Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Empirical Analysis of Performance Bottlenecks in Graph Neural Network Training with GPUs

Zhaokang Wang, Yunpan Wang, Chunfeng Yuan, Rong Gu*, Yihua Huang*

State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Jiangsu, China

Correspondence
Rong Gu* and Yihua Huang* with equal contribution.
Email: {gurong, yhuang}@nju.edu.cn

Present Address
No.163 Xianlin Avenue, Nanjing 210023, Jiangsu, China

Summary

The graph neural network (GNN) has become a popular research area for its state-of-the-art performance in many graph analysis tasks. Recently, various graph neural network libraries have emerged.They make the development of GNNs convenient, but their performance on large datasets is not satisfying. In this work, we analyze the performance bottleneck in training GNN with GPUs empirically. A GNN layer can be decomposed into two parts: the vertex and the edge calculation parts. According to their computational complexity, we select four representative GNNs (GCN, GGNN, GAT, GaAN) for evaluation. We breakdown their training time and memory usage, evaluate the effects of hyper-parameters, and assess the efficiency of the sampling techniques. The experimental evaluation indicates that the edge-related calculation is the performance bottleneck for most GNNs, dominating the training time and memory usage. Future optimization can focus on it. The sampling techniques are essential for training big graphs on GPUs, but their current implementations still have room for improvement.

KEYWORDS:
graph neural network, performance bottleneck analysis, empirical evaluation, machine learning system, GPU

1 | INTRODUCTION

In recent years, the graph neural network (GNN) becomes a hot research topic in the field of artificial intelligence. Many GNNs [1,2,3,4,5,6,7] are proposed. They can learn the representation of vertices/edges in a graph from its topology and the original feature vectors in an *end-to-end* manner. The powerful expression ability of GNNs makes them achieve good accuracy in many graph analysis tasks [8,9,10], like node classification, link prediction, and graph classification.

To train GNNs easily, a series of GNN libraries/systems [11,12,13,14,15] are proposed. PyG [11], NeuGraph [13], PGL [15] and DGL [12] build upon the existing deep learning frameworks (PyG on PyTorch, NeuGraph on TensorFlow, PGL on PaddlePaddle, DGL on multiple backends). They provide users with a high-level programming models (the message-passing model for PyG/PGL/DGL and the SAGA-NN model for NeuGraph) to describe the structure of a GNN. They take advantage of the common tools provided by the underlying frameworks like the automatic differentiation to simplify the development. They utilize specially optimized CUDA kernels (like kernel fusion [12 13]) and other implementation techniques (like 2D graph partitioning [13]) to improve the speed of GNN training on GPUs.

However, what is the real performance bottleneck in the GNN training is still in doubt. Yan et al.[16] and Zhang et al.[17] experimentally analyze the architectural characteristics of the GNN *inference*. They find that the GNN inference is more cache-friendly than the traditional graph analysis tasks(like PageRank) and is suitable for GPUs. They verify the effectiveness of the kernel fusion optimization in reducing the time of inference. Nevertheless, they only analyze the inference stage, ignoring the effects of the backpropagation during the training.

To explore the essential performance bottleneck in the GNN training, we conduct a range of experimental analysis in deep in this work. We focus on the efficiency bottleneck of GNN training instead of accuracy. We model the GNNs with the message-passing framework that decomposes a GNN layer into two parts: the vertex calculation and the edge calculation. According to the time complexity of the two parts, we classify the typical GNNs into four quadrants({high, low} complexity × {vertex, edge} calculation). We choose GCN[1], GGNN[4], GAT[6] and GaAN[7] as representative GNNs of the four quadrants.

We implement them with PyG and evaluate their efficiency with six real-world datasets on a GPU. We identify the most time-consuming stage in the GNN training by decomposing the training time per epoch from the layer level to the operator level. We also analyze the memory usage during the training to discover the main factor that limits the data scalability of GNN training on GPUs. Finally, we evaluate whether or not the sampling techniques affect the performance bottleneck. The key findings and insights are summarized below.

- **The training time and the memory usage of a GNN layer is mainly affected by the dimensions of the input/output hidden feature vectors.** Fixing other hyper-parameters, the training time and the memory usage of a GNN layer increase linearly as the input/output dimensions.
- **The edge-related calculation is the performance bottleneck for most GNNs.** For GNNs with high edge calculation complexity, most of the training time is spent on conducting the messaging function for every edge. For GNNs with low edge calculation complexity, the message collection and aggregation consumes most of the training time.
- **The high memory usage of the edge calculation is the main factor limiting the data scalability of GNN training.** The edge calculation generates and caches many intermediate results. They are an order of magnitude larger than the dataset itself. As GPUs have limited on-chip memory, the high memory consumption prevents us from training big graphs on GNNs.
- **The sampling techniques can significantly reduce training time and memory usage.** They are essential for training big graphs on GPUs. However, the existing implementation is still inefficient. Under big batch sizes, the time spent on the sampling may exceed the time spent on the training. Under small batch sizes, the sampled graphs are also small, wasting the computing power of GPUs.

Based on the insights, we provide several potential optimization directions:

- To reduce training time, **optimizations should focus on improving the efficiency of the edge calculation**. One may consider developing optimized operators for the messaging step that is the major source of computing costs in the edge calculation. Fusing operators of the collection step, messaging function and the aggregation step together is another way to reduce the overheads in the edge calculation.
- To reduce memory usage, **optimizations should focus on reducing the intermediate results in the edge calculation**. One may consider adopting the checkpoint mechanism to cache less intermediate results during the forward phase and re-calculate the needed data on the fly during the backpropagation.
- To improve the efficiency of the sampling techniques, **one may consider overlapping the sampling on the CPU side with the training on the GPU side.** Choosing a proper batch size automatically is another potential optimization.

We hope that our analysis can help the developers of the GNN libraries/systems have a better understanding of the characteristics of GNN training and propose more targeted optimizations.

Outline

We briefly survey the typical GNNs in Section 2. We introduce our experimental setting and targets in Section 3. The experimental results are presented and analyzed in Section 4. We summarize the key findings and give out potential optimization directions in Section 5. We introduce the related work in Section 6 and finally conclude our work in Section 7.

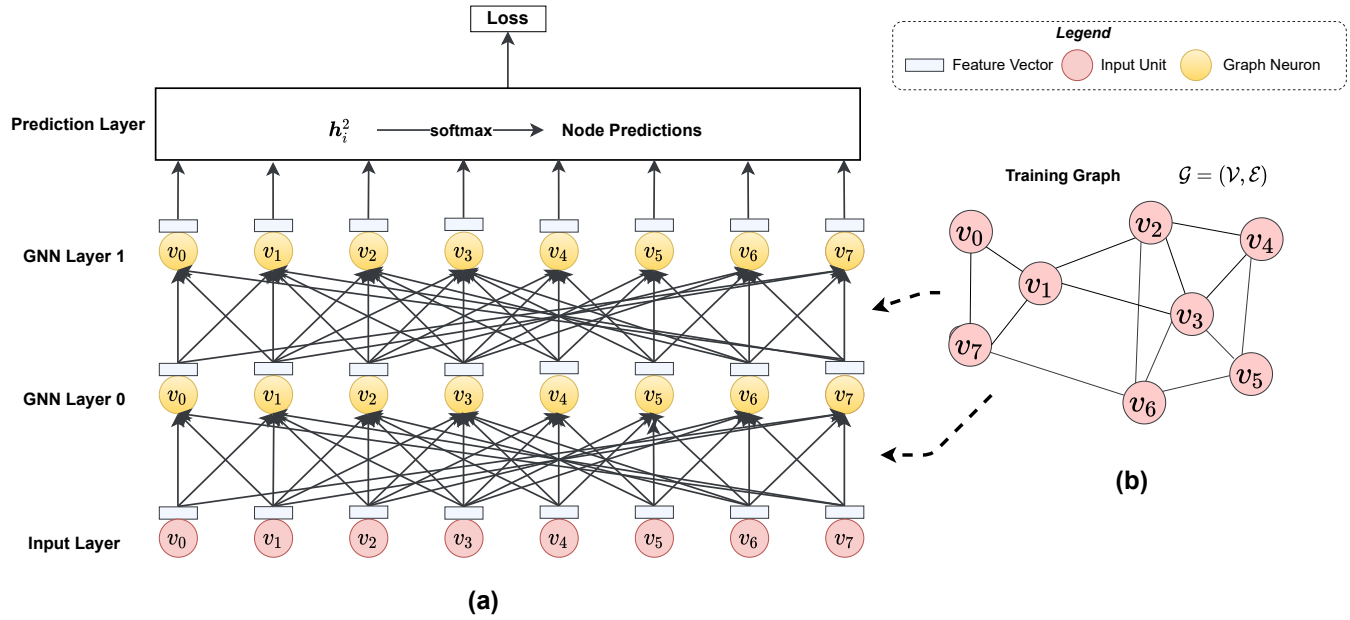


FIGURE 1 Structure of a typical graph neural network. (a) Demo GNN, (b) Demo graph. The target application is the node classification. The demo GNN has two GNN layers.

2 | REVIEW OF GRAPH NEURAL NETWORKS

In this section, we formally define the graph neural networks and briefly survey typical graph neural networks. We denote a simple graph $\mathcal{G}$ as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ and $\mathcal{E}$ are the vertex set and the edge set of $\mathcal{G}$, respectively. Let $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$ as the number of vertices/edges. We use v_i $(0 \leq i < n)$ to denote a vertex and $e_{i,j} = (v_i, v_j)$ to denote the edge pointing from v_i to v_j. The adjacency set of v_i is $\mathcal{N}(v_i) = \{v | (v_i, v) \in \mathcal{E}\}$. We denote a *vector* with a bold lower case letter like $\boldsymbol{x}$ and a *matrix* with a bold upper case letter like $\boldsymbol{X}$.

2.1 | Structure of Graph Neural Networks

As illustrated in **FIGURE** 1, a typical GNN can be decomposed into three parts: an input layer + several GNN layers + a prediction layer.

A GNN receives a graph $\mathcal{G}$ as the input. Every vertex v_i in $\mathcal{G}$ is attached with a feature vector $\boldsymbol{x}_i$ to describe the properties of the vertex. The edges of $\mathcal{G}$ may also be attached with feature vectors $\boldsymbol{e}_{i,j}$ The input layer of a GNN receives feature vectors from all vertices and passes them to GNN layers.

A GNN layer consists of n graph neurons, where n is the number of vertices in $\mathcal{G}$. Each graph neuron corresponds to a vertex in $\mathcal{G}$. In the first GNN layer (Layer 0), the graph neuron of the vertex v_i collects input feature vectors of itself and the vertices $\boldsymbol{x}_j$ that are adjacent to v_i in $\mathcal{G}$ (i.e., $v_j \in \mathcal{N}(v_i)$) from the input layer. The graph neuron aggregates input feature vectors, apply non-linear transformation, and outputs a hidden feature vector $\boldsymbol{h}_i^1$ for v_i. Take the demo GNN in **FIGURE** 1(a) as the example. Since $\mathcal{N}(v_3) = \{v_1, v_2, v_4, v_5, v_6\}$, the graph neuron of v_3 at the GNN layer 0 collects the feature vectors $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4, \boldsymbol{x}_5, \boldsymbol{x}_6\}$ from the input layer and outputs $\boldsymbol{h}_3^1$. Different GNNs mainly differ in the graph neurons that they use. We elaborate on their details later.

Different from the traditional neural networks that layers are fully connected, the layers are sparsely connected in GNNs. The connection between the input layer and the first GNN layer is determined by the topology of $\mathcal{G}$. Two graph neurons are connected only if their corresponding vertices have an edge between them in $\mathcal{G}$. Most real-world graphs are very *sparse*, i.e. $|\mathcal{E}| \ll |\mathcal{V}|^2$.

In the next GNN layer (GNN Layer 1), the graph neuron of v_i collects the hidden feature vectors of itself $\boldsymbol{h}_i^1$ and its neighbors ($\boldsymbol{h}_j^1$ with $v_j \in \mathcal{N}(v_i)$) from the *previous* GNN layer. Based on the collected hidden vectors, the graph neuron in Layer 1 outputs a new hidden feature vector $\boldsymbol{h}_i^2$ for v_i. A GNN allows stacking more GNN layers to support deeper graph analysis. Assume there are L GNN layers in traditional. The last GNN layer (GNN Layer $L-1$) outputs a hidden feature vector $\boldsymbol{h}_i^L$ for every vertex v_i.

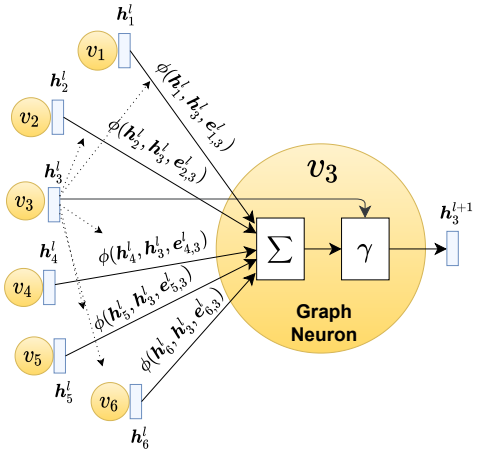


FIGURE 2 Graph neuron of v_3 at the GNN layer l with the graph $\mathcal{G}$ in **FIGURE** 1b. ϕ/Σ/γ are the messaging/aggregation/vertex updating functions in the message-passing model, respecitvely.

As an embedding vector, $\boldsymbol{h}_i^L$ encodes the knowledge learned from the input layer and all the previous GNN layers. Since $\boldsymbol{h}_i^L$ is affected by v_i and the vertices in the L-hop neighborhood of v_i, analyzing a graph with a *deeper* GNN means analyzing each vertex with a *wider* scope.

