that precedes the label propagation module following Huang et.al. [38].
Limited by space, we select five representative approaches that covers all branches as we introduced, including GraphSAGE [3], LADIES [26], ClusterGCN [23], SAGN [29], and C&S [38]. We illustrate the selected results in Fig. 1 and the results of other methods in Fig. A5. For each subplot, from left to right, each column denotes the search results for one HP. Once one HP was searched, its value will be fixed to the best results for the rest HP searching. Iteratively, we obtain the best performance in the last column. For convenience and clarity, we list the searched optimal hyperparameter settings of all test methods in Table A5.

3.2 Experimental Observations

Table 1: The search space of hyperparameters for benchmarked methods.

<table>
<thead>
<tr>
<th>Category</th>
<th>Hyperparameter (Abbr.)</th>
<th>Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning rate</td>
<td>LR</td>
<td>(1e−2, 1e−3, 1e−4)</td>
</tr>
<tr>
<td>Weight Decay</td>
<td>WD</td>
<td>(1e−4, 1e−5, 1e−6)</td>
</tr>
<tr>
<td>Dropout Rate</td>
<td>DP</td>
<td>(0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7)</td>
</tr>
<tr>
<td>Training Epochs</td>
<td>#E</td>
<td>[20, 30, 40, 50]</td>
</tr>
<tr>
<td>Hidden Dimension</td>
<td>HD</td>
<td>128, 256, 512</td>
</tr>
<tr>
<td># layers (#L)</td>
<td>#L</td>
<td>[2]</td>
</tr>
<tr>
<td>Batch size (BS)</td>
<td>BS</td>
<td>[1000, 2000, 5000]</td>
</tr>
<tr>
<td>Diffusion Type</td>
<td>DT</td>
<td>[residual, zero]</td>
</tr>
<tr>
<td># Propagations</td>
<td>#Prop</td>
<td>[2, 20, 50]</td>
</tr>
<tr>
<td>Aggregation Ratio</td>
<td>AR</td>
<td>[0.5, 0.75, 0.9, 0.99]</td>
</tr>
<tr>
<td>Adj. Norm (Adj)</td>
<td>Adj</td>
<td>[D−1, A, AD, D−1/2AD−1/2]</td>
</tr>
<tr>
<td>Auto Scale (AS)</td>
<td>AS</td>
<td>[True, False]</td>
</tr>
<tr>
<td># MLP Layers (eML)</td>
<td>eML</td>
<td>[2, 3, 4]</td>
</tr>
</tbody>
</table>

Obs. 1. Sampling-based methods are more sensitive to the hyperparameters related to MP. According to Fig. 1, in comparison with precomputing, all sampling-based methods are non-sensitive to hyperparameters (HPs) that are related to the feature transformation matrices, including weight decay, dropout, and hidden dimension; but particularly sensitive to the MP-related HPs, including the number of layers and batch size. For model depth, sampling-based methods generally achieve the sweet spots when the number of layers is confined to shallow and suffers from the oversmoothing problem [31–35] as the GNN models go deeper. However, this issue is moderately mitigated in decoupling-based methods as the model depth does not align with the number of MP hop.

Obs. 2. Sampling-based methods’ performance is nearly positive-correlated with the training batch size. According to the results of the last column of all sampling-based methods, the performance of the layer-wise and subgraph-wise sampling methods is roughly proportional to the batch size. Expectedly, the model performance could further increase as the batch size grows till the upper bound of full-batch training because more links can be preserved. Particularly, in our experiment, we set the number of sampled neighbors of node-wise sampling to a large threshold such that the performance of GraphSAGE can be regarded as full-batch training’s. It can be easily found that the performance of sampling-based methods is inferior to full-batching training (GraphSAGE), further proving our conjecture that the missing links by sampling are non-trivial.

Obs. 3. Precomputing-based methods generally perform better on larger datasets. As show in Fig. 1 and Fig. A5, C&S (label propagation) outperforms the full-batch training (GraphSAGE as introduced in Obs. 2) on the largest dataset ogb-products by a large margin of 4.5%, although both two branches have on-par performance on smaller datasets. Remarkably, our searched results for GraphSAGE and LP on ogb-products also reached better performance, compared with the ones on the OGB leaderboard.

Figure 1: The greedy hyperparameter searching results for selected representative methods. The x-axis denotes the searched HPs, where the abbreviations are consistent with Table 1.

1https://ogb.stanford.edu/docs/leader_nodeprop/
2We rerun it on a GPU with larger memory and the accuracy is 80.56%
error with increasing depth, the observation partially indicates that, limited by model depth and neighboring explosion problem, it is possibly not powerful for extremely large-scale graphs to learn expressive representations.

4 Benchmarking Over Efficiency

4.1 Time And Space Complexity

In this section, we present another benchmark regarding the efficiency of scalable graph training methods. Firstly, we briefly summarize a general complexity analysis in Table 2. For sampling-based methods, we note that the time complexity is for training GNNs by iterating over the whole graph. The time complexity $O(L|A||D + LND^2)$ consists of two parts. The first part $L|A||D$ is from the Sparse-Dense Matrix Multiplication, i.e., $AX$. The second part $LND^2$ is from the normal Dense-Dense Matrix Multiplication, i.e., $(AX)W$. Regarding the space complexity, we need to store the activations of each layer in memory, which has a $O(bLD)$ space complexity. Note that we ignore the memory usage of model weights and the optimizer here since they are negligible compared to the activations. For decoupling-based methods, the training paradigm is simplified as MLPs, and thus the complexity is the same as the traditional mini-batch training. We do not include label propagation in our analysis since it can be trained totally on CPUs.

4.2 Throughput And Memory Usage

Table 2: The time and space complexity for training GNNs with sampling-based and decoupling-based methods, where $b$ is the averaged number of nodes in the sampled subgraph and $r$ is the averaged number of neighbors of each node. Here we do not consider the complexity of pre-processing since it can be done in CPUs.

<table>
<thead>
<tr>
<th>Category</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node-wise Sampling [3]</td>
<td>$O(rLND^2)$</td>
<td>$O(brLD)$</td>
</tr>
<tr>
<td>Layer-wise Sampling [28, 26]</td>
<td>$O(rLND^2)$</td>
<td>$O(brLD)$</td>
</tr>
<tr>
<td>Subgraph-wise Sampling [23, 24]</td>
<td>$O(L</td>
<td>A</td>
</tr>
<tr>
<td>Precomputing [27–29]</td>
<td>$O(LND^2)$</td>
<td>$O(bLD)$</td>
</tr>
</tbody>
</table>

Implementation Details: To fairly benchmark the training speed and memory usage for large-scale graph training methods, we empirically evaluate the throughputs and actual memory for various methods during the training procedure. Here “Throughput” measures how many times can we complete the training steps within a second. Note that we omit the label propagation methods since it is not trained by backward propagation. We provide our implementation details for computing the throughput and memory usage in section A3.2. We report the hardware throughput and activation usage in Table 3. We summarize three main observations.

Table 3: The memory usage of activations and the hardware throughput (higher is better). The hardware here is an RTX 3090 GPU.

<table>
<thead>
<tr>
<th>Category</th>
<th>Flickr</th>
<th>Reddit</th>
<th>ogb-products</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphSAGE</td>
<td>230.63</td>
<td>65.96</td>
<td>687.21</td>
</tr>
<tr>
<td>ClusterGCN</td>
<td>18.45</td>
<td>171.46</td>
<td>79.91</td>
</tr>
<tr>
<td>GraphSAINT</td>
<td>16.51</td>
<td>151.77</td>
<td>21.25</td>
</tr>
<tr>
<td>FastGCN</td>
<td>19.77</td>
<td>226.93</td>
<td>22.53</td>
</tr>
<tr>
<td>LADIES</td>
<td>33.26</td>
<td>195.34</td>
<td>43.21</td>
</tr>
<tr>
<td>SGC</td>
<td>0.01</td>
<td>115.02</td>
<td>0.02</td>
</tr>
<tr>
<td>SIGN</td>
<td>16.99</td>
<td>96.20</td>
<td>16.38</td>
</tr>
<tr>
<td>SAGN</td>
<td>72.94</td>
<td>55.28</td>
<td>72.37</td>
</tr>
</tbody>
</table>

Obs. 4. GraphSAGE is significantly slower and occupies more memory compared to other baselines. This is partially because of the large neighbor sampling threshold we set and inherently owing to its neighborhood explosion. Namely, to compute the loss for a single node, it requires the neighbors’ embeddings at the down-streaming layer recursively. Please refer to Sec. 2.1 for details.