The hidden feature vectors $\boldsymbol{h}_i^L$ of the last GNN layer are fed to the prediction layer to generate the output of the whole GNN. The prediction layer is a standard neural network. The structure of the prediction layer depends on the prediction task of the GNN. Take the node classification task in **FIGURE** 1 as the example. The node classification predicts a label for every vertex in $\mathcal{G}$. In this case, the prediction layer can be a simple softmax layer with $\boldsymbol{h}_i^L$ as the input and a vector of probabilities as the output. If the prediction task is edge prediction, the hidden feature vectors of two vertices are concatenated and fed into a softmax layer. If we need to predict a label for the whole graph, a pooling (max/mean/...) layer is added to generate an embedding vector for the whole graph and the embedding vector is used to produce the final prediction.

Supporting end-to-end training is a prominent advantage of GNN, compared with traditional graph-based machine learning methods. The traditional methods need to construct feature vectors manually or from the embedding methods like DeepWalk [18] and node2vec [19]. The feature vector generation is independent from the downstream prediction task. Thus, the vectors have to be adjusted for every prediction task individually. In GNNs, the gradients are propagated from the prediction layer back to the previous GNN layers layer by layer. The model parameters in the GNN layers are updated based on the feedbacks from the downstream prediction task. In a fully parameterized way, the GNN automatically extracts an embedding vector for each vertex from its L-hop neighborhood, tuned according to the specific prediction task.

2.2 | Graph Neuron and Message-passing Model

Graph neurons are building blocks of a GNN. A GNN layer consists of $|\mathcal{V}|$ graph neurons. Each vertex corresponds to a graph neuron. A graph neuron as shown in **FIGURE** 2 is a small neural network. The graph neuron of v_i at layer l receives hidden feature vectors $\boldsymbol{h}_j^l$ from the graph neurons of v_i and its neighbors ($v_j \in \{v_i\} \cup \mathcal{N}(v_i)$) at the previous GNN layer [†]. The graph neuron aggregates the received hidden feature vectors, applies non-linear transformations, and outputs a new hidden feature vector $\boldsymbol{h}_i^{l+1}$.

[†]For the GNN layer 0, graph neurons receive input feature vectors, i.e., $\boldsymbol{h}_i^0 = \boldsymbol{x}_i$

We follow the message-passing model[20] to formally define a graph neuron. The message-passing model is widely used in the cutting-edge GNN libraries like PyTorch Geometric (PyG)[11] and Deep Graph Library (DGL)[12]. **FIGURE** 2 shows the structure of a graph neuron in the message-passing model. Graph neurons at layer l are made of three *differentiable* functions: ϕ^l, Σ^l and γ^l. The graph neuron calculates the output hidden vector $\boldsymbol{h}_i^{l+1}$ by

$$\boldsymbol{h}_i^{l+1} = \gamma^l(\boldsymbol{h}_i^l, \Sigma^l_{v_j \in \mathcal{N}(v_i)} \phi^l(\boldsymbol{h}_i^l, \boldsymbol{h}_j^l, \boldsymbol{e}_{j,i})).$$

ϕ^l is the *messaging* function. For every incident edge (v_j, v_i) of v_i, ϕ receives the output hidden feature vectors $\boldsymbol{h}_i^l$ and $\boldsymbol{h}_j^l$ from the previous GNN layer and the edge feature vector $\boldsymbol{e}_{j,i}$ as the input. ϕ^l emits a message vector $\boldsymbol{m}_{j,i}^l$ for every edge (v_j, v_i) at layer l, i.e., $\boldsymbol{m}_{j,i}^l = \phi^l(\boldsymbol{h}_i^l, \boldsymbol{h}_j^l, \boldsymbol{e}_{j,i})$. For every v_i, the message vectors $\boldsymbol{m}_{j,i}^l$ of its adjacent edges ($v_j \in \mathcal{N}(v_i)$) are aggregated by the *aggregation* function Σ^l to produce an aggregated vector $\boldsymbol{s}_i^l$, i.e., $\boldsymbol{s}_i^l = \Sigma^l_{v_j \in \mathcal{N}(v_i)} \boldsymbol{m}_{j,i}^l$. v_i's aggregated vector $\boldsymbol{s}_i^l$ and its hidden vector $\boldsymbol{h}_i^l$ from the previous GNN layer are fed into the *vertex updating* function γ^l to calculate the output hidden vector $\boldsymbol{h}_i^{l+1}$ of the current layer l, i.e., $\boldsymbol{h}_i^{l+1} = \gamma^l(\boldsymbol{h}_i^l, \boldsymbol{s}_i^l)$ The end-to-end training requires ϕ^l and γ^l (like multi-layer perceptrons and GRU) and Σ_l (like mean, sum, element-wise min/max) are *differentiable* to make the whole GNN differentiable.

Different GNNs have different definitions of the three functions. We regard ϕ and Σ as the *edge calculation* functions, since they are conducted over every edge in $\mathcal{G}$. We regard γ as the *vertex calculation* function, as it is conducted over every vertex in $\mathcal{G}$. **TABLE** 1 and **TABLE** 2 list the edge functions and the vertex functions of typical GNNs, respectively. For ChebNet, we report its GNN sub-layer in the tables. A ChebNet layer consists of K GNN sub-layers and a summation layer: $\boldsymbol{H}^{l+1} = \sum_{k=1}^{K} \boldsymbol{Z}^{(k)} \boldsymbol{W}^{(k)}$ with the GNN sub-layers $\boldsymbol{Z}^{(1)} = \boldsymbol{H}^l$, $\boldsymbol{Z}^{(2)} = \hat{\boldsymbol{L}}\boldsymbol{H}^l$, and $\boldsymbol{Z}^{(k)} = 2\hat{\boldsymbol{L}}\boldsymbol{Z}^{(k-1)} - \boldsymbol{Z}^{(k-2)}$, where $\boldsymbol{H}^l$ is the matrix of output hidden feature vectors of the layer $l-1$. For GAT, a GAT layer consists of two sub-layers and it conducts part of the vertex calculation before the two sub-layers. For GaAN, a GaAN layer consists of four sub-layers: the first sub-layer calculates the summation $\boldsymbol{m}_{j,i,(k)}^{l,0} = \exp((\boldsymbol{W}_{xa}^{l,k}\boldsymbol{h}_j^l + \boldsymbol{b}_{xa}^{l,k})^T(\boldsymbol{W}_{ya}^{l,k}\boldsymbol{h}_i^l + \boldsymbol{b}_{ya}^{l,k}))$, and the other three sub-layers calculate $\boldsymbol{m}_{j,i,1}^l / \boldsymbol{m}_{j,i,2}^l / \boldsymbol{m}_{j,i,3}^l$.

2.3 | Classification of GNNs

Since we focus on analyzing the performance bottleneck in training GNNs, we classify the typical GNNs from the view of time complexity. We use $O_\phi / O_\Sigma / O_\gamma$ to denote the time complexity of the three functions in the message-passing model. The time complexity of a GNN layer is made up of two parts: the edge calculation complexity $m * (O_\phi + O_\Sigma)$ and the vertex calculation complexity $n * O_\gamma$.

In **TABLE** 1 and **TABLE** 2, we list the edge and vertex calculation complexity, respectively. The time complexity of a graph neuron is affected by the dimensions of the input/output hidden vectors d_{in} and d_{out} and the dimensions of the model parameters (like the number of heads K in GAT and the dimensions of the view vectors $d_a / d_v / d_m$ in GaAN).

We classify the typical GNNs into four quadrants based on their edge/vertex complexity as shown in **FIGURE** 3. We pick GCN, GGNN, GAT, and GaAN as the *representative GNNs* of the four quadrants.

GCN (low vertex & edge calculation complexity): Graph convolution network (GCN[1]) introduces the first-order approximation of the spectral-based graph convolutions. It has only one parameter to learn at each layer, i.e. the weight matrix $\boldsymbol{W}^l$ in γ. A GCN graph neuron can be expressed as $\boldsymbol{h}_i^{l+1} = \boldsymbol{W}^l \sum_{v_j \in \mathcal{N}(v_i)} e_{j,i} \boldsymbol{h}_j^l$, where $e_{j,i}$ is the normalized weight of the edge (v_j, v_i). According to the associative law of the matrix multiplication, $\boldsymbol{h}_i^{l+1} = \sum_{v_j \in \mathcal{N}(v_i)} e_{j,i} \boldsymbol{W}^l \boldsymbol{h}_j^l$. Since the dimension of $\boldsymbol{h}_i^{l+1}$ is usually smaller than $\boldsymbol{h}_i^l$ in practical GCNs, the implementation of GCN in PyTorch Geometric chooses to first conduct the vertex calculation $\hat{\boldsymbol{h}}_j^l = \boldsymbol{W}^l \boldsymbol{h}_j^l$ for each vertex v_j and then conduct the edge calculation $\boldsymbol{h}_i^{l+1} = \sum_{v_j \in \mathcal{N}(v_i)} e_{j,i} \hat{\boldsymbol{h}}_j^l$. As $\hat{\boldsymbol{h}}_j^l$ has the same dimension as $\boldsymbol{h}_i^{l+1}$, the implementation significantly reduces the computation cost of the edge calculation.

GGNN (high vertex & low edge calculation complexity): GGNN[4] introduces the gated recurrent unit (GRU) into the graph neural networks. The vertex updating function ϕ of GGNN is a modified GRU unit that has 12 model parameters to learn, having high computational complexity. To lower the training cost, all GNN layers share the same group of parameters in GGNN. GGNN further requires the dimension of $\boldsymbol{h}^{l+1}$ is equal to the dimension of $\boldsymbol{h}^l$. Since the messaging function ϕ only uses the hidden feature vector $\boldsymbol{h}_j^l$ of the source vertex v_j of an edge (v_j, v_i). In the implementation of PyG, GGNN conducts the pre-processing vertex calculation $\hat{\boldsymbol{h}}_i^l = \boldsymbol{W}\boldsymbol{h}_i^l$ for every vertex v_i before the message-passing. The messaging function ϕ directly uses $\hat{\boldsymbol{h}}_j^l$ as the message vector for the edge (v_j, v_i). In this way, GGNN further reduces the time complexity of the edge calculation to $O(1)$ without increasing the time complexity of the vertex calculation.

GAT (low vertex & high edge calculation complexity): GAT[6] introduces the attention and multi-head mechanism into the graph neural networks. The K heads generate K independent views for an edge, where K is a hyper-parameter. The views of K

GNN	Type	Σ	ϕ	Complexity
ChebNet [2]	Spectral	sum	$\boldsymbol{m}_{j,i}^{(k)} = e_{j,i}\boldsymbol{z}_j^{(k-1)}$	$O(d_{in})$
GCN [1]	Spectral	sum	$\boldsymbol{m}_{j,i}^{l} = e_{j,i}\boldsymbol{h}_j^{l}$	$O(d_{in})$
AGCN [3]	Spectral	sum	$\boldsymbol{m}_{j,i}^{l} = \vec{e}_{j,i}^{\,l}\boldsymbol{h}_j^{l}$	$O(d_{in})$
GraphSAGE [5]	Non-spectral	mean/LSTM	$\boldsymbol{m}_{j,i}^{l} = \boldsymbol{h}_j^{l}$	$O(1)$
GraphSAGE-pool [5]	Non-spectral	max	$\boldsymbol{m}_{j,i}^{l} = \delta(\boldsymbol{W}_{pool}^{l}\boldsymbol{h}_j^{l} + b^{l})$	$O(d_{in} * d_{out})$
Neural FPs [21]	Non-spectral	sum	$\boldsymbol{m}_{j,i}^{l} = \boldsymbol{h}_j^{l}$	$O(1)$
SSE [22]	Recurrent	sum	$\boldsymbol{m}_{j,i}^{l} = [\boldsymbol{x}_j \parallel \boldsymbol{h}_j^{l}]$	$O(f + d_{in})$
GGNN [4]	Gated	sum	$\boldsymbol{m}_{j,i}^{l} = \boldsymbol{W}^{l}\boldsymbol{h}_j^{l}$	$O(d_{in} * d_{out})$
GAT [6]	Attention	sum	Sub-layer 0:	concat: $O(d_{out})$
			$\boldsymbol{m}_{j,i,(k)}^{l,0} = \exp(LeakyReLU(\boldsymbol{a}^{T}[\hat{\boldsymbol{h}}_{i,(k)}^{l} \parallel \hat{\boldsymbol{h}}_{j,(k)}^{l}]))$	average: $O(K * d_{out})$
			Sub-layer 1:	Two sub-layers
			$\alpha_{j,i,(k)}^{l} = \frac{\exp(LeakyReLU(\boldsymbol{a}^{T}[\hat{\boldsymbol{h}}_{i,(k)}^{l} \parallel \hat{\boldsymbol{h}}_{j,(k)}^{l}]))}{\hat{\boldsymbol{a}}_{i,(k)}^{l,0}}$	
			Multi-head concatenation : $\boldsymbol{m}_{j,i}^{l,1} = \parallel_{k=1}^{K} \alpha_{j,i,(k)}^{l}\hat{\boldsymbol{h}}_{j,(k)}^{l}$	
			Multi-head average : $\boldsymbol{m}_{j,i}^{l,1} = \frac{1}{K}\sum_{k=1}^{K}\alpha_{j,i,(k)}^{l}\hat{\boldsymbol{h}}_{j,(k)}^{l}$	
GaAN [7]	Attention	sum,max,mean	Sub-layer 0:	$O(max(d_a, d_m, d_v) * K * d_{in})$
			$\boldsymbol{m}_{j,i,(k)}^{l,0} = \exp((\boldsymbol{W}_{xa}^{l,k}\boldsymbol{h}_j^{l} + b_{xa}^{l,k})^{T}(\boldsymbol{W}_{ya}^{l,k}\boldsymbol{h}_i^{l} + \boldsymbol{b}_{ya}^{l,k}))$	Four sub-layers
			Sub-layer 1:	
			$\alpha_{j,i,(k)}^{l} = \frac{\exp((\boldsymbol{W}_{xa}^{l,k}\boldsymbol{h}_j^{l} + b_{xa}^{l,k})^{T}(\boldsymbol{W}_{ya}^{l,k}\boldsymbol{h}_i^{l} + \boldsymbol{b}_{ya}^{l,k}))}{\hat{\boldsymbol{a}}_{i,(k)}^{l,0}}$	
			$\boldsymbol{m}_{j,i}^{l,1} = \parallel_{k=1}^{K} \alpha_{j,i,(k)} LeakyReLU(\boldsymbol{W}_{v}^{l,k}\boldsymbol{h}_j^{l} + \boldsymbol{b}_{v}^{l,k})$	
			Sub-layer 2:	
			$\boldsymbol{m}_{j,i}^{l,2} = \boldsymbol{W}_{m}^{l}\boldsymbol{h}_j^{l} + \boldsymbol{b}_{m}^{l}$	
			Sub-layer 3:	
			$\boldsymbol{m}_{j,i}^{l,3} = \boldsymbol{h}_j^{l}$	

TABLE 1 Typical graph neural networks and their edge calculation functions. d_{in} and d_{out} are dimensions of the input and output hidden feature vectors, respectively. Blue variables are model parameters to learn.

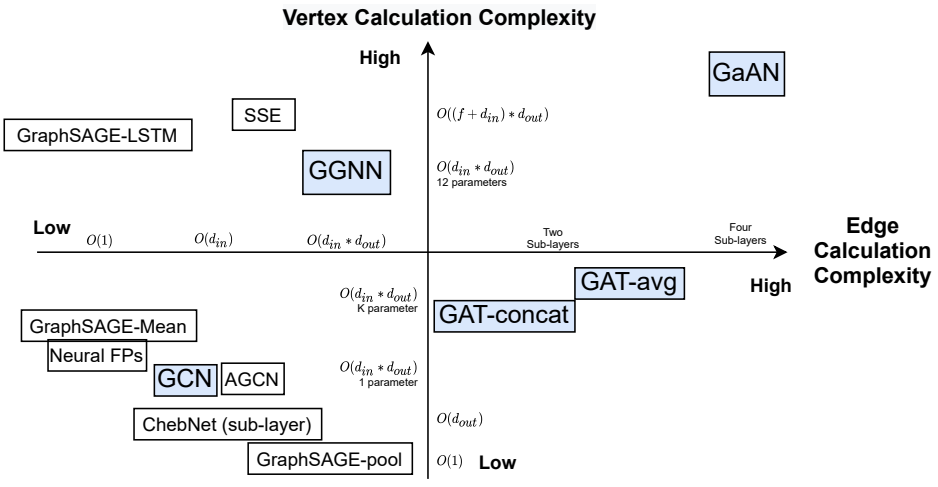


FIGURE 3 Complexity quadrants of typical GNNs. We compare the complexity according to the number of sub-layers, the Big-O notation and the number of parameters to train.

GNN	γ	Complexity
ChebNet[2]	$\boldsymbol{z}_i^{(k)} = 2\boldsymbol{s}_i^{(k)} - \boldsymbol{z}_i^{(k-2)}$	$O(d_{out})$
GCN[1]	$\boldsymbol{h}_i^{l+1} = \boldsymbol{W}^l \boldsymbol{s}_i^l$	$O(d_{in} * d_{out})$
AGCN[3]	$\boldsymbol{h}_i^{l+1} = \boldsymbol{W}^l \boldsymbol{s}_i^l$	$O(d_{in} * d_{out})$
GraphSAGE[5]	$\boldsymbol{h}_i^{l+1} = \delta(\boldsymbol{W}^l[\boldsymbol{s}_i^l \parallel \boldsymbol{h}_i^l])$	$O(d_{in} * d_{out})$
GraphSAGE-pool[5]	$\boldsymbol{h}_i^{l+1} = \boldsymbol{s}_i^l$	$O(1)$
Neural FPs[21]	$\boldsymbol{h}_i^{l+1} = \delta(\boldsymbol{W}^{l,\lvert\mathcal{N}(i)\rvert}(\boldsymbol{h}_i^l + \boldsymbol{s}_i^l))$	$O(d_{in} * d_{out})$
SSE[22]	$\boldsymbol{h}_i^{l+1} = (1-\alpha)\boldsymbol{h}_i^l + \alpha\delta(\boldsymbol{W}_1^l\delta(\boldsymbol{W}_2^l[\boldsymbol{x}_i \parallel \boldsymbol{s}_i^l]))$	$O((f + d_{in}) * d_{out})$
GGNN[4]	$\boldsymbol{z}_i^l = \delta(\boldsymbol{W}^z \boldsymbol{s}_i^l + \boldsymbol{b}^{sz} + \boldsymbol{U}^z \boldsymbol{h}_i^l + \boldsymbol{b}^{hz})$	$O(d_{in} * d_{out})$
	$\boldsymbol{r}_i^l = \delta(\boldsymbol{W}^r \boldsymbol{s}_i^l + \boldsymbol{b}^{sr} + \boldsymbol{U}^r \boldsymbol{h}_i^l + \boldsymbol{b}^{hr})$	
	$\boldsymbol{h}_i^{l+1} = tanh(\boldsymbol{W}\boldsymbol{s}_i^l + \boldsymbol{b}^s + \boldsymbol{U}(\boldsymbol{r}_i^l \odot \boldsymbol{h}_i^l) + \boldsymbol{b}^h))$	
	$\boldsymbol{h}_i^{l+1} = (1 - \boldsymbol{z}_i^l) \odot \boldsymbol{h}_i^l + \boldsymbol{z}_i^l \odot \boldsymbol{h}_i^{l+1}$	
GAT[6]	Preprocessing:	concat:$O(d_{in} * d_{out})$
	$\hat{\boldsymbol{h}}_{i,(k)}^l = \boldsymbol{W}^{l,k}\boldsymbol{h}_i^l$	average:$O(K * d_{in} * d_{out})$
	Sub-layer 0:	Two sub-layers
	$\hat{\boldsymbol{a}}_{i,(k)}^{l,0} = \boldsymbol{s}_{i,(k)}^{l,0}$	
	Sub-layer 1:	
	$\boldsymbol{h}_i^{l+1} = \delta(\boldsymbol{s}_i^{l,1})$	
GaAN[7]	Sub-layer 0:	$O(max(K * d_v + d_{in}, 2 * d_{in} + d_m) * d_{out})$
	$\hat{\boldsymbol{a}}_{i,(k)}^{l,0} = \boldsymbol{s}_{i,(k)}^{l,0}$	Four sub-layers
	After sub-layer 0,1,2,3:	
	$\boldsymbol{g}_i^l = \boldsymbol{W}_g^l[\boldsymbol{h}_i^l \parallel \boldsymbol{s}_i^{l,2} \parallel \boldsymbol{s}_i^{l,3}] + \boldsymbol{b}_g^l$	
	$\boldsymbol{h}_i^{l+1} = \boldsymbol{W}_o^l[\boldsymbol{h}_i^l \parallel (\boldsymbol{g}_i^l \odot \boldsymbol{s}_i^{l,1})] + \boldsymbol{b}_o^l$	

TABLE 2 Typical graph neural networks and their vertex calculation functions. d_{in} and d_{out} are dimensions of the input and output hidden feature vectors, respectively. Blue variables are model parameters to learn. In Neural FPs, $\boldsymbol{W}^{l,|\mathcal{N}(i)|}$ is the weight matrix for vertices with degree $|\mathcal{N}(i)|$ at layer l.

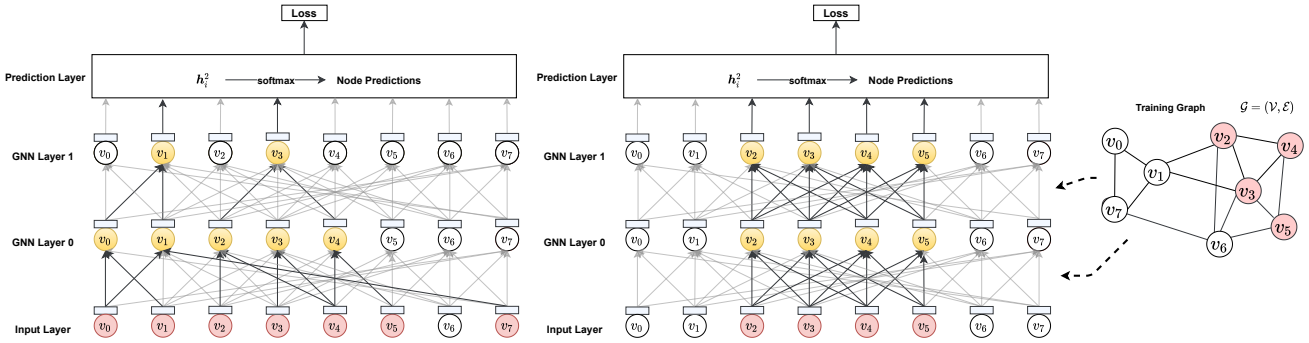
heads can be merged by concatenating or by averaging. For concatenating, the dimension of the hidden feature vector of each head d_{head} is d_{out}/K. For averaging, d_{head} is d_{out}, multiplying the complexity by K. Each GAT layer consists of a preprocessing step and two sub-layers. In the preprocessing step, GAT calculates the attention vectors $\hat{\boldsymbol{h}}_{i,(k)}^l$ for every vertex v_i in every head k. The first sub-layer uses the attention vectors to calculate the attention weights of every edge in every head $\boldsymbol{m}_{j,i,(k)}^{l,0}$. After aggregation, the first sub-layer gets the summation of attention weights of every vertex in every head $\hat{\boldsymbol{a}}_{i,(k)}^{l,0}$. The second sub-layer calculates the normalized attention weights $\alpha_{j,i,(k)}$ for every edge in every head and aggregates the hidden feature vectors with the normalized weights in the message-passing. GAT uses the aggregated vector $\boldsymbol{s}_i^{l,1}$ of the second sub-layer as the output hidden vector directly.

GaAN (high vertex & edge calculation complexity): Based on the multi-head mechanism, GaAN[7] introduces a convolutional subnetwork to control the weight of each head. Each GaAN layer consists of four sub-layers. The first two sub-layers are similar to GAT, and the last two sub-layers build the convolutional subnetwork. The first sub-layer aims to get the summation of attention weights of every vertex in each head $\hat{\boldsymbol{a}}_{j,i,(k)}^{l,0}$. The second sub-layer aggregates the feature vectors with the normalized attention weights $\hat{\boldsymbol{a}}_{i,(k)}^{l,0}$. The third and fourth sub-layers aggregate feature vectors with element-wise max and element-wise mean operators separately. After the four sub-layers, the vertex updating funciton uses the aggregated vectors of last three sub-layers $\boldsymbol{s}_i^{l,1}$, $\boldsymbol{s}_i^{l,2}$, $\boldsymbol{s}_i^{l,3}$ to generate the output hidden vector $\boldsymbol{h}_i^{l+1}$

2.4 | Sampling Techniques

By default, GNNs are trained in a full-batch way, using the whole graph in each iteration. The full-batch gradient descent has two disadvantages[23]. It has to cache intermediate results of all vertices in the forward phase, which consumes lots of memory space. It updates the parameters only once for each epoch, slowing the convergence of gradient descent.

To train GNNs in a mini-batch way, the sampling techniques are proposed. In each mini-batch, they sample a small subgraph from the whole graph $\mathcal{G}$ and uses the subgraph to update the model parameters. The sampling techniques only active the graph



(a) Neighbor sampling **(b)** Graph sampling

FIGURE 4 Training a GNN with sampling techniques. The faded graph neurons and their connections are inactivated.

neurons and the connections that appear in the sampled subgraph between GNN layers, as shown in **FIGURE** 4. In active graph neurons and connections do not participate in the training of this mini-batch, saving lots of computation and storage costs. Moreover, it may reduce the risk of overfitting the training graph. The existing sampling techniques can be classified into two groups **neighbor sampling and graph sampling** based on whether different GNN layers sample different subgraphs.

The neighbor sampling techniques[5,24,25,26,27] sample nodes or edges layer by layer. The sampled subgraphs of different GNN layers may be *different*, as shown in **FIGURE** 4a. GraphSAGE[5] is the representative technique. The tecnnique first samples several vertices from $\mathcal{V}$ in the last GNN layer. Then it repeatedly samples the neighbors of those sampled vertices in the previous layer until the input layer. For every sampled vertex v_i in the GNN layer l, GraphSAGE samples at most S^l neighbors of v_i in the previous GNN layer. S^l is the hyper-parameter that is usually much smaller than $|\mathcal{V}|$. In this way, GraphSAGE limits the neighborhood sizes of the vertices in the sampled subgraph, especially high-degree vertices.

The graph sampling techniques[28,23,29] sample a subgraph for each mini-batch and use the same sampled graph for all GNN layers, as shown in **FIGURE** 4b. They differ in the methods to sample subgraphs. The cluster sampler technique[23] is the representative technique. Given a training graph $\mathcal{G}$, it partitions $\mathcal{G}$ into dense clusters. For each mini-batch, it randomly pick K clusters to form the sampled subgraph, where K is the hyper-parameter.

3 | EVALUATION DESIGN

We design a series of experiments to explore the performance bottleneck in training graph neural networks. We first introduce the experimental setting in Section 3.1 and then give out our experimental scheme in Section 3.2. The evaluation results are presented and analyzed later in Section 4.