Obs. 5. SGC does not occupy any activation memory. As shown in Table 3, SGC only occupies about 0.01 MB actual memory during training. This is because SGC only has one linear layer and the activation is exactly the input feature matrix, which has been stored in memory. Thus, it is not accounted towards the activation memory.

Obs. 6. In general, the speed of decoupling-based methods is comparable to sampling-based methods. Besides the scale of sparse adjacency matrix, the feature set size is also crucial for...
occupying memory. Although precomputing-based methods avoid storing the graph structures in a GPU, they may take advantage of multi-hop features, where the corresponding memory is multiplied many times.

4.3 Convergence Analysis

![Figure 2: Convergence Analysis](image)

For convergence analysis, we test all benchmarked methods on Flickr, Reddit, and ogbn-products. The training loss and validation accuracy (val_acc) are shown in Fig. 2. Based on the empirical results, we summarize the main observation as follows:

**Obs. 7.** In general, precomputing-based methods have faster and more stable convergence than sampling-based methods. This is because sampling-based methods usually incur a variance among batches that poses unstable and slow convergence \[35\]. However, precomputing-based methods mitigate this by moving message passing from backward propagation to the precomputing stage.

5 EnGCN: Rethinking Graph Convolutional Networks With Ensembling

5.1 An Empirical Summary: Pros And Cons

Based on our benchmark results in section 3 and 4, we summarize the advantages (marked as **Pros**) and constraints (marked as **Cons**) for different branches as follows. Besides the summary, we also provide a joint comparison of effectiveness and efficiency for various methods in Appendix A2.2.

* **Sampling-based:** (**Pros**) Sampling subgraphs into GPU training allows them taking advantage of numerous graph convolution layers, such as GCN [1], GraphSAGE [3], and GIN [4], and this is flexible for them to design specific architecture for different downstream tasks, e.g. node classification and graph classification. (**Cons**) As aforementioned, sampling-based methods suffers from link sparsity \[24\] (section 3) and unstable and low convergence (section 4) problems, both of which prevent sampling-based methods from achieving SOTA performance.

* **Precomputing-based:** (**Pros**) Decoupling message passing from GPU training to CPU precomputing allows precomputing-based methods utilize a mixture of feature transformation units (e.g. attention mechanism and MLP) to train in a well-studied manner. This guarantees stable and fast convergence (section 4). Particularly, integrating with the add-ons like SLE [29] and GIANT [46], precomputing-based methods achieve SOTA performance on large-scale open graph benchmark (ogb) \[11\] datasets. (**Cons**) In general, precomputing-based methods at least occupy a CPU memory space of \(O(LNd)\), where \(L\) is the number of layers; \(N\) is the number of nodes; and \(d\) is the dimension of input features. In comparison, it is \(L\) times as large as the others, which is not affordable for extremely large-scale graphs. For example, containing about 111 million nodes, the largest ogb dataset, ogbn-papers100M, requires approximately 57 Gigabytes (GB) to store the initial feature matrix, given the data type is float and the dimension of features is 128. As the number of layers increases, the required CPU memory space will grow proportionally to an unaffordable number.
\* Label Propagation: \textbf{(Pros)} As a traditional branch of graph learning algorithm, label propagation is a simple but effective add-on as a post-processing trick nowadays. Because of its mode-agnostic nature, it can be simply attached to the end of any graph representation learning algorithm to boost the final prediction. \textbf{(Cons)} Label propagation has many additional sensitive hyperparameters as we introduced in Table 1 and is specifically designed for the node classification task.