3.1 | Experimental Setting

Experimental Environment

All the experiments were conducted in a CentOS 7 Linux server with kernel version 3.10.0. The server had 40 cores and 90 GB main memory. The server was equipped with an NVIDIA Tesla T4 GPU with 16GB GDDR6 memory. For the software environment, we adopted Python 3.7.7, PyTorch 1.5.0, and CUDA 10.1. We implemented all the GNNs with PyTorch Geometric 1.5.0.

Dataset

We used six real-world graph datasets as listed in **TABLE** 3 that were popular in the GNN accuracy evaluation[30,29,31]. For directed graphs, PyG converts them into undirected ones during the data loading. Thus, the average degree of a directed graph $\bar{d} = \frac{2|\mathcal{E}|}{|\mathcal{V}|}$. For an undirected graph, $\mathcal{E}$ already contains two-direction edges and $\bar{d} = \frac{|\mathcal{E}|}{|\mathcal{V}|}$. For the `cam` dataset, we generated random dense feature vectors. We also used random graphs generated by the R-MAT graph generator[32] in the experiments, to explore the effects of graph topological characteristics (like the average degree) on the performance bottleneck. Input feature

Dataset	$\|\mathcal{V}\|$	$\|\mathcal{E}\|$	$\bar{d}$	$dim(\boldsymbol{x})$	#Class	Directed
pubmed (pub)[30]	19,717	44,324	4.5	500	3	Yes
amazon-photo (amp)[31]	7,650	119,081	31.1	745	8	Yes
amazon-computers (amc)[31]	13,752	245,861	35.8	767	10	Yes
coauthor-physics (cph)[31]	34,493	247,962	14.4	8415	5	Yes
flickr (fli)[29]	89,250	899,756	10.1	500	7	No
com-amazon (cam)[33]	334,863	925,872	2.8	32	10	No

TABLE 3 Dataset overview. $\bar{d}$ represents the average vertex degree. $dim(\boldsymbol{x})$ is the dimension of the input feature vector.

vectors of random graphs were random dense vectors with a dimension of 32. Vertices of random graphs were classified into 10 classes randomly.

Learning Task

We used the node classification as the target task in GNNs due to its popularity in real-world applications. We trained GNNs with the semi-supervised learning setting. All vertices and their input feature vectors were used, but only parts of the vertices were attached with labels during the training and they were used to calculate the loss and gradients. The vertices with unseen labels were used in the evaluation phase to evaluate the accuracy of the current parameters.

GNN Implementation

We implemented the four typical GNNs (GCN, GGNN, GAT, GaAN) with PyTorch Geometric 1.5.0. To compare the performance characteristics of four GNNs side-by-side, we used a unified GNN structure for them: Input Layer → GNN Layer 0 → GNN Layer 1 → Softmax Layer (to prediction). The structure was popular in the experimental evaluation of GCN[1], GAT[6] and GaAN[7]. Since a GGNN layer requires the input and output hidden feature vectors have the same dimension, we added two multi-layer perceptron(MLP) layers to transform the dimensions of the input/output feature vectors of the whole GNN: Input Layer → MLP → GGNN Layer 0 → GGNN Layer 1 → MLP → Softmax Layer. We stored the dataset and the model parameters on the GPU side. All the training was conducted on the GPU.

Hyper-parameters

We use $dim(\boldsymbol{v})$ to denote the dimension of a vector $\boldsymbol{v}$. We picked the hyper-parameters of GNNs according to their popularity in the original papers. We used the same set of hyper-parameters for all the datasets. Some hyper-parameters like the dimensions of hidden feature vectors were common in the four GNNs and we set them to the same values. For GCN/GAT/GaAN, we set $\boldsymbol{h}_i^0 = \boldsymbol{x}_i$, $dim(\boldsymbol{h}_i^1) = 64$, and $dim(\boldsymbol{h}_i^2) = \#Classes$. For GAT, we set the hyper-parameters according to its paper[6]. The first GAT layer had 8 heads with the dimension of each head $d_{head}^0 = 8$. The first GAT layer merged the heads by concatenating. The second GAT layer used a single head with the dimension $d_{head}^1 = d_{out}^1 = \#Classes$. For GGNN, since it uses extra MLP layers to transform the dimensions of the input/output feature vectors, we set $dim(\boldsymbol{h}_i^0) = dim(\boldsymbol{h}_i^1) = dim(\boldsymbol{h}_i^2) = 64$. We used 8 heads in the both GaAN layers with $d_a = d_v = 8$ and $d_m = 64$.

Sampling Techniques

We picked the neighbor sampler from GraphSAGE[5] and the cluster sampler from ClusterGCN[23] as the typical sampling techniques respectively. For the neighbor sampler, we set the neighborhood sample sizes as 25 for GNN Layer 1, and 10 for GNN Layer 0 and set the default batch size as 512 from its paper[5]. For the cluster sampler, we partitioned every input graph into 1500 partitions and used 20 partitions per batch according to the parameters used in its paper[23].

3.2 | Experimental Scheme

To find out the performance bottleneck in GNN training, we conducted the experimental analysis with four questions. The answers to those questions will give us a more comprehensive view of the performance characteristics of GNN training.

Q1 *How do the hyper-parameters affect the training time and the memory usage of a GNN?* (Section 4.1)

Every GNN has a group of hyper-parameters like the number of GNN layers and the dimensions of hidden feature vectors. The hyper-parameters affect the training time per epoch and the peak memory usage during the training. To evaluate their effects, we measured how the training time per epoch and the peak memory usage (of the GPU) changed as we increased the values of the hyper-parameters. Through the experiments, we want to verify the validity of the computational complexity analysis in **TABLE** 1 and **TABLE** 2. If the complexity analysis is valid, we can analyze the bottleneck theoretically.

Q2 *Which stage is the most time-consuming stage in GNN training?* (Section 4.2)

We can decompose the training time on different levels: layer level, edge/vertex calculation level, and the basic operator level. On each level, we decomposed the training time of an epoch into several stages. The most time-consuming stage is the performance bottleneck. Optimizing its implementation can significantly reduce the training time.

Q3 *Which consumes most of memory in GNN training?* (Section 4.3)

The limited memory capacity of a GPU is the main factor preventing us from training GNNs on big graphs. We measured the peak memory usage of GNN training under different graph scales, input feature dimensions, and average vertex degrees. Based on the results, we analyzed which is the most memory-consuming component in a GNN. Reducing its memory usage will enable us to train GNNs on bigger graphs under the same memory capacity.

Q4 *Can the sampling techniques remove the performance bottleneck in GNN training?* (Section 4.4)

Theoretically, the sampling techniques can significantly reduce the number of graph neurons that participate in the training of a batch. Consequently, the training time and the memory usage should also decrease. To validate the effectiveness of the sampling techniques, we measured the training time and peak memory usage under different batch sizes. If the sampling techniques are effective, they are the keys to conduct GNN training on very big graphs. If they are not effective, we want to find out which impairs its efficiency.

4 | EVALUATION RESULTS AND ANALYSIS

We answer the four questions in Section 3.2 one by one with experiments. Without otherwise mentioned, the reported training time per epoch is the average wall-clock training time of 50 epochs, excluding abnormal epochs [‡].

4.1 | Effects of Hyper-parameters on Performance

According to **TABLE** 1 and **TABLE** 2, the time complexities of ϕ and γ are linear to each hyper-parameter separately. If we increase one of the hyper-parameters and fix the others, the training time should increase linearly.

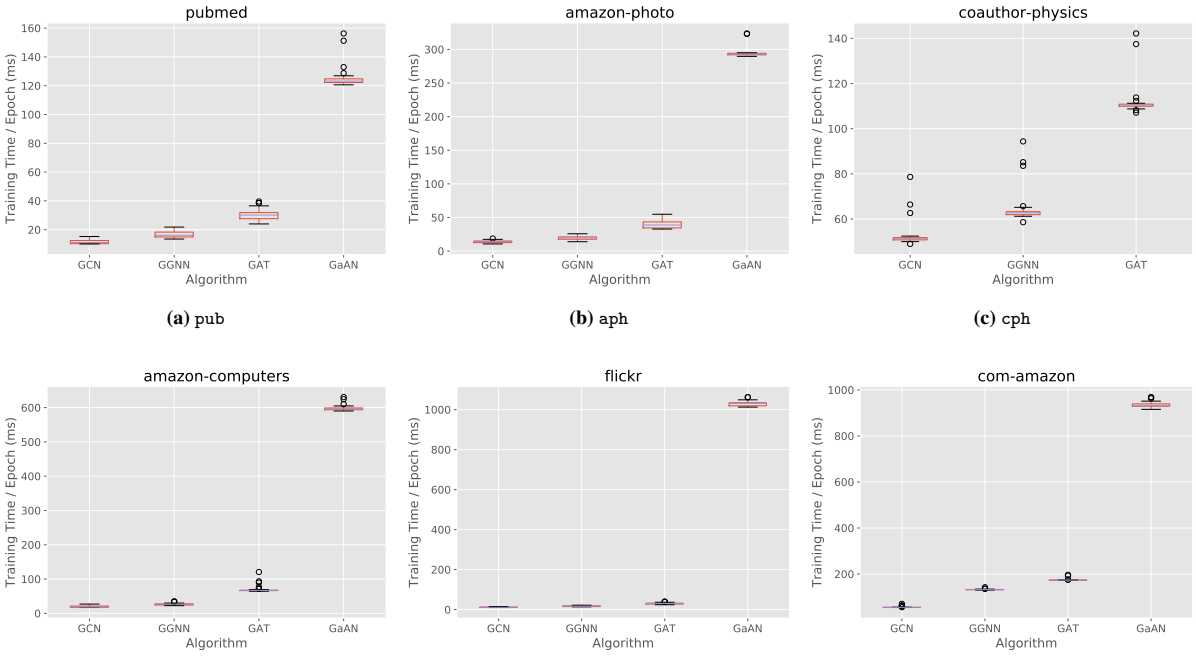
To verify the time complexity analysis in **TABLE** 1 and **TABLE** 2, we first compare the training time of the four GNNs on the real-world datasets in 3. The ranking of the training time is GaAN $\gg$ GAT $>$ GGNN $>$ GCN in all cases. Since the real-world graphs have more edges than edges ($m > n$), the time complexity of the edge calculation stage affects more than the vertex calculation stage. The ranking is consistent with the time complexity analysis.

To further evaluate the effects of the hyper-parameters, we measure the training time of each GNN with varying hyper-parameters in **FIGURE** 6.

Since $dim(\boldsymbol{h^0})$ and $dim\boldsymbol{h^1}$ are determined by the dataset with $dim(\boldsymbol{h}^0 = dim(\boldsymbol{x}))$ and $dim(\boldsymbol{h}^2) = \#Classes$, for GCN and GGNN, the only modifiable hyper-parameter is the hidden dimension $dim(\boldsymbol{h}^1)$ with $dim(\boldsymbol{h}^1) = d^0_{out} = d^1_{in}$. According to the time complexity analysis, if we fix other hyper-parameters but only increase $dim(\boldsymbol{h}^1)$, the computing costs of the GNN layer 0 and the GNN layer 1 both increase linearly with $dim(\boldsymbol{h}^1)$, causing the training time of the whole GNN also increasing linearly. **FIGURE** 6a and **FIGURE** 6b show that the training time of GCN and GGNN increased linearly with $dim(\boldsymbol{h}^1)$ when $dim(\boldsymbol{h}^1)$ is big, consistent with the time complexity analysis.

For GAT, we modify the number of heads K and the dimension of each head d_{head} in the GAT layer 0. The dimension of the hidden feature vector is determined correspondingly as $d^0_{out} = d^1_{in} = dim(\boldsymbol{h}^1) = Kd_{head}$. Thus, the computing costs of the GAT layer 0 and the GAT layer 1 increase linearly with K and d_{head} separately. **FIGURE** 6c confirms the theoretical analysis.

[‡]During the training of some epochs, there are extra profiling overheads from NVIDIA Nsight Systems and GC pauses from the Python interpreter that significantly increase the training time. Assume Q1 and Q3 are the 25% and 75% quantiles of the training time of 50 epochs, respectively. We regard the epochs with the training time *outside* the range of $[Q1 - 1.5 * (Q3 - Q1), Q3 + 1.5 * (Q3 - Q1)]$ as abnormal epochs.



(a) `pub` **(b)** `aph` **(c)** `cph`

(d) `amc` **(e)** `fli` **(f)** `cam`

FIGURE 5 Distribution of the wall-clock training time of 50 epoches on different datasets. GaAN crashed due to out of memory exception on the `cph` dataset.

For GaAN, it is also based on the multi-head mechanism. Its time complexity is affected by $dim(\boldsymbol{h}^1)$ ($d^0_{out} = d^1_{in} = dim(\boldsymbol{h}^1)$), d_a, d_v, d_m and the number of heads K. **FIGURE** 6d demonstrates that the training time increases linearly with the hyper-parameters, except for $dim(\boldsymbol{h}^1)$. As $dim(\boldsymbol{h}^1)$ increases, the training time increases first slightly and then linearly. We observe similar phenomena in GCN, GGNN, and GAT with low hyper-parameters. When the hyper-parameters is too low, the GNN training cannot make full use of the computing power of the GPU. When the hyper-parameter becomes high enough, the training time increases linearly, supporting the time complexity analysis.

We further measured the effects of the hyper-parameters on the peak GPU memory usage in **FIGURE** 7. The memory usage also increases linearly as the hyper-parameters increase for all GNNs, except for GaAN on $dim(\boldsymbol{h}^1)$. As the hidden feature vectors $\boldsymbol{h}^1$ consume a small proportion of memory in GaAN, the growth in the memory usage is not noticeable until $dim(\boldsymbol{h}^1)$ is large enough.