5.2 Motivation and Related Works

To address the above constraints of \textit{sampling-based} methods and \textit{precomputing-based} methods, let us first recap the full-batch message passing in Equ. (1). We reformulate it into a more general form:

\[
X^{(k)} = \Phi^{(k-1)}(A \Phi^{(k-2)}(\cdots A \Phi^{(0)}(AX^{(0)}))) ,
\]

where \(\Phi^{(i)}\) denotes the feature mapping model for the \(i\)-th layer. To make the message passing scalable, following the rationale of decoupling, we propose a different training scheme from precomputing: \textit{Instead of end-to-end training, we sequentially train the }\Phi\text{s in a layer-wise manner}. In this way, no precomputing is required and thus the corresponding constraint of CPU memory occupation is essentially mitigated. To elaborate on this, we present the layer-wise training manner:

\[
\begin{align*}
\tilde{X}^{(l)} &= AX^{(l-1)}, & & \text{Message passing on CPUs} \\
\widetilde{Z}^{(l)} &= \Phi^{(l)}(X^{(l)}), & & \text{forward propagation} \\
\n\n\end{align*}
\]

From layer \(0\) to \(k\), we do message passing once and then train \(\Phi^{(l)}\) in batches for epochs. Finally, one can simply use the output of model \(\Phi\) as the prediction. Besides, from the perspective of ensembling, the models \(\Phi\) can be naturally viewed as a set of weak learners trained on multiple views of the input \(X\). As a result, it is compatible to use ensembling to boost the final predictions, such as majority voting. In addition, Based on our empirical results, this training manner is capable of achieving SOTA methods on relatively small datasets without exhaustive finetuning. To further boost the performance, we organically integrate SLE, which \textit{has achieved new SOTA performance on several representative datasets}. We name this model \textit{EnGCN} (Ensembling GCN).

\textbf{Related Works.} Interestingly, the layer-wise training manner and majority voting are naturally consistent with the \textit{boosting} algorithms, where we sequentially train weak learners with instance reweighting and make the final prediction by majority voting. In the scope of graph representation learning, AdaGCN \cite{56} first applies adaboosting \cite{57,58} to address the \textit{oversmoothing} problem of deep GCNs. Though focusing on different topics, AdaGCN has a similar training scheme as ours. Therefore, we implement a scalable version for it, which is included as a SOTA baseline in our experiment. In addition, AdaClusterGCN \cite{59} proposed an adaboosting application that ensembles weak learners trained on different clusters.

5.3 Methodology

Considering a large-scale graph \(G = (A, X, y)\), where \(A\) is the adjacent matrix, \(X\) is the node features, and \(y\) is the true labels. Respectively, \(T_{train}, T_{val}\) and \(T_{test}\) denotes the training, validation, and test set. Let \(X^{(l)}\) and \(Y^{(l)}\) denotes the embeddings of node features and labels at the \(l\)-th layer, respectively. We use \(\hat{y}^{(l)}\) and \(\hat{T}_{train}\) denoting the pseudo labels, pseudo training set for self training at layer \(l\).

\textbf{Initialization.} We initialise several important matrices and vectors:

\[
X^{(0)} = X, \quad Y^{(0)}_{i;i} = \begin{cases} \text{one\_hot}(y_i), & i \in T_{train} \\ 0, & \text{otherwise} \end{cases}, \quad \hat{T}_{train}^{(0)} = T_{train}, \quad \hat{y}^{(0)}_{i} = \begin{cases} y_i, & i \in T_{train} \\ 0, & \text{otherwise} \end{cases}
\]

\textbf{Layer-wise Training.} From 0 to \(k\), we follow a layer-wise training manner, where each training stage contains three phases: \textit{pre-processing}, \textit{training}, and \textit{post-processing}. For layer \(l\), the three phases are introduced as follows.

\textbf{Pre-processing.} For pre-processing, we precompute \(X^{(l)}\) and \(Y^{(l)}\) in CPUs as follows:

\[
X^{(l)} = \hat{A} X^{(l-1)}, \quad Y^{(l)} = \hat{A} Y^{(l-1)},
\]

where \(\hat{A}\) is symmetrically normalized \cite{1}. Note that pre-processing is skipped when \(l = 0\).
Training. We solely train two simple models till convergence, which empirically takes dozens of epochs on real-world datasets. The forward propagation is:

$$o^{(l)} = \Omega(X^{(l)}, Y^{(l)}) = \Phi(X^{(l)}) + \Psi(Y^{(l)}),$$  \hspace{1cm} (7)

where $\Phi$ and $\Psi$ are two MLP models that are shared through all layers. Specifically, when $l = 0$, the forward propagation is reduced to $o^{(0)} = \Phi(X^{(0)})$ where $\Psi$ is not evolved. This is because the initialized $Y^{(0)}$ contains many zero vectors and will pose the overfitting problem. For backward propagation, we compute the training loss using the pseudo labels $\hat{y}^{(l)}$ instead of $y^{(l)}$.