Summary

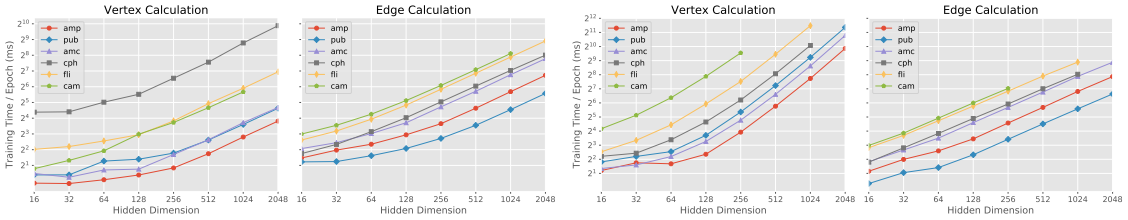
The complexity analysis in **TABLE** 1 and **TABLE** 2 is valid. Fixing other hyper-parameters, each hyper-parameter itself affects the training time and the memory usage of a GNN Layer **in a linear way**. Algorithm engineers can adjust hyper-parameters according to the time complexity to avoid explosive growth in the training time and memory usage.

4.2 | Training Time Breakdown

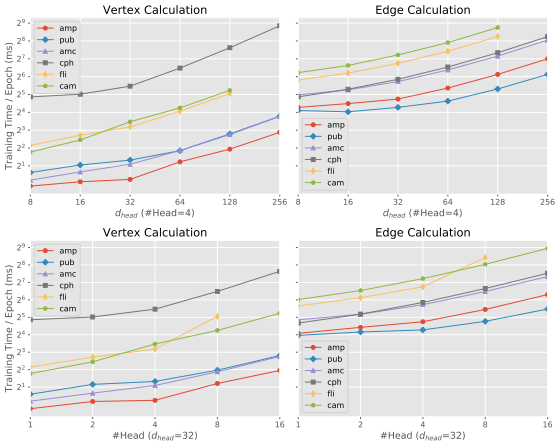
To find out which stage/step dominates the training time, we decompose the training time and analyze the performance bottleneck level by level.

4.2.1 | Layer Level

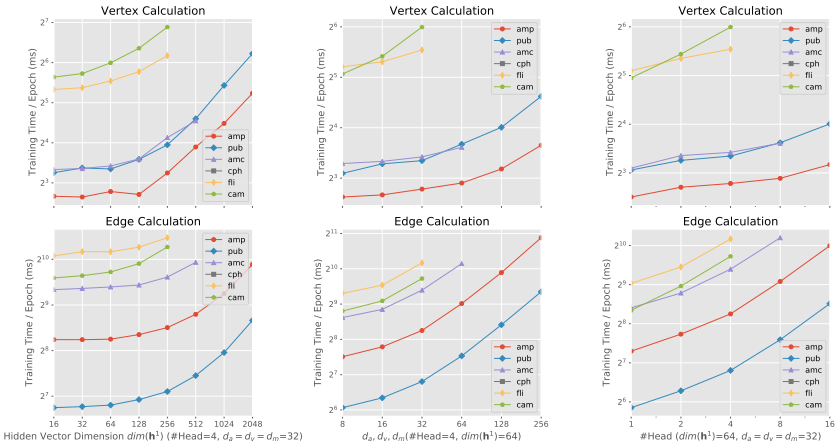
FIGURE 8 decomposes the training time of a GNN on the layer level. The training time of each layer is the summation of the time in the forward, backward, and evaluation phases. In GCN, GAT, and GaAN, the time spent on the layer 0 is much larger than the layer 1. In those GNNs, the dimensions of the input/output feature vectors in the layer 0 are much larger than the



(a) GCN **(b)** GGNN



(c) GAT

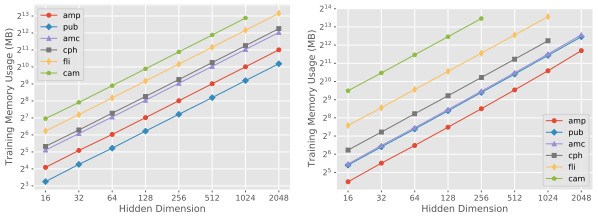


(d) GaAN

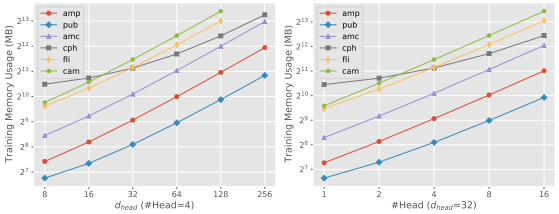
FIGURE 6 Effects of hyper-parameters on the edge/vertex calculation time.

dimensions in the layer 1. $d_{in}^0 = dim(\boldsymbol{x})$, $d_{out}^0 = d_{in}^1 = 64$ and $d_{out}^1 = \#Class$ and $dim(\boldsymbol{x}) \gg \#Class$. For GaAN, since it requires the dimensions of the input/output feature vectors must be the same, the hyper-parameter are $d_{in}^0 = d_{out}^0 = d_{in}^1 = d_{out}^1 = 64$ and the training time of both layers is close.

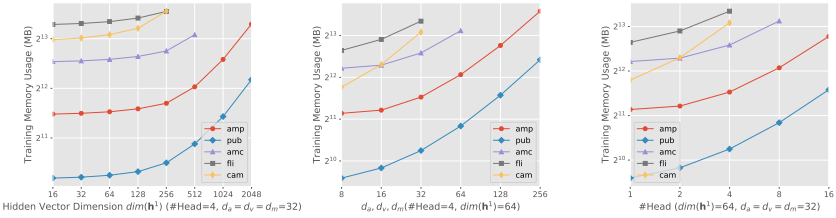
Each GNN layer can be further divided into the vertex and the edge calculation stages. In **FIGURE** 8, GCN spends most of the training time on the edge calculation stage in most datasets. A special case is cph dataset. The dimension of the input feature vectors is very high in cph, making the vertex calculation stage of the GCN Layer 0 spend considerable time. GGNN also spends the majority of its training time on the edge calculation stage. But the high time complexity of its vertex updating function γ makes the ratio of the vertex calculation in the total training time much higher than the other GNNs. For GAT and



(a) GCN **(b)** GGNN



(c) GAT



(d) GaAN

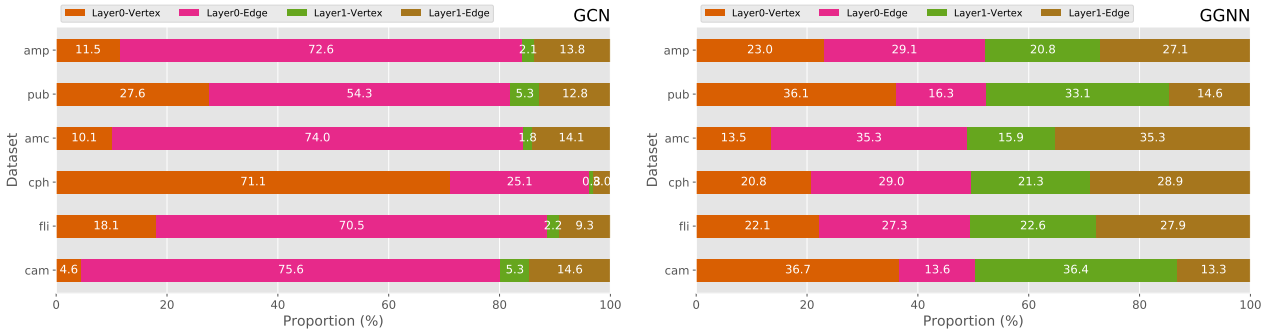
FIGURE 7 Effects of hyper-parameters on the peak GPU memory usage during the training, excluding the memory used by the dataset and the model parameters.

GaAN, due to their high edge calculation complexity, the edge calculation stage is the absolutely dominant stage. In summary, *the edge calculation stage is the most time-consuming stage in the GNN training*.

The experimental results also indicate that the average degree of the dataset affects the time-consuming proportion of the edge/vertex calculation time. For GaAN, the time spent on the vertex calculation stage exceeds the edge calculation stage on the `pub` and `cam` datasets, because the average degrees of the two datasets are low, making $|\mathcal{E}|$ and $|\mathcal{V}|$ much closer. To evaluate the effects of the average degree, we used the R-MAT generator to generate random graphs with 50k vertices and the average degrees ranging from 2 to 100. **FIGURE** 9 shows the training time of the four GNNs under different average degrees. As the average degree increases, the training time of the edge calculation stage grows *linearly*. For GCN, GAT, and GaAN, the edge calculation stage dominates the entire training time even under small average degrees. Only for GGNN that has high vertex and low edge calculation complexities, the training time of the vertex calculation stage exceeds the edge calculation stage under low average degrees (< 5). Therefore, *improving the efficiency of the edge calculation stage is the key to reduce the GNN training time*.

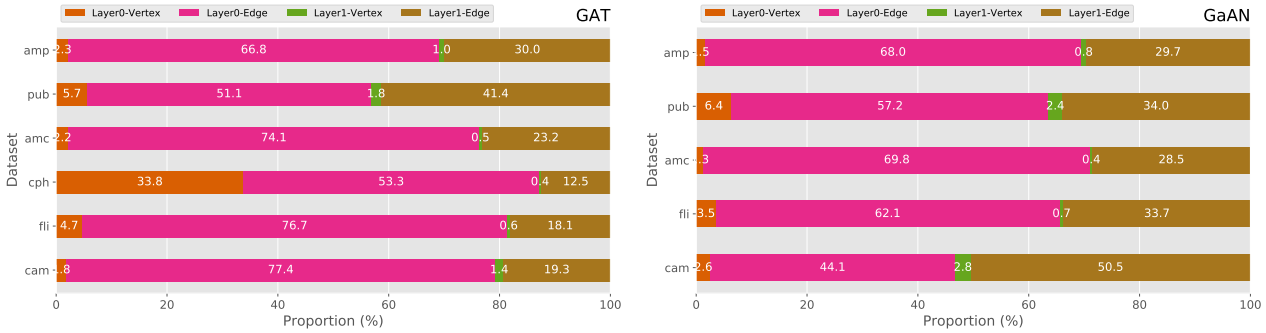
4.2.2 | Step Level in Edge Calculation

In the implementation of PyG, the edge calculation stage can be decomposed into four steps: collection, messaging, aggregation, and updating, as shown in **FIGURE** 10. The edge index is a matrix with M rows and 2 columns that holds the edge set of the graph, where $M = |\mathcal{E}|$. The two columns of the matrix store the source vertex and the target vertex of each edge, respectively. The collection step copies the vertex feature vectors from the previous layer $\boldsymbol{h}_i^l$ to the ends of each edge in the edge index, to form the parameters tensor $[\boldsymbol{h}_i^l, \boldsymbol{h}_j^l, \boldsymbol{e}_{i,j}^l]$ of the messaging function ϕ. This step only involves the data movement. The messaging step calls the messaging function ϕ to get message vectors of all edges $\boldsymbol{m}_{i,j}^l$. The aggregation step aggregates the message vectors with the same target vertex into an aggregated vector $\boldsymbol{s}_i^l$ with the aggregation operation Σ. The updating step is optional. It performs



(a) GCN

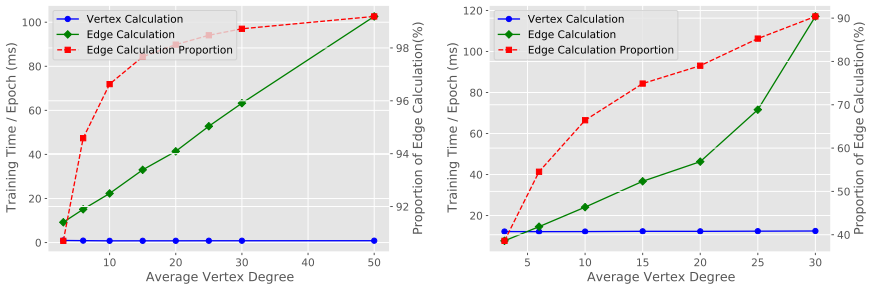
(b) GGNN



(c) GAT

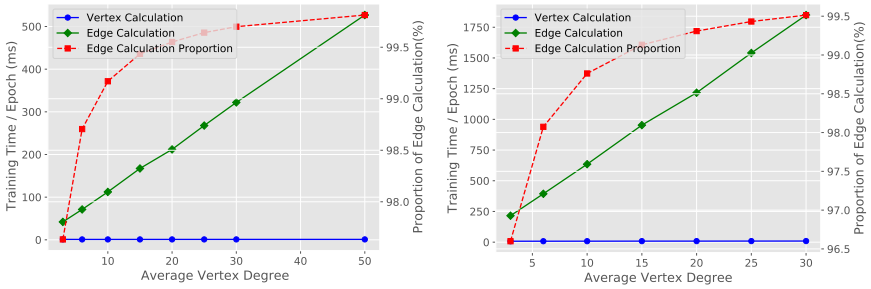
(d) GaAN

FIGURE 8 Training time breakdown on the layer level. The training time of each layer includes the time spent on the forward, backward and evaluation phases. Each layer is further decomposed into the vertex and the edge calculation stages.



(a) GCN

(b) GGNN



(c) GAT

(d) GaAN

FIGURE 9 Effects of the average degree on the time proportion of the edge/vertex calculation. Graphs were generated with the R-MAT generator by fixing the number of vertices as 50,000.

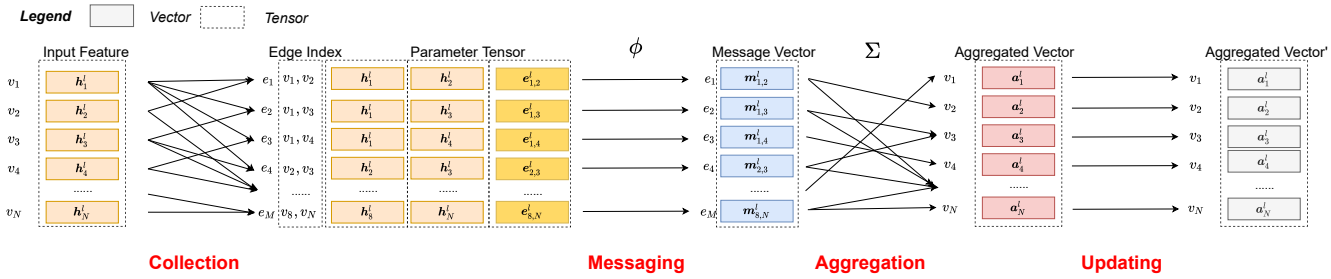


FIGURE 10 Step decomposition of the edge calculation in the GNN layer l.

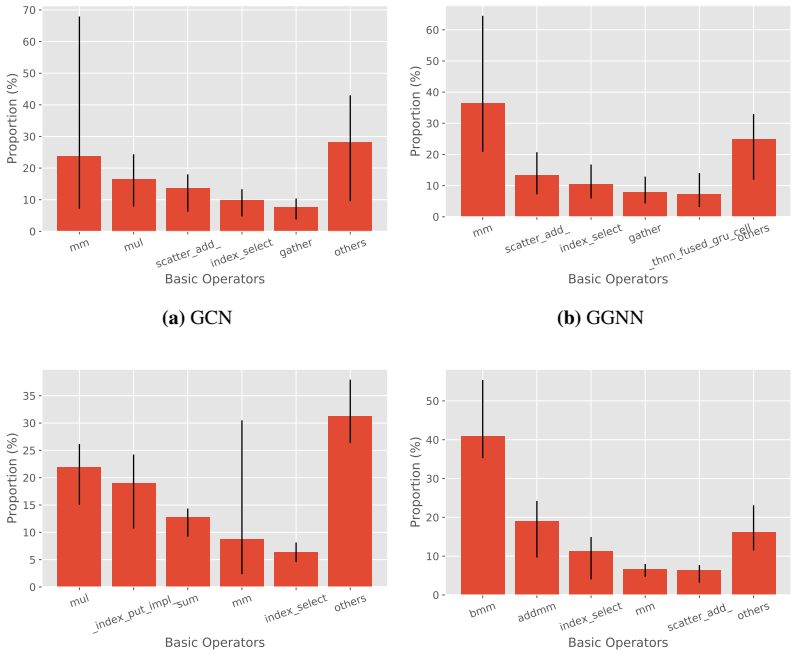
(a) GCN **(b)** GGNN

(c) GAT **(d)** GaAN

FIGURE 11 Training time breakdown of the edge calculation stage (including both GNN layers).

an additional transformation on the aggregated vectors (for example, adding bias in GCN). The aggregated vectors $\boldsymbol{s}_i^l$ (after the updating step) will be fed into the vertex updating function γ as one of the input parameters.

We decompose the execution time of the edge calculation stage in **FIGURE** 11. In each GNN, the proportions of the four steps are rather stable, rarely affected by datasets. For GAT and GaAN with the high edge calculation complexity, the messaging step consumes most of the training time. For GCN and GGNN with the low edge complexity, the proportions of the steps are close. Since the messaging function ϕ of GGNN uses the pre-computed $\hat{\boldsymbol{h}}_i^l$ as the message vector directly, the time spent on the messaging step of GGNN is negligible. Although the collecting step does not conduct any computation and only involves data movement, it occupies noticeable execution time in all the GNNs. The experiments show that *the performance bottleneck on the step level depends on the complexity of the messaging function ϕ*. For GNNs with the high complexity, the messaging function ϕ is the performance bottleneck. Optimizing the implementation of ϕ can significantly reduce the training time. For the other GNNs, optimization should focus on reducing the costs of the collection and the aggregation steps. Additionally, improving the efficiency of the collection step can benefit all GNNs.



(a) GCN **(b)** GGNN

(c) GAT **(d)** GaAN

FIGURE 12 Top 5 time-consuming basic operators of typical GNNs. The time proportion of each basic operator is averaged over all graphs with the error bar indicating the maximal and the minimal.

4.2.3 | Operator Level

The functions ϕ, Σ and γ in the vertex and edge calculation are made up of a series of basic operators implemented on the GPU, like the matrix multiplication `mm`, the elementwise multiplication `mul` and the index-based selection `index_select`. **FIGURE** 12 shows the five most time-consuming basic operators in each GNN, averaged over all the real-world graphs in **TABLE** 3.

GCN

The most time-consuming basic operator is the matrix multiplication `mm` used in the vertex updating function γ. The elementwise multiplication `mul` used in the messaging function ϕ is also time-consuming. The other three operators are used in the edge calculation stage: `scatter_add_` for the aggregation step in the forward phase, `gather` for the aggregation step in the backward phase, and `index_select` for the collection step. For GCN, the basic operators related to the edge calculation stage consume the majority of the training time.

GGNN

The top basic operator is `mm` used in the vertex updating function γ. Due to its high time complexity, the proportion of the `mm` is much higher than the other operators. The `thnn_fused_gru_cell` operator used in the backward phase of γ is also noticeable. The other three operators are used in the edge calculation stage.

GAT

All the top basic operators except for `mm` are related to the edge calculation stage. The `mm` operator is used in the vertex updating function γ.

GaAN

The top basic operator is `bmm` used in the messaging function ϕ. The `addmm` operator and the `mm` operator are used in both the vertex and the edge calculation stages, where the edge calculation stage is dominant.

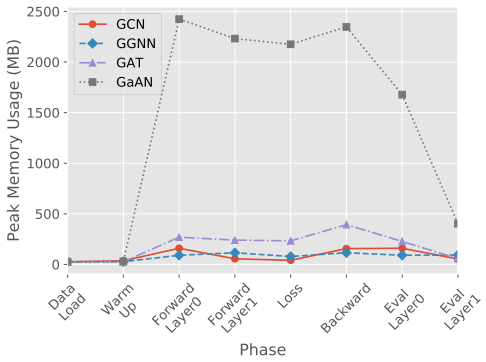


FIGURE 13 Memory usage of each phase. Dataset: `amp`.

In general, the most time-consuming operator in the four GNNs, is still the matrix multiplication `mm` and the elementwise multiplication `mul`, *making GNN training suitable for GPUs*. Although the aggregation step in the edge calculation stage is relatively simple (like sum and mean), the related operators `scatter_add` and `gather` still consume a certain amount of the time. The operators have to synchronize between hardware threads to avoid updating the same aggregated vector at the same time. They also conduct non-regular memory access with the access pattern determined by the edge set dynamically. For GPUs, they are less efficient than `mm`. The index-based selection `index_select` operator from the collection step consume around 10% of the time in all GNNs. Though GPUs have high on-chip memory bandwidth, improving the efficiency of `scatter_add/gather/index_select` can benefit the training of all kinds of GNNs.

Summary

The GNN training is suitable for GPUs. **The edge calculation stage is the main performance bottleneck in most cases**, except for training GNNs with high vertex calculation complexity on low-average-degree graphs. The performance bottleneck in the edge calculation stage depends on the time complexity of the messaging function ϕ.

- If the time complexity of ϕ is **high**, the **efficiency of** ϕ limits the performance. Reducing its computation cost (via optimizing its implementation or modifying the algorithm) can significantly reduce the training time.
- If the time complexity of ϕ is **low**, the **collection step** and the **aggregation step** limit the performance. The collection step involves lots of data movement. The aggregation step suffers from data synchronization and non-regular data access. Optimizing their implementations can significantly reduce the training time.

4.3 | Memory Usage Analysis

During the GNN training, all data (including datasets and intermediate results) are stored in the on-chip memory of the GPU. Compared with the main memory on the host side, the capacity of the GPU memory is very limited. *The GPU memory capacity limits the scales of the graphs that a GPU can train GNNs on*. For example, GaAN is unable to train on `cph` dataset due to the out of memory exception.

FIGURE 13 shows the peak memory usage of each phase in the GNN training on the `amp` dataset. The trend is similar on the other datasets. *The GNN training achieves its peak memory usage in the forward and the backward phases*. The forward phase generates lots of intermediate results. Some key intermediate results are cached, increasing memory usage. The cached results are used in the gradient calculation in the backward phase. **FIGURE** 14 shows the computation graph of the vertex updating function γ of GGNN. The computation graph has a large number of operators. Each operator generates an intermediate tensor. Some key intermediate tensors are cached. The cached tensors are the main source of memory usage in the loss phase. By the end of the backward phase, the cached tensors are released. Since the evaluation phase does not need to calculate the gradients, it does not cache intermediate tensors. Its memory usage declines sharply.

The peak memory usage during the GNN training far exceeds the size of the dataset itself. We define the *memory expansion ratio* (MER) as the ratio of the peak memory usage during the training to the memory usage after loading the dataset. **FIGURE** 15

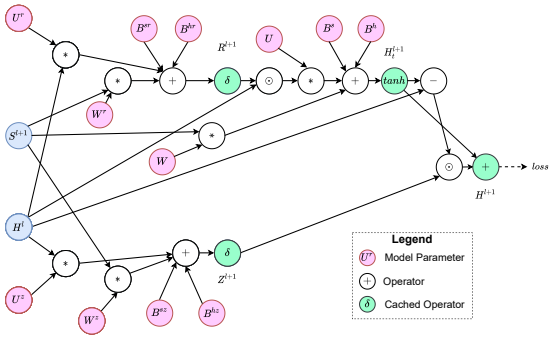


FIGURE 14 Computation graph of the vertex updating function γ of GGNN.

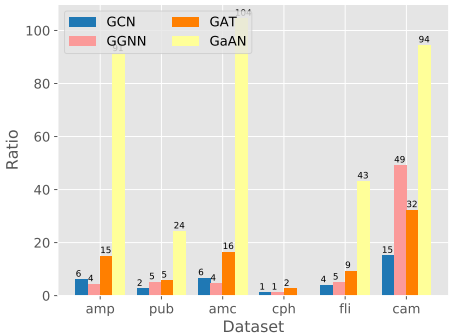


FIGURE 15 Memory expansion ratios of typical GNNs.

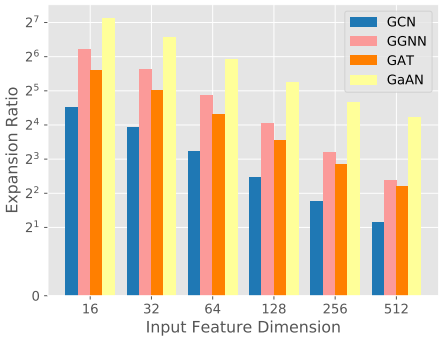
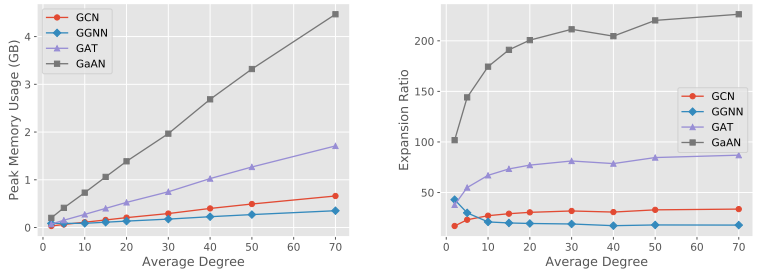


FIGURE 16 Memory expansion ratio under different dimensions of the input feature vectors. Dataset: `cam`.

compares MER of different GNNs. GCN has the lowest MER (up to 15) while GaAN has the highest MER (up to 104). *The high MER limits the data scalability of GNNs*, making GPUs unable to handle big graphs.

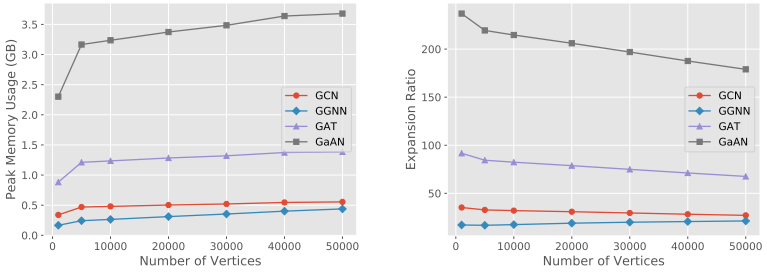
FIGURE 15 also indicates that the same GNN has different MERs for different datasets. Two characteristics of a dataset affect the MER: the dimension of the input feature vector and the average degree.

Given the same graph, the scales of the intermediate results are mainly affected by the hyper-parameters of the GNN. If the dimension of the input feature vectors is high (like the `cph` dataset), the size of the dataset is large. The size may become comparable to the scales of the intermediate results, making the MER low. To find out how the dimension affects the MER, we generated random input feature vectors with different dimensions for the `cam` dataset and measured the MER in **FIGURE** 16. For a GNN under the same hyper-parameters, *the MER decreases as the dimension of the input feature vectors increases*.



(a) Peak memory usage **(b)** Memory expansion ratio

FIGURE 17 Memory usage under different average degrees of the graph. The graph was generated with the R-MAT generator fixing the number of vertices at 10K and the dimension of the input feature vectors at 32.



(a) Peak memory usage **(b)** Memory expansion ratio

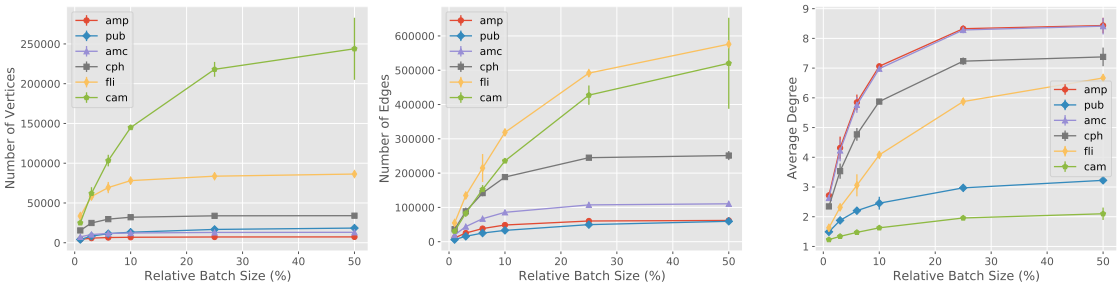
FIGURE 18 Memory usage under different numbers of vertices of the graph. The graph was generated with the R-MAT generator fixing the number of edges at 500K and the dimension of the input feature vectors at 32.