post-processing. After obtaining the trained models, we save the state of them as $\Omega^{(l)} = (\Phi^{(l)}, \Psi^{(l)})$ for ensembling. Furthermore, self training is used to enhance the training set. Following Sun et.al. [29], the pseudo labels and pseudo training masks are updated as follows.

$$\bar{T}_{train}^{t(l+1)} = T_{train}^{t(l)} \cup \{i | \max_c (\tau(o_i^{(l)})) \geq \alpha \}, \quad \hat{y}_i^{(l+1)} = \begin{cases} y_i, & i \in T_{train}^{t(l)} \\ z, & \text{else if } \max_c (\tau(o_i^{(l)})) \geq \alpha \end{cases} ,$$  \hspace{1cm} (8)

where $\tau$ is the softmax function.

Inference With Majority Voting. After k layers, we have obtained a series of weak learners $\{\Omega^{(l)} | 0 \leq l \leq k \}$. The final prediction of node $n$ is made by weighted majority voting [58]:

$$\hat{y}_n = \arg\max_c \sum_{i=0}^k \left( z_{n,i}^{(l)} - \frac{1}{d} \sum_{i=1}^d (o_{n,i}^{(l)}) \right),$$  \hspace{1cm} (9)

where $z^{(l)} = \log\text{softmax}(o^{(l)})$.

5.4 Empirical Analysis

Experiment Settings. Consistent with our effectiveness benchmark, we test our proposed EnGCN on Flickr, Reddit, and ogbn-products. A similar hyperparameter (HP) search was conducted to find its suitable HP setting. The search space is provided in Appendix A2.3. For the baselines, we directly use all benchmark results from section 3, where the SOTA performance has been achieved.

Main Experiment. As shown in Table 4, EnGCN outperforms the sampling-based and decoupling-based methods on multi-scale datasets. For Flickr and Reddit, EnGCN outperforms the baselines by a large margin. Remarkably, EnGCN has achieved new SOTA performance on ogbn-products, outperforming C&S by 2.88% and the SOTA model (GIANT-XRT+SAGN+MCR+C&S) in the ogb leaderboard by 1.26%. In addition to the comparison experiment, we also conduct a couple of ablation studies to provide more insights into EnGCN in Appendix A2.4.

<table>
<thead>
<tr>
<th>Category</th>
<th>Baselines</th>
<th>Flickr</th>
<th>Reddit</th>
<th>ogbn-products</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampling-based</td>
<td>GraphSAGE [3]</td>
<td>53.63 ± 0.13%</td>
<td>96.50 ± 0.03%</td>
<td>80.61 ± 0.16%</td>
</tr>
<tr>
<td></td>
<td>FastGCN [25]</td>
<td>50.51 ± 0.13%</td>
<td>79.50 ± 1.22%</td>
<td>73.46 ± 0.20%</td>
</tr>
<tr>
<td></td>
<td>LADIES [26]</td>
<td>50.51 ± 0.13%</td>
<td>86.96 ± 0.37%</td>
<td>75.31 ± 0.56%</td>
</tr>
<tr>
<td></td>
<td>ClusterGCN [23]</td>
<td>51.20 ± 0.13%</td>
<td>95.68 ± 0.03%</td>
<td>78.62 ± 0.61%</td>
</tr>
<tr>
<td></td>
<td>GraphSAINT [24]</td>
<td>51.81 ± 0.17%</td>
<td>95.62 ± 0.05%</td>
<td>75.36 ± 0.34%</td>
</tr>
<tr>
<td>Decoupling-based</td>
<td>SGC [27]</td>
<td>50.35 ± 0.05%</td>
<td>93.51 ± 0.04%</td>
<td>67.48 ± 0.11%</td>
</tr>
<tr>
<td></td>
<td>SIGN [28]</td>
<td>51.60 ± 0.11%</td>
<td>95.05 ± 0.02%</td>
<td>78.85 ± 0.56%</td>
</tr>
<tr>
<td></td>
<td>SAGN [29]</td>
<td>50.07 ± 0.11%</td>
<td>96.48 ± 0.03%</td>
<td>81.21 ± 0.07%</td>
</tr>
<tr>
<td></td>
<td>GAMLP [30]</td>
<td>52.58 ± 0.12%</td>
<td>96.73 ± 0.03%</td>
<td>83.76 ± 0.19%</td>
</tr>
<tr>
<td></td>
<td>C&amp;S [35]</td>
<td>51.24 ± 0.17%</td>
<td>93.33 ± 0.08%</td>
<td>85.11 ± 0.07%</td>
</tr>
<tr>
<td>Other SOTA Methods</td>
<td>AdaGCN [56]</td>
<td>52.97 ± 0.01%</td>
<td>96.05 ± 0.00%</td>
<td>76.41 ± 0.00%</td>
</tr>
<tr>
<td></td>
<td>SAGN+SLE [29]</td>
<td>54.60 ± 0.40%</td>
<td>97.10 ± 0.00%</td>
<td>84.28 ± 0.14%</td>
</tr>
<tr>
<td></td>
<td>GIANT-XRT+</td>
<td>-</td>
<td>-</td>
<td>86.73 ± 0.08%</td>
</tr>
<tr>
<td></td>
<td>SAGN+MCR+C&amp;S [60]</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Ours</td>
<td>EnGCN</td>
<td><strong>56.43 ± 0.21%</strong></td>
<td><strong>97.14 ± 0.03%</strong></td>
<td><strong>87.99 ± 0.04%</strong></td>
</tr>
</tbody>
</table>

*: the results are from the original papers

The Training Efficiency and Convergence Landscape of EnGCN. For EnGCN, since all we need to train is two simple shallow MLPs (Section 5.3), the GPU throughput and memory consumption...
are expected to be sufficiently efficient. The remained concern is solely about the convergence of EnGCN. Due to the nature of layer-wise training, the convergence of EnGCN is more complicated than other end-to-end training methods. In Figure 3, we show the convergence landscape of EnGCN and provide several interesting observations as follows.

1. As shown in Figure 3, the train accuracy and validation accuracy generally increase layer-wisely till convergence. Noticeably, though the training accuracy occasionally drops, the validation accuracy still relatively remains positive. 2. At the beginning of each layer, the accuracy changes rapidly, indicating the remarkable distribution difference for various hops. 3. Different datasets are sensitive to different hops. For example, the 2-nd hop is crucial to boost the training and validation accuracy on Flickr, while for Reddit and ogbn-products, 1-hop neighbors are more important.

Figure 3: The convergence landscape of EnGCN. All models are trained with 4 layers’ features. For each layer-wise phase, we train the model with 70 epochs.

The CPU memory consumption of EnGCN. To confirm the low CPU memory consumption of EnGCN, we provide a comparison experiment and illustrate the results in Figure 4. The x-axis denotes the models’ number of layers while the y-axis records their allocated memories that are reported by “aten::empty” of PyTorch. As shown in Figure 4, the precomputing-based methods, SIGN and SAGN, suffer from expensive CPU memory consumption as the model depth increases. For sampling-based methods, since there is no need to pre-store a large number of feature matrices, the memory consumption increases much more smoothly. For EnGCN, as no precomputing is required, the CPU memory consumption is considerably reduced in comparison with SIGN and SAGN, which intuitively validates the CPU memory efficiency of EnGCN.

6 Conclusion

The scalability issue of graph convolutional networks has been a notoriously challenging research problem. In this work, we establish a fair and consistent benchmark for large-scale graph training w.r.t effectiveness and efficiency. We provide a unified formulation for dozens of works and further assess them on the basis of accuracy, memory usage, throughput, and convergence. Furthermore, provided with the comprehensive benchmark results, we rethink the scalability issue of GCNs from the perspective of ensembling and then present an ensembling-based trainer scheme (EnGCN) that solely needs to train a couple of simple MLPs to achieve new SOTA on multi-scale large datasets. We hope our study on benchmarking and rethinking to help lay a solid, practical, and systematic foundation for the scalable GCN community and provide researchers with broader and deeper insights into large-scale graph training.
References