The average degree of the graph also affects MER by influencing the relative scales of the intermediate results from the edge/vertex calculation stages. Fixing the number of vertices $|\mathcal{V}|$, we used the R-MAT generator to generate random graphs with different average degrees. **FIGURE** 17 shows how the memory usage changes according to the average degrees. As the average degree $\bar{d}$ increases, the number of edges $|\mathcal{E}|$ increases and the peak memory usage increases *linearly* with $\bar{d}$. The edge calculation stage gradually dominates the memory usage and *the MER converges to a stable value*. The stable value is determined by the complexity of the edge calculation stage. Except for GGNN, the MER of the other GNNs increases as $\bar{d}$ increases. As GGNN has high vertex calculation complexity, the MER related to the vertex calculation stage is much larger than the edge calculation stage. When the edge calculation stage dominates the memory usage, its MER becomes smaller.

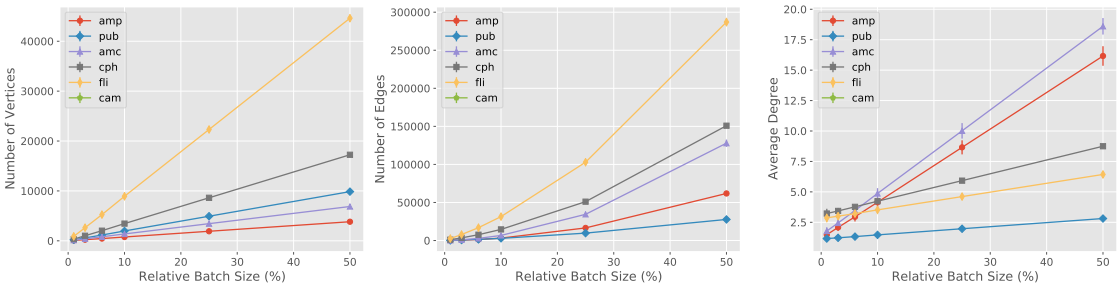
We also fixed the number of edges $|\mathcal{E}|$ in the graphs and generated random graphs with different $|\mathcal{V}|$. **FIGURE** 18 shows how the memory usage changes according to $|\mathcal{V}|$. All GNNs are insensitive to the changes in $|\mathcal{V}|$ compared to $|\mathcal{E}|$. Except for GGNN, the MERs of the other GNNs decline as $|\mathcal{V}|$ increases because the scales of the datasets increase more quickly than the scales of the intermediate results. As GGNN has high vertex calculation complexity, the scales of the intermediate results are much more sensitive to $|\mathcal{V}|$. It indicates that *the intermediate results of the edge calculation stage dominated the memory usage during the GNN training*.

Summary

The *high* memory expansion ratio severely restrictes the data scalability of the GNN training. The memory usage mainly comes from the intermediate results of the *edge calculation stage*. Fixing the number of vertices, the memory usage increases *linearly* along with the number of edges. Optimizing the memory usage of the edge calculation stage can significantly reduce the memory expansion ratio. Fixing the GNN structures and the hyper-parameters, increasing the dimension of the input feature vectors can also reduce the memory expansion ratio.



(a) Neighbor sampler



(b) Cluster sampler

FIGURE 19 Sizes of the sampled subgraphs under different batch sizes. Each batch size was sampled 50 times and the average value was reported. The error bar indicates the standard deviation. The batch size is relative to the full graph.

4.4 | Effects of Sampling Techniques on Performance

With the sampling techniques, GNNs can be trained in a mini-batch manner. Each mini-batch updates the model parameters based on a small subgraph sampled from the original input graph. Thus, the training time per batch and the peak memory usage during the training should both decline significantly.

In the implementation in PyG, the GNN model and the dataset resides on the GPU side. To process each epoch, PyG samples the original dataset in the main memory and generates several batches. Each batch is a small subgraph of the dataset. To train on each batch, PyG sends the sampled subgraph to the GPU, calculates the gradients on the subgraph, and updates the model parameters directly on the GPU. With the sampling techniques, the model parameters are updated by a stochastic gradient descent optimizer. PyG conducts the evaluation phase every several epochs (either on the CPU side or the GPU side) to determine whether to stop the training. In this section, the experiments focus on the training phase of each batch.

FIGURE 19 shows how the size of the sampled subgraph changes with the batch size. For the neighbor sampler, the relative batch size is the proportion of the sampled vertices of the last GNN layer in $|\mathcal{V}|$. For the cluster sampler, the relative batch size is the proportion of the sampled partitions to all partitions of the graph. The neighbor sampler is very sensitive to the batch size. As the batch size increases, the size of the sampled subgraph first increases quickly and then stabilizes. The cluster sampler is much less sensitive compared to the neighbor sampler. The number of vertices and the average degree of the sampled subgraphs increases linearly with the batch size.

It is worth noting that the average degree of the sampled subgraph is *much lower* than the average degree of the whole graph, especially when the relative batch size is low. Taking the neighbor sampler with the relative batch size of 6% as an example, the average degree of the `amp` dataset is 31.1, but the average degree of the sampled subgraph is only 5.8. For the cluster sampler, the average degree is 3.0. **FIGURE** 20 compares the degree distribution of the sampled subgraphs with the original graph. The slopes of the curves are similar. It indicates that the sampled subgraphs still follow the power-law degree distribution. However, there are much less vertices in the sampled subgraphs, significantly lowering the average degrees. According to the experimental results in Section 4.2, if the average degree becomes lower, the proportion of the training time spent on the vertex calculation stage will become higher, especially for GGNN.

To find out the performance bottleneck with the sampling techniques, we decompose the training time per batch into three phases: *sampling* on the CPU, *transferring* sampled subgraphs from the CPU to the GPU and *training* with the subgraphs on the

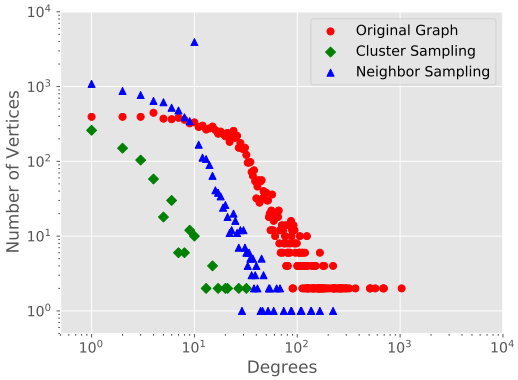
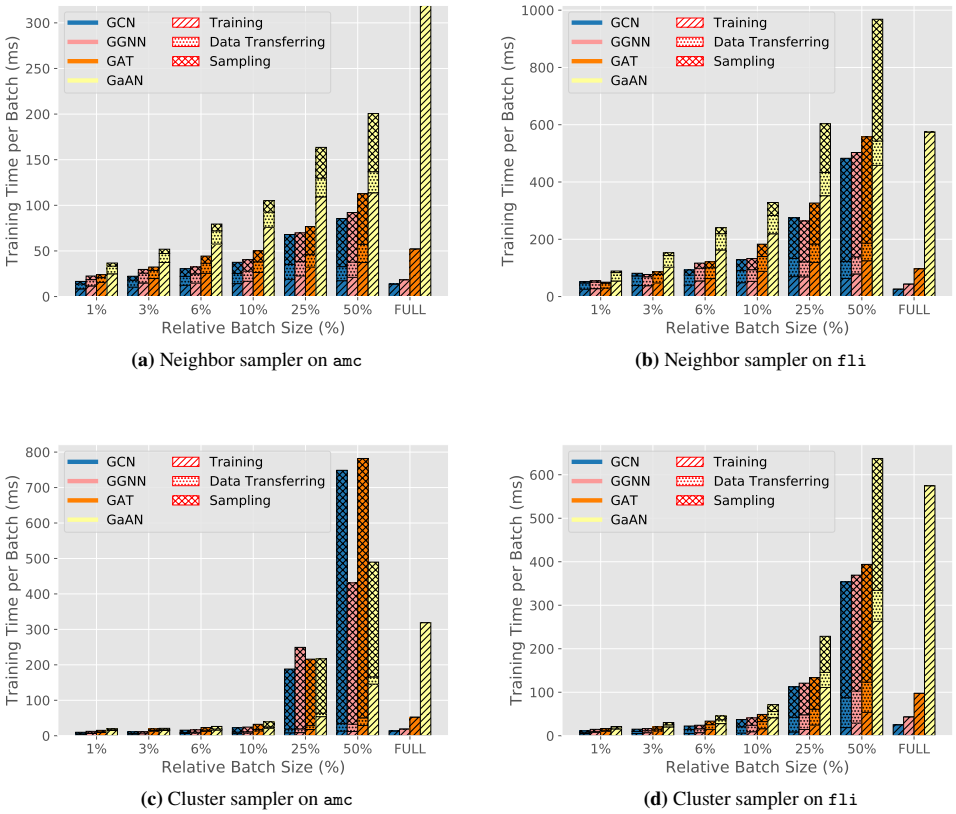


FIGURE 20 Vertex degree distribution of the sampled subgraph (relative batch size: 6%) and the original graph. Dataset:amp.



(a) Neighbor sampler on amc

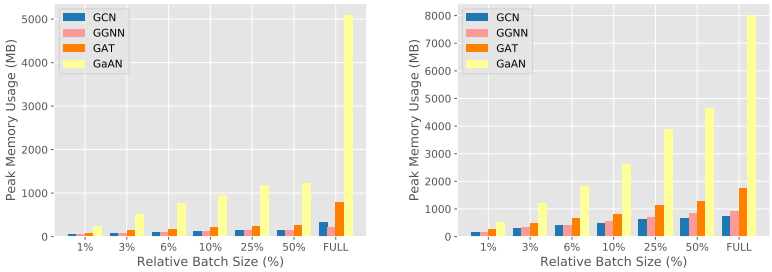
(b) Neighbor sampler on fli

(c) Cluster sampler on amc

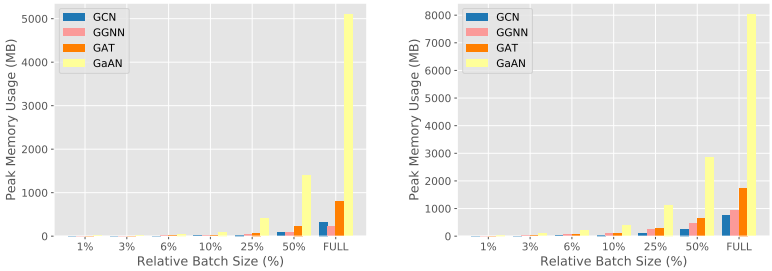
(d) Cluster sampler on fli

FIGURE 21 Training time per batch breakdown. FULL means that the full graph participates in the training.

GPU. **FIGURE** 21 shows the time breakdown of the four GNNs under different relative batch sizes. For the neighbor sampler, the sampling technique reduces the training time per batch only when the batch size is very small. When the batch becomes bigger, the sampling and the data transferring phases introduce noticeable overheads, making the training time exceed the full-batch training. For the clustering sampler, the sampled subgraph is smaller than the neighbor sampler under the same relative batch size. The reduction in the training time is more obvious than the neighbor sampler. However, the overheads increase quickly as the relative batch size increase. The training time under the 25% relative batch size already exceede the time of full-batch training. The experimental results indicate that the current implementation of the sampling techniques in PyG is inefficient.



(a) Neighbor sampler on `amc` **(b)** Neighbor sampler on `fli`



(c) Cluster sampler on `amc` **(d)** Cluster sampler on `fli`

FIGURE 22 Peak memory usage under different batch sizes. FULL means the full graph participates in the training.

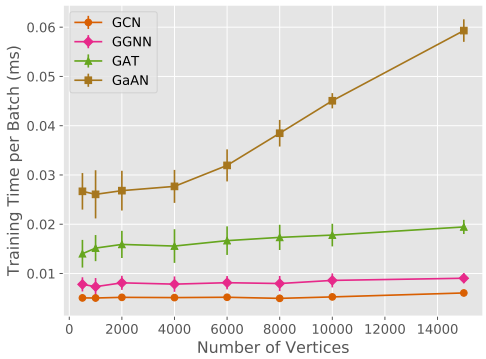


FIGURE 23 Training time per epoch on small random graphs. For each number of vertices, we generated 50 random R-MAT graphs with the average degree of 4.0 and reported the average training time per epoch (without the evaluation phase). The error bar indicates the standard deviation.

When the batch size is slightly big, more than 50% of the time has been spent on sampling and data transferring. *The sampling techniques are only efficient under small batch sizes.*

The main advantage of the sampling technique is *reducing the peak memory usage* during the training. **FIGURE** 22 shows the memory usage under different batch sizes. The peak memory usage declines significantly even under big batch sizes. The sampling techniques make training GNNs on big graphs possible for GPUs.

The disadvantage of the sampling technique is wasting GPU resources. As the sampling techniques are only effective under small batch sizes, the sampled subgraphs will be very small in those cases. They cannot make full use of the computing power of a GPU. To simulate the situation, we generated random graphs with few vertices and measured the training time per epoch in **FIGURE** 23. As the number of vertices increases, the training time is almost unchanged except for GaAN. The training time of GaAN increases only with $|\mathcal{V}| \geq 4000$.

Summary

The sampled subgraphs have lower average degrees than the whole graph. With small batch sizes, the sampling techniques can significantly reduce the training time per batch and the peak memory usage during the training. However, small batch sizes cannot make full use of the computing power of a GPU. With big batch sizes, the current implementation of the sampling techniques in PyG is inefficient. The time spent on the sampling phase and the data transferring phase even exceeds the training phase.

5 | INSIGHTS

Through the extensive experiments, we propose the following key findings and suggestions for how to optimize the performance of the GNN training.

1. *The time complexity in **TABLE** 1 and **TABLE** 2 points out the performance bottleneck theoretically.* The experimental results validate the time complexity analysis. The time complexity points out where the bottleneck comes from. Optimization should focus on complex operations in the messaging function ϕ and the vertex updating function γ.

2. *The computational cost of a GNN layer is mainly affected by the dimensions of the input and the output hidden feature vectors.* Theoretically and empirically, the training time and the memory usage of a GNN layer both increase *linearly* with the dimensions of the input/output hidden feature vectors separately. GNNs are friendly to high-dimensional scenarios. Algorithm engineers can use high-dimensional feature vectors to improve the expressive power of a GNN without worrying exponential growth in the training time and memory usage

3. *Performance optimizations should focus on improving the efficiency of the edge calculation stage.* The edge calculation stage is the most time-consuming stage in most GNNs.

    - If the complexity of the messaging function ϕ is high, the implementation of ϕ is critical to performance. Improving its efficiency can significantly reduce the training time. For example, the attention mechanism in GNNs (like GAT and GaAN) requires an extra sub-layer to calculate the attention weight of each edge. Implementing it with the specially optimized basic operators on the GPU is a potential optimization.
    - If the complexity of ϕ is low, the efficiency of the collection step and the aggregation step becomes critical. The existing GNN libraries [12,11,13] already introduce the *fused* operator to improve their efficiency. When the messaging function ϕ is an assignment or a scalar multiplication of the hidden feature vector of the source vertex, the libraries replace the collection, messaging, and aggregation steps with a single fused operator. The fused operator calculates the aggregated vectors directly from the input hidden feature vectors, minimizing the memory footprints and overlapping the memory accessing with computation. In this way, it significantly reduces the training time of GNNs with low edge calculation complexity (like GCN) [16,17]. However, the applicable condition of the fused operator is very restricted. It does not work for ϕ with more complex operations like matrix multiplication. A potential optimization is to develop composite CUDA kernels that can read the input hidden feature vectors and aggregate message vectors on the fly, without materializing the parameter vectors and the message vectors.

4. *The high memory usage caused by the intermediate results of the edge calculation stage limits the data scalability of the GNN training.* The memory expansion ratios of the typical GNNs are very high, making GPUs unable to handle big graphs. One solution is to distribute the dataset among several GPUs and frequently swap parts of the dataset between GPUs and the main memory [13]. Another possible solution [34] comes from the deep neural network training. It only checkpoints key intermediate results during the forward propagation and re-calculates the missing results on demand during the backpropagation. Implementing the checkpoint mechanism in the GNN training is another potential optimization.

5. *Sampling techniques can significantly reduce the training time and memory usage, but its implementation is still inefficient*. The sampling techniques are effective under small batch sizes. Its current implementation brings considerable overheads when the batch size becomes large. Improving the efficiency of the sampling is a potential optimization. The sampled subgraphs are usually small. They cannot make full use of the computing power of a GPU. How to improve the GPU utilization under small batch sizes is another problem to solve. One possible solution is to train multiple batches asynchronously on the same GPU and use the asynchronous stochastic gradient descent to speed up the converge.

6 | RELATED WORK

Survey of GNNs

Zhou et al.[8], Zhang et al.[9] and Wu et al.[10] survey the existing graph neural networks and classify them from an algorithmic view. They summarize the similarities and differences between the architectures of different GNNs. The typical applications of GNNs are also briefly introduced. Those surveys focus on comparing the existing GNNs theoretically, not empirically.

Evaluation of GNNs

Shchur et al.[31] evaluate the accuracy of popular GNNs on the node classification task. Dwived et al.[35] further compare the accuracy of popular GNNs fairly in a controlled environment. Hu et al.[36] propose the open graph benchmark that provides a standard datasets and a standard evaluation workflow. The benchmark makes comparisons between GNNs easily and fairly. Those model evaluation efforts focus on evaluating the accuracy of different GNNs. They provide insightful suggestions to improve accuracy.

From the efficiency aspect, Yan et al.[16] compare the performance characteristics of graph convolutional networks, typical graph processing kernels(like PageRank), and the MLP-based neural networks on GPUs. They provide optimization guidelines for both the software and the hardware developers. Zhang et al.[17] analyze the architectural characteristics of the GNN inference on GPUs under SAGA-NN[13] model. They find that the GNN inference has no fixed performance bottleneck and all components deserve to optimize. These two efforts focus on the *inference* phase of GNNs and they investigate the potential optimizations mainly from an architectural view. In this work, our target is to find out the performance bottleneck in the training phase from a system view. We consider the performance bottleneck in both time and memory usage. We also evaluate the effects of the sampling techniques. Our work and the related evaluation[16,17] form a complementary study on the efficiency issue of GNNs.

Libraries/Systems of GNNs

PyG[11] and DGL[12] both adopt the message-passing model as the underlying programming model for GNNs and support training big datasets with the sampling techniques. PyG[11] is built upon PyTorch and it uses optimized CUDA kernels for GNNs to achieve high performance. DGL[12] provides a group of high-level user APIs and supports training GNNs with a variety of backends (TensorFlow, MXNet, and PyTorch) transparently. It also supports LSTM as the aggregation functions. NeuGraph[13] proposes a new programming model *SAGA-NN* for GNNs. It focuses on training big datasets efficiently without sampling. It partitions the dataset sophisticatedly, schedules the training tasks among multiple GPUs, and swaps the data among GPUs and the host asynchronously. AliGraph[14] targets at training GNNs on big attributed heterogeneous graphs that are common in e-commerce platforms. The graphs are partitioned among multiple nodes in a cluster and AliGraph trains GNNs on the graphs in a distributed way with system optimizations. PGL[15] is another graph learning framework from Baidu based on the PaddlePaddle platform.

7 | CONCLUSION

In this work, we systematically explore the performance bottleneck in graph neural network training. We model the existing GNNs with the message-passing framework. We classify the GNNs according to their edge and vertex calculation complexities to select four typical GNNs for evaluation. The experimental results validate our complexity analysis. Fixing other hyper-parameters, the training time and the memory usage increase linearly with each hyper-parameter of the four GNNs. To find out the performance bottleneck in the training time, we decompose the training time per epoch on different levels. The training time breakdown analysis indicates that the edge calculation stage and its related basic operators are the performance bottleneck for most GNNs. Moreover, the intermediate results produced by the edge calculation stage cause high memory usage, limiting the data scalability. Adopting sampling techniques can reduce the training time and the memory usage significantly. However, the current implementation of the sampling techniques in PyG brings considerable sampling overheads. The small sampled sub-graphs cannot make full use of the computing power of a GPU card either. Our analysis indicates that the edge calculation stage should be the main target of optimizations. Reducing its memory usage and improving its efficiency can significantly improve the performance of the GNN training. Based on the analysis, we propose several potential optimizations for the GNN frameworks. We believe that our analysis can help developers to have a better understanding of the characteristics of GNN training.

ACKNOWLEDGEMENTS

This work is funded in part by National Key R&D Program of China [grant number 2019YFC1711000]; China NSF Grants [grant number U1811461]; Jiangsu Province Industry Support Program [grant number BE2017155]; Natural Science Foundation of Jiangsu Province [grant number BK20170651]; Collaborative Innovation Center of Novel Software Technology and Industrialization; and the program B for Outstanding PhD candidate of Nanjing University.

References

1. Kipf TN, Welling M. Semi-Supervised Classification with Graph Convolutional Networks. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net; 2017.

2. Defferrard M, Bresson X, Vandergheynst P. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In: Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain. ; 2016: 3837–3845.

3. Li R, Wang S, Zhu F, Huang J. Adaptive Graph Convolutional Neural Networks. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018. ; 2018: 3546–3553.

4. Li Y, Tarlow D, Brockschmidt M, Zemel RS. Gated Graph Sequence Neural Networks. In: 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings. ; 2016.

5. Hamilton WL, Ying Z, Leskovec J. Inductive Representation Learning on Large Graphs. In: Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA. ; 2017: 1024–1034.

6. Velickovic P, Cucurull G, Casanova A, Romero A, Liò P, Bengio Y. Graph Attention Networks. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. ; 2018.

7. Zhang J, Shi X, Xie J, Ma H, King I, Yeung D. GaAN: Gated Attention Networks for Learning on Large and Spatiotemporal Graphs. In: Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018, Monterey, California, USA, August 6-10, 2018. ; 2018: 339–349.

8. Zhou J, Cui G, Zhang Z, et al. Graph Neural Networks: A Review of Methods and Applications. https://arxiv.org/abs/1812.08434; 2018.

9. Zhang Z, Cui P, Zhu W. Deep Learning on Graphs: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 2020; (Early Access): 1–1. doi: 10.1109/TKDE.2020.2981333

10. Wu Z, Pan S, Chen F, Long G, Zhang C, Yu PS. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 2020; (Early Access): 1-21. doi: 10.1109/TNNLS.2020.2978386

11. Fey M, Lenssen JE. Fast Graph Representation Learning with PyTorch Geometric. https://pytorch-geometric.readthedocs.io/; 2019.

12. Wang M, Yu L, Zheng D, et al. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. https://www.dgl.ai/; 2019.

13. Ma L, Yang Z, Miao Y, et al. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In: 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019. ; 2019: 443–458.

14. Zhu R, Zhao K, Yang H, et al. AliGraph: A Comprehensive Graph Neural Network Platform. *Proc. VLDB Endow.* 2019; 12(12): 2094–2105. doi: 10.14778/3352063.3352127

15. Baidu . Paddle Graph Learning. https://github.com/PaddlePaddle/PGL; .

16. Yan M, Chen Z, Deng L, et al. Characterizing and Understanding GCNs on GPU. *IEEE Computer Architecture Letters* 2020; 19(1): 22-25.

17. Zhang Z, Leng J, Ma L, Miao Y, Li C, Guo M. Architectural Implications of Graph Neural Networks. *IEEE Computer Architecture Letters* 2020; 19(1): 59–62.

18. Perozzi B, Al-Rfou R, Skiena S. DeepWalk: online learning of social representations. In: The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014. ; 2014: 701–710

19. Grover A, Leskovec J. node2vec: Scalable Feature Learning for Networks. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016. ; 2016: 855–864

20. Gilmer J, Schoenholz SS, Riley PF, Vinyals O, Dahl GE. Neural Message Passing for Quantum Chemistry. In: . 70 of *Proceedings of Machine Learning Research*. Proceedings of the 34th International Conference on Machine Learning. PMLR; 2017; International Convention Centre, Sydney, Australia: 1263–1272.

21. Duvenaud D, Maclaurin D, Aguilera-Iparraguirre J, et al. Convolutional Networks on Graphs for Learning Molecular Fingerprints. In: Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada. ; 2015: 2224–2232.

22. Dai H, Kozareva Z, Dai B, Smola AJ, Song L. Learning Steady-States of Iterative Algorithms over Graphs. In: Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. ; 2018: 1114–1122.

23. Chiang WL, Liu X, Si S, Li Y, Bengio S, Hsieh CJ. 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. ACM; 2019: 257—266

24. Ying R, He R, Chen K, Eksombatchai P, Hamilton WL, 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, KDD 2018, London, UK, August 19-23, 2018. ; 2018: 974–983

25. Chen J, Ma T, Xiao C. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. ; 2018.

26. Chen J, Zhu J, Song L. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In: Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. ; 2018: 941–949.

27. Huang W, Zhang T, Rong Y, Huang J. Adaptive Sampling Towards Fast Graph Representation Learning. In: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada. ; 2018: 4563–4572.

28. Zeng H, Zhou H, Srivastava A, Kannan R, Prasanna VK. Accurate, Efficient and Scalable Graph Embedding. In: 2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019. ; 2019: 462–471

29. Zeng H, Zhou H, Srivastava A, Kannan R, Prasanna VK. GraphSAINT: Graph Sampling Based Inductive Learning Method. In: 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. OpenReview.net; 2020.

30. Yang Z, Cohen WW, Salakhutdinov R. Revisiting Semi-Supervised Learning with Graph Embeddings. In: . 48 of *JMLR Workshop and Conference Proceedings*. Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016. JMLR.org; 2016: 40–48.

31. Shchur O, Mumme M, Bojchevski A, Günnemann S. Pitfalls of Graph Neural Network Evaluation. *CoRR* 2018; abs/1811.05868.

32. Chakrabarti D, Zhan Y, Faloutsos C. R-MAT: A Recursive Model for Graph Mining. In: Proceedings of the 2004 SIAM International Conference on Data Mining. : 442-446

33. Yang J, Leskovec J. Defining and Evaluating Network Communities Based on Ground-Truth. In: 12th IEEE International Conference on Data Mining, ICDM 2012, Brussels, Belgium, December 10-13, 2012. IEEE Computer Society; 2012: 745–754

34. Chen T, Xu B, Zhang C, Guestrin C. Training Deep Nets with Sublinear Memory Cost. https://arxiv.org/abs/1604.06174; 2016.

35. Dwivedi VP, Joshi CK, Laurent T, Bengio Y, Bresson X. Benchmarking Graph Neural Networks.; 2020.

36. Hu W, Fey M, Zitnik M, et al. Open Graph Benchmark: Datasets for Machine Learning on Graphs. https://arxiv.org/abs/2005.00687; 2020.